A Hierarchical Component Model with Interaction Protocols

Sébastien Pavel

UNIVERSITÉ DE NANTES
FACULTÉ DES SCIENCES ET DES TECHNIQUES

ÉCOLE DOCTORALE
SCIENCES ET TECHNOLOGIES
DE L'INFORMATION ET DES MATÉRIAUX

Année 2008 N° attribué par la bibliothèque

ED 503-037

Un modèle de composants hiérarchiques avec protocoles d'interaction

A Hierarchical Component Model with Interaction Protocols

THÈSE DE DOCTORAT

Spécialité Informatique
Présenté et soutenue publiquement par

Sebastian Pavel

Le 21 Octobre 2008
à l'École Nationale Supérieure des
Techniques Industrielles et des Mines de Nantes

devant la commission d'examen composée de :

Rapporteurs	:	Laurence Duchien	Professeur des Universités, Lille I
		Antoine Beugnard	Enseignant-Chercheur, HDR, ENST de Bretagne
Examinateurs	:	Frédéric Besson	Chargé de Recherche, INRIA
	:	Mourad Oussalah	Professeur des Universités, Nantes
	:	Jacques Noyé	Maître-Assistant, École des Mines de Nantes
Directeur de thèse	:	Jean-Claude Royer	Professeur, École des Mines de Nantes

Laboratoire d'accueil : Laboratoire d'Informatique de Nantes-Atlantique (UMR 6241)
Équipe d'accueil : École des Mines de Nantes - Objets, Aspects et Composants (OBASCO)

Résumé

L'utilisation et la gestion des composants sont au coeur des nouvelles architectures logicielles. Les composants représentent les briques de bases des logiciels. Les efforts de recherche actuels se concentrent sur l'élaboration de modèles à base de composants qui intègrent des propriétés importantes comme, par exemple, la description et l'intégration des composants avec des comportements explicites (protocoles d'interaction). Ce sont ces descriptions plus complètes que les interfaces classiques (les points d'entrée et de sortie), qui ouvrent la voie vers la correction des assemblages.

Comme aboutissement des travaux de cette thèse, nous proposons un modèle de composants qui utilise des Systèmes de Transitions Symboliques (STSs) pour décrire les comportements des composants. Les composants de notre modèle sont des boîtes noires communicant exclusivement par l'intermédiaire de leurs interfaces étendues avec des protocoles d'interaction. Le modèle spécifie aussi les règles de compatibilité, les algorithmes de vérification des assemblages des composants et de la substitution et un langage de description des composants.

Nous proposons une implémentation dans le langage Java en suivant une approche générative ou le code Java est généré à partir des descriptions des composants de haut niveau. Le code est donc garanti a être conforme à la spécification.

Mots-clés : modèle de composants, protocoles d'interaction, systèmes de transitions symboliques (STS), génération de code, Java.

Abstract

Title : A Hierarchical Component Model with Interaction Protocols

The component-based Software Engineering (CBSE) represents an important trend in the development of software architectures. The components are the core of the software applications and the recent efforts concentrate on conceiving component models integrating important properties as for example explicit interaction protocols. Interaction protocols allow a component to publish its behavior in terms of message emission and receipt. Thus, component assemblages can be more effective than in the traditional approach where only static interfaces (input and output) are available.

In this thesis, we propose a component model considering black-box components integrating interaction protocols described as Symbolic Transition Systems (STSs). STSs allow the description and verification of complex protocols while dealing with the state explosion problem for example. The model also specifies the compatibility rules, the substitution and the assemblage verification algorithms and a component description language. We propose a generative approach on implementing our model in the Java programming language. The Java code is automatically generated starting from high level component descriptions assuring that the component code is conforming with its specification.

Keywords : component models, interaction protocols, symbolic transition systems (STS), code generation, Java.

Remerciements

Je remercie tout d'abord les membres du jury qui ont accepté d'évaluer mon travail. Merci aux rapporteurs de cette thèse, Laurence Duchien professeur à l'Universités de Lille I qui m'a soutenu pendant les quelques années d'interaction dans le cadre du projet ACI Dispo et à Antoine Beugnard, enseignant-chercheur à ENST de Bretagne qui, m'a beaucoup influencé par son travail dans le domaine des composants logiciels. Merci aux autres membres pour l'intérêt dans mon travail et pour l'honneur qui me font de participer dans ce jury.

Merci à Jean-Claude Royer, mon directeur de thèse. Sans toi Jean-Claude, je n'ai surement pas pu arriver si loi. Ta confiance et ton énergie m'ont beaucoup aidé à surmonter les difficultés et suivre le bon chemin. Merci aussi à Jacques Noyé, mon co-encadrant pour son rigueur et sa minutie. C'est la poursuite de la perfection qui te définit Jacques et je ne peut-être que reconnaissant pour tes conseils.

Merci aux membres du département informatique de l'Ecole de Mines de Nantes qui m'ont accueilli parmi eux avec un très grand merci à Pierre Cointe, ancien chef du service, qui a tout fait pour que je puisse intégrer l'EMN dans les meilleures conditions. Merci à Remy Douence et à Mario Südholt pour leur soutien et leur bonne parole dans des moments difficiles.

Merci aussi à toi Thierry Petit pour ton courage et ta volonté qui me donne la force de continuer à me battre et aussi à profiter des choses simples.

Merci aussi aux membres du projet ACI Dispo avec lesquels j'ai pu échanger beaucoup sur la recherche et qui ont représenté pour moi un soutien important tant sur le plan personnel que professionnel.

Merci aux anciens thésards parmi lesquels j'ai fait mes premiers pas dans le monde de la recherche : Yann-Gaël Guéhéneuc, Marc Ségura-Devillechaise, Gustavo Bobeff, Simon Denier et tous les autres.

Merci à ma famille et à mes amis qui ont su me soutenir et aussi me supporter dans beaucoup des moments très difficiles.

Sans toutes ces personnes, ce long travail n'aurait pas pu être amené au bout. Un grand merci à tous.

Table des matières

Table des figures

Première partie

Résumé en Francais

Chapitre 1

Résumé

Dans les années soixante il est devenu évident que la façon de produire les logiciels avait besoin d'être reconsidérée. Les logiciels étaient développés sur demande et très souvent sans rien capitaliser des développements précédents. Les premiers désavantages sont apparus au fur et à mesure que la demande a augmenté : la qualité était souvent médiocre et les coûts d'exploitation et de maintenance très élevés. De plus, la dynamique du marché a imposé que les applications logicielles soient rapidement modifiables pendant la phase de développement et aussi de plus en plus complexes, évolutives ou adaptables. Les pressions exercées par le marché et les clients avaient également un impact négatif sur la documentation : le temps alloué à celle-ci était insuffisant et le résultat souvent un produit incomplet voire même incorrect.

L'ingénierie des composants (*Component-Based Software Engineering* [Heineman 01a] en anglais) propose des réponses pour palier aux difficultés présentées ci-dessus. L'ingénierie des composants considère les composants logiciels comme la brique de base de toute construction logicielle. La notion de composant logiciel a été mentionnée pour la première fois par McIlroy [McIlroy 68] dans son discours à la conférence de l'OTAN en 1968. L'idée de brique logicielle comme solution aux problèmes de développement a émergé à cette époque. Les composants logiciels sont liés aux modules [Wirth 77, Mitchell 79] et à la programmation par objets (*Object-Oriented Programming* en anglais).

Les composants logiciels sont liés aux modules par le fait que la technologie des composants amène à mettre en place des solutions modulaires. Ceci dit, si la modularité est un prérequis, des règles supplémentaires (comme par exemple l'indépendance des composants et le contrôle explicite des dépendances entre les composants) sont nécessaires pour former des composants plutôt que des modules.

Á la fin des années 90, le constat a été fait que l'approche de la programmation par objets était insuffisante pour construire des logiciels de grande taille et d'améliorer fortement la réutilisation. En contraste avec les mécanismes à objets qui introduisent un couplage fort entre les classes de base et les classes dérivées (problème connu sous le nom de classe de base fragile [Szyperski 96, Mikhajlov 98]), l'approche par composants fournit une meilleure séparation entre l'implémentation d'un composant et son interface. Chaque composant est une entité indivisible de type boîte noire qui peut être composée avec d'autres composants

et peut être déployée indépendamment des autres composants. La construction des logiciels est réalisée par assemblage de ces composants. Dans cette approche, l'interface du composant est d'une importance primordiale car elle définit un contrat entre le composant et son environnement. L'interface décrit d'une manière très explicite ce que le composant fournit et ce qu'il requiert de son environnement dans le but de réaliser sa finalité dans l'architecture de l'application.

Les caractéristiques particulières d'un composant sont décrites par les *modèles de composants*. Un *modèle de composants* décrit ce qu'un composant est (par définition de ses parties constitutives) et spécifie comment les composants peuvent être assemblés en suivant des règles de composition. Le modèle de composant décrit également le cycle de vie du composant et les rôles associés aux différents acteurs du développement logiciel et de l'exploitation des applications.

Les composants sont des entités qui collaborent en échangeant des messages pour coordonner des actions ou seulement échanger des données. Pour une interaction réussie, les composants doivent se conformer à une forme de contrat d'interaction. Les protocoles d'interaction décrivent généralement le comportement d'une entité (objet, acteur, agent ou composant) en termes de séquences de messages qui peuvent être échangés par les entités pour réaliser le comportement global attendu de l'application.

L'ingénierie des composants considère les protocoles d'interaction au niveau des interfaces des composants. Au début, les interfaces ont seulement décrit les signatures des services fournis et requis par le composant. Plus récemment, les protocoles d'interaction ont fait leur apparition dans des modèles de composants plus élaborés où le comportement dynamique est explicitement décrit. Ceci permet une meilleure documentation du comportement visible du composant et une meilleure intégration dans une architecture logicielle, mais ouvre également la voie à des outils et des techniques de vérification pour par exemple vérifier, *a priori*, la validité d'un assemblage.

En découplant l'interface de son implémentation, une question importante se pose : comment assurer que l'implémentation est *conforme* (ou *cohérente*) avec l'interface. Si dans une approche purement syntaxique où seulement les signatures des services sont spécifiées par une interface, le test de conformité est simple. Dans le cas où le comportement dynamique est également spécifié, l'implémentation se révèle plus difficile à réaliser. Beaucoup de modèles de composants de l'état de l'art actuel ne prennent pas en compte ce problème. D'autres se basent sur des techniques d'analyse de code pour assurer la conformité du code. Les langages de programmation de composants (*component programming language* en anglais) considèrent ce problème en permettant la programmation avec des composants plutôt que des objets. Le code est conforme à son interface car la description de l'interface se fait au même niveau que l'implémentation et le compilateur vérifie certaines propriétés importantes comme l'intégrité de la communication qui stipule que les composants ne peuvent communiquer que par des canaux bien spécifiés. Finalement, les techniques génératives assurent par construction que l'implémentation est conforme à la spécification de l'interface.

Les modèles et langages de composants actuels intègrent souvent des concepts trop nombreux ou ils sont trop spécifiques à certains contextes. Il est alors difficile d'analyser les conséquences de l'ajout de nouveaux concepts (comme les protocoles d'interaction) ou

de nouvelles fonctionnalités dans un modèle de composants existant. Les objectives de cette thèse sont donc de proposer un modèle de composants qui intègre des protocoles d'interaction explicites donnés sous la forme des systèmes de transitions symboliques.

Suite à des expérimentations avec la programmation par composants (voir [Pavel 04] pour une proposition générative de développement des composants en ArchJava [Aldrich 02b]) et avec l'élaboration d'un modèle de composants qui intègre des protocoles d'interaction et des communications asynchrones [Pavel 05a, Pavel 05b, Noyé 05], nous proposons un modèle de composants appelé CwSTS (*Components with Symbolic Transition Systems* en anglais). Des détails de notre proposition sont présentés dans la section 1.2 à la page 24.

1.1 Domaine d'étude

1.1.1 L'ingénierie des composants

Les caractéristiques particulières d'un composant sont décrites par les modèles de composants. Un modèle de composants décrit ce qu'est un composant (par la définition de ses parties constitutives) et spécifie la façon dont les composants peuvent être assemblés en suivant des règles de composition. Le modèle de composant décrit également le cycle de vie du composant et les rôles associés aux différents acteurs du développement logiciel et de l'exploitation des applications.

L'état de l'art actuel dans le domaine des modèles de composants n'a pas établi de consensus concernant une définition du terme composant. Plusieurs définitions coexistent [Brown 98, Larsson 00, Heineman 01b, Szyperski 02, Marvie 02]. Malgré ces différences, il est communément admis qu'un composant a au moins les propriétés suivantes :

- il est une unité indépendante de déploiement ;
- son état interne n'est pas visible de l'extérieur ;
- il est une unité de composition.

Szyperski [Szyperski 02] définit un composant logiciel comme une unité de composition avec des interfaces spécifiées contractuellement contenant uniquement des dépendances contextuelles explicites. Il peut être déployé d'une manière indépendante et il fait l'objet de composition par des tiers.

Cette définition a plusieurs implications. Premièrement, un composant a besoin d'encapsuler son implémentation et peut communiquer avec son environnent uniquement par l'intermédiaire des interfaces explicites. Ensuite, le composant est préparé pour être composé avec d'autres composants pour obtenir des applications logicielles. Cette définition implique aussi l'existence d'un cycle de vie associé au composant qui, d'une manière traditionnelle, consiste en quatre phases : la création, l'assemblage, le déploiement, et l'exécution. Une notion moins explicite est celle d'environnement où le composant peut se trouver dans ses différentes phases du cycle de vie.

1.1.1.1 Les contrats d'interfaces

La précédente définition des composants implique le fait que les dépendances contextuelles des composants soient explicites. Ceci est réalisé par l'intermédiaire des interfaces des composants. Les interfaces sont définies comme un ensemble de services que le composant fournit ou requiert de son environnement. Elles sont vues comme des contrats entre le composant et son environnement d'exécution. Antoine Beugnard [Beugnard 99] propose une taxonomie des contrats qui est liée aux interfaces de composants. Conformément à cette taxonomie, il y a quatre niveaux de contrats :

- *basique* ;
- *comportemental* ;
- *de synchronisation* ;
- *quantitatif* ;

Le niveau *basique* considère les propriétés syntaxiques : des noms, types de paramètres, types de retour et les exceptions des messages échangés par le composant. Le niveau *comportemental* considère des propriétés qui peuvent être spécifiées par des préconditions, postconditions et invariants. Ces propriétés sont liées à une opération et spécifient les types de paramètres et les dépendances entre les valeurs de ces paramètres. Le niveau *de synchronisation* considère les propriétés concernant les interactions des composants. Ces propriétés ne peuvent être exprimées seulement par l'intermédiaire de pré- et postconditions, car elles doivent décrire une séquence de pas à exécuter par le composant. Le niveau *quantitatif* correspond aux propriétés dites *non-fonctionnelles* comme la qualité de service, la gestion des ressources et le temps de réponse. Ces types de contrats sont les plus difficiles à exprimer et analyser car ils dépendent des propriétés qui ne sont pas connues au moment de la description de l'architecture.

1.1.1.2 Le mécanisme de composition

La *composition* est un mécanisme permettant la construction de composants complexes (appelés composants *composites*) à partir de composants plus simples (appelés composants *primitifs*). La composition est réalisée en mettant en correspondance les services des composants définis dans leurs interfaces. Plusieurs schémas de composition (*bind*, *export*, *import* et *glue*) sont utilisées pour obtenir des applications exécutables à partir des composants (voir la section 3.2.1.3 à la page 64).

La *compatibilité* est un concept très important qui intervient dans le mécanisme de composition. Indifféremment des schémas de composition utilisés, les services mis en correspondance doivent être *compatibles*. En fonction du niveau de contrats auquel nous nous plaçons, la compatibilité a un sens différent. Les services (et par extrapolation les interfaces) *incompatibles* ne devraient pas être connectés dans une architecture. Une adaptation peut être réalisée dans certaines conditions et par l'intermédiaire d'un *adaptateur*. L'adaptateur est appelé du *code glu* [Cherinka 98] et dans certains cas il faut plus de temps pour le développer que les composants concernés.

1.1.1.3 Modèles de composants dans la recherche et l'industrie

Aujourd'hui, il existe deux catégories de modèles de composants : les modèles académiques et les modèles industriels. Les modèles académiques (voir par exemple [Kenney 95, Bellissard 95, Magee 95, Medvidovic 96, Plašil 98, Flatt 98, Siegel 00, van Ommering 00, Cardone 00, Choppy 01, McDirmid 01, Seco 02, Sreedhar 02, Aldrich 02b, Coupaye 02]) se concentrent sur les concepts clés liés aux composants comme la définition des composants, de leurs interfaces et des mécanismes de composition. Différentes propriétés comme par exemple la substituabilité, la compatibilité et l'adaptation dynamique du comportement des composants sont étudiées. Les modèles industriels se concentrent sur la production, la distribution et l'exécution des applications industrielles. Ils fournissent des solutions comme par exemple la distribution, les transactions, la persistance et la sécurité dans le contexte des applications distribuées.

1.1.1.4 Architectures logicielles

Le standard ANSI/IEEE 1471 de 2000 définit l'*architecture logicielle* [Garlan 94, Bass 98] comme *l'organisation fondamentale d'un système incarnée dans ses composants, leurs relations avec l'environnement et les principes qui guident sa conception et son évolution* [Shaw 96b, Szyperski 02]. Plus spécifiquement, l'architecture logicielle est l'organisation d'un système logiciel comme une collection de composants, de connexions entre les composants et des contraintes sur les modes d'interaction entre les composants.

L'utilisation d'une architecture logicielle peut être faite soit pour la conception et l'implémentation d'un système logiciel individuel, soit en tant que ligne de produits logiciels[1], soit comme une architecture standard utilisée pour un modèle public de composants [Bosch 00]. Les architectures logicielles sont généralement utilisées dans les but de :

1. Réduire le coût de développement.
2. Améliorer la qualité des logiciels en termes de fiabilité, maintenance et utilisation efficace des ressources.
3. Réduire le temps de mise sur le marché.
4. Réduire le coût de la maintenance.

Les *connecteurs logiciels* sont des éléments réutilisables de conception qui encapsulent un style particulier d'interaction entre les composants. Ils sont considérés comme des entités de première classe dans les architectures logicielles [Shaw 96a]. Les connecteurs régissent les règles d'interaction et précisent les mécanismes auxiliaires nécessaires dans ces interactions [Shaw 96b]. Plusieurs catégories de connecteurs existent, par exemple client-serveur, architecture en anneau, *publish-subscribe*, etc. [Mehta 00]. Alors que les connecteurs de communication assurent la transmission des données entre composants, ceux de coordination assurent le transfert de contrôle d'un composant à un autre. Des connecteurs spécifiques peuvent aussi convertir l'interaction fournie par un composant vers l'interaction requise par un autre composant. Finalement, les connecteurs peuvent aussi faciliter la médiation de

1. http ://www.sei.cmu.edu/productlines/

l'interaction entre composants. Des mécanismes comme les optimisations des interactions sont typiquement réalisés par ce type de connecteurs.

1.1.1.5 Les langages de description d'architectures

Les langages de description d'architecture (*Architecture Description Languages* en anglais), représentent une avancée dans le domaine des architectures logicielles. Leur objectif est d'aider dans la structuration et la composition des briques logicielles dans le but d'obtenir des architectures valides. Un langage de description d'architecture est défini comme une notation formelle ou informelle, textuelle ou graphique, qui permet la spécification des architectures logicielles et qui est accompagnée d'outils spécifiques comme présenté dans [Medvidovic 00a]. Le résultat de l'utilisation d'un langage de description d'architecture est une vision abstraite de l'application en termes de composants, connecteurs et configurations. Medvidovic et Taylor [Medvidovic 00b] donnent un cadre de classification et de comparaison des langages de description d'architecture existants. Parmi les langages de description d'architecture actuels, Darwin [Magee 95], Wright [Allen 97] et Rapide [Kenney 95] sont les plus connus.

1.1.1.6 Langages de programmation de composants

La plupart du temps, la vision concrète de l'implémentation est ignorée dans une approche de type langage de description d'architecture. L'analyse architecturale réalisée avec un langage de description d'architecture peut révéler des propriétés importantes, mais celles-ci ne sont pas en général garanties par l'implémentation. Pour permettre une analyse architecturale au niveau du code, le code doit être conforme à l'implémentation [Luckham 95a]. La programmation par composants propose le développement de logiciels en les programmant directement avec des composants plutôt que des objets. Parmi les langages de programmation par composants, ArchJava [Aldrich 02a, Aldrich 02b], Java/A [Baumeister 06] et Jiazzi [McDirmid 01] sont les plus avancés.

1.1.1.7 Le paradigme de la coordination

La coordination est une approche centrée sur la collaboration entre processus. Cette approche considère la programmation des systèmes distribués ou parallèles comme la combinaison de deux activités distinctes : l'activité de calcul proprement dite qui contient les processus impliqués dans la manipulation des données et l'activité de coordination qui est responsable de la communication et de la coopération des processus. Papadopoulos et Arbab [Papadopoulos 98] argumentent le fait que les modèles de coordination peuvent être rangés en deux catégories : les modèles orientés données et les modèles orientés contrôle. Dans la première approche, l'évolution des calculs est contrôlée par les types et les propriétés des données impliquées dans l'activité de coordination. Dans le cas des modèles orientées contrôle (ou orientées processus), les changements dans l'état des processus coordonnés sont réalisés par des événements spécifiques.

La coordination peut-être utilisée comme base pour des modèles de composants. Arbab et al. [Arbab 01], par exemple, propose un modèle de composants qui utilise des *canaux*

mobiles pour une communication anonyme, point à point et qui permette une configuration dynamique des connexions à l'exécution.

La coordination est largement présenté dans la section 3.3.7 à la page 89.

1.1.2 Protocoles d'interaction : formalismes connus

Les entités qui collaborent pour réaliser des actions utilisent très souvent l'échange de messages pour coordonner leurs actions ou pour simplement échanger des données. Pour réaliser une interaction, les entités (objets, agents ou composants) ont besoin de se conformer à un contrat d'interaction. Les protocoles d'interaction décrivent le comportement de l'entité en termes de séquences de messages qui doivent être échangés par l'entité pour participer à la réalisation du comportement global du système. Plusieurs approches coexistent pour formaliser, analyser et implémenter les interactions entre les composants. Les algèbres de processus, les types comportementaux, les machines à états finis et les logiques temporelles sont les formalismes les plus utilisés pour spécifier des protocoles d'interaction.

1.1.2.1 Algèbres de processus

Les algèbres de processus représentent une famille de techniques de spécification particulièrement adaptées à la spécification des systèmes de composants concurrents communicants. Elles décrivent les interactions d'un processus en termes de calcul. Les plus importantes algèbres de processus sont CCS [Milner 89], CSP [Hoare 85] et LOTOS [Brinksma 87] car elles servent aussi de base pour d'autres algèbres de processus comme Pi-calcul [Milner 92, Milner 99] et les agents mobiles [Cardelli 98]. Même si à première vue, CCS, CSP et LOTOS semblent être très similaires (elles se basent sur la notion de processus composé d'actions atomiques avec des sémantiques opérationnelles et beaucoup d'opérateurs communs), des différences subtiles existent notamment au niveau des sémantiques des opérateurs qui sont souvent identiques d'un point de vue syntaxique.

1.1.2.2 Types comportementaux

Des travaux récents présents dans [Nierstrasz 93, Puntigam 96] défendent la thèse de l'extension des théories de types classiques pour permettre la description des propriétés dynamiques des objets dans le cadre d'un modèle comportemental. Ces nouveaux types sont appelés *types comportementaux* car ils ne spécifient pas seulement l'ensemble de messages que les entités vont échanger mais aussi, le plus important, des contraintes sur les séquences acceptables de ces messages.

1.1.2.3 Machines à états finis

Les protocoles d'interaction peuvent être spécifiés comme des *processus réguliers*, c.-à-d. des processus avec un nombre fini d'états ou de comportements. Les formalismes basés sur les machines à états finis (comme les systèmes de transitions étiquetées [Keller 76], systèmes de transitions symboliques, automates à entrées et sorties [Lynch 87] et les diagrammes d'états UML [UML 03]) permettent la description des comportements des systèmes à un

certain niveau d'abstraction. Les machines à états finis décrivent l'ensemble des traces possibles qu'un composant peut produire en interagissant avec ses partenaires.

1.1.2.4 Les logiques temporelles

Les logiques temporelles sont utilisées pour décrire et raisonner sur les propositions qualifiées en termes de temps. En pratique, les logiques temporelles permettent la spécification et la vérification des propriétés comme la sûreté, la vivacité, l'équité et l'absence de blocage. Les plus connues des logiques temporelles sont PLTL [Clarke 81] et CTL [Pnueli 81]. L'expression des protocoles d'interaction est réalisée directement en utilisant les opérateurs de base spécifiés par [Clarke 00], mais compte tenu du fait que les logiques temporelles sont très coûteuses en termes de verification, il y a très peu de logiques temporelles réellement utilisables.

1.1.2.5 Langages et modèles à composants avec protocoles d'interaction

Des nombreux modèles et langages de composants intègrent les formalismes présentés antérieurement pour décrire des protocoles d'interaction. Par exemple, Darwin [Magee 95] permet la description des protocoles d'interaction sous la forme de LTS et la vérification des propriétés est réalisée en utilisant FSP (*Finite State Processes* en anglais). Une autre approche basée sur des automates [Barros 05] permet la description des protocoles d'interaction dans le modèle de composants Fractal [Coupaye 02]. Au niveau des types comportementaux, les travaux de Yellin et Strom [Yellin 97] sont considérés comme une référence dans le domaine de l'intégration des descriptions des comportements au niveau des interfaces des composants. Le modèle de composants CwEP [Farias 03] et les travaux de Cyril Carrez [Carrez 03] sont d'autres avancées importantes dans ce domaine. Arbab *et al.* [Arbab 02] propose un modèle de coordination pour les systèmes à base de composants qui est basé sur la notion de *canaux mobiles*. Les canaux mobiles permettent des communications point à point anonymes avec la possibilité de reconfigurer dynamiquement les connexions. En dehors des types de formalismes présentés, le modèle de composants SOFA [Plašil 98], l'environnement visuel PACOSUITE [Vanderperren 03] et le langage de spécification des comportements MIDAS [Pryce 98] représentent des approches alternatives à la spécification et l'implémentation des protocoles d'interaction.

1.2 Objectifs et propositions

Les modèles et langages de composants actuels intègrent souvent des concepts trop nombreux ou ils sont trop spécifiques à certains contextes. Il est alors difficile d'analyser les conséquences de l'ajout de nouveaux concepts (comme les protocoles d'interaction) ou de nouvelles fonctionnalités dans un modèle de composants existant.

Dans cette thèse nos objectifs sont de proposer un nouveau modèle de composants qui intègre des protocoles d'interaction explicites donnés sous la forme de systèmes de transitions symboliques (STS). Les STS sont une extension des LTS (*Labelled Transition Systems*

en anglais) où les transitions sont dites *symboliques*. Le symbolisme se manifeste par la présence de données échangées, de complexité quelconque, et l'utilisation de conditions pour le déclenchement des transitions (concept classique de *garde*).

Les protocoles d'interaction, qu'ils soient basés sur des STS ou sur d'autres formalismes, ont comme but principal une meilleure intégration des composants logiciels dans une architecture. Les règles spécifiées par les protocoles d'interaction sont utilisées pour détecter de possibles incompatibilités entre les composants. Une fois la propriété de compatibilité prouvée, les architectures à base de composants bénéficient d'une meilleure qualité, indispensable surtout dans certains contextes ou domaines d'activité.

L'intérêt d'utiliser des formalismes basés sur les automates à états finis comme les LTS ou les STS réside dans les techniques automatiques de vérification et dans les outils correspondants. La lisibilité et la compacité de ce type de formalisme sont souvent avancées comme un des avantages pour leur utilisation. En contraste avec les LTSs, les transitions des STSs décrivent des classes d'opérations possibles. Les transitions sont paramétrées avec des paramètres formels d'entrée et peuvent être gardées avec des conditions (opérations booléennes) paramétrées. Le principal avantage lié à l'utilisation des STS consiste dans leur format compact, la lisibilité et l'expressivité. Les descriptions STS sont plus faciles à utiliser par un ingénieur classique ce qui permet de les utiliser plus souvent en pratique.

Nous souhaitons également avoir un langage de description de composants qui facilite l'écriture et la compréhension des composants par les développeurs. Notre volonté est d'orienter nos travaux vers une intégration facile dans le monde du développement des applications à base de composants. Dans cette optique nous pensons que les STS présentent des avantages notables par rapport à d'autres formalismes : notamment leur lisibilité et compacité.

Un des problèmes soulevés par les modèles et langages de composants qui intègrent des protocoles d'interaction est la conformité entre la spécification et l'implémentation. Pour pallier à ce problème nous proposons une approche générative en ce qui concerne l'implémentation. En effet, la génération du code peut assurer *par construction* la conformité de l'implémentation à sa spécification conceptuelle.

Sur la base des travaux préliminaires présentés précédemment, nous avons développé un modèle de composants que nous appelons CwSTS (*Components with Symbolic Transition Systems* en anglais). CwSTS est conçu comme un modèle de composants où les communications sont réalisées exclusivement par l'intermédiaire d'une seule interface (composée de deux types d'interfaces). L'interface décrit les services fournis et les services requis par le composant. En plus de cette interface, que nous appelons *interface structurelle* car elle ne définit que les signatures des services, CwSTS propose une interface complémentaire décrivant le protocole d'interaction du composant. Cette interface est appelée *interface comportementale* car elle spécifie le comportement du composant en terme des messages envoyés et reçus et de leur ordonnancement logique. Notre modèle est basé sur une vision hiérarchique de la composition des composants. Deux ou plusieurs composants peuvent êtres composés dans un composant unique appelé composant composite. Le composant composite adhère aux mêmes règles qu'un composant primitif et en plus il peut être utilisé comme s'il était un composant primitif dans des compositions futures. Ceci corresponds à l'idée du patron de conception Composite [Gamma 95].

CwSTs-IDL est le langage de description d'interface (*Interface Description Language* en anglais) que nous proposons avec notre modèle. Il est utilisé pour décrire les interfaces structurelles des composants et aussi les interfaces comportementales. Les interfaces comportementales sont décrites avec un langage formel sous-jacent que nous appelons SFSP (*Symbolic Finite State Processes* en anglais). SFSP est inspiré du langage appelé FSP (*Finite State Processes* en anglais) [Magee 99] mais nous ne retenons que les transitions, le choix et la récursivité. De plus, les actions sont paramétrables avec des types de données quelconques dans SFSP, contrairement à FSP.

1.2.1 Le modèle CwSTS

Le modèle CwSTS est un modèle simple conçu dans le but de pouvoir analyser l'intégration des protocoles d'interaction donnés sous la forme des STS. De plus, nous adoptons une approche générative quant à la construction des composants et nous proposons une implémentation dite *distribuée* (en rapport avec la réalisation du comportement global) des architectures.

Dans ce but nos considérons que dans notre modèle :

- chaque composant expose une seule et unique interface (contenant les informations structurelles et comportementales) ;
- les communications entre les composants sont de type point à point ;
- l'envoie et le réception d'un message se réalisent d'une manière ; synchrone (le client qui envoie le message est bloqué tant que la réception n'as pas eu lieu) mais il n'y a pas de restriction quant à l'exécution proprement dite du message (donc, en fonction de l'implémentation du composant, l"exécution du message peut être synchrone ou asynchrone relatif au composant qui envoie le message) ;
- les architectures sont statiques, c.a.d. qu'il n'y a pas de possibilité de réconfiguration de l'architecture en cours d'exécution ;

Ces hypothèses peuvent paraître limitatives mais nous avons opté pour la simplicité dans notre approche. Néanmoins, certains extensions de ce modèle pour permettre des communications de groupe, notamment, sont faciles a réaliser (voir la section 7.1 à la page 163 pour les perspectives de ce travail).

1.2.1.1 Les composants CwSTS

La figure 1.1 représente la représentation graphique d'un composant CwSTS. Le composant implémente une interface unique. L'interface est composée de l'ensemble des services que le composant fournit et requiert de son environnement. En plus l'interface inclut un protocole d'interaction (donné sous la forme d'un STS) qui spécifie les règles qui régissent l'ordonnancement des messages que le composant échange avec son environnement.

L'interaction entre deux composants est réalisée par l'envoi d'une requête d'exécution de service de la part du composant qui requiert le service envers le composant qui fournit (et implémente) le service correspondant. D'un point de vue opérationnel, le composant qui requiert un service envoie un message qui va être reçu et finalement exécuté par le composant qui fournit le service. En effet, nous ne spécifions pas si l'exécution effective

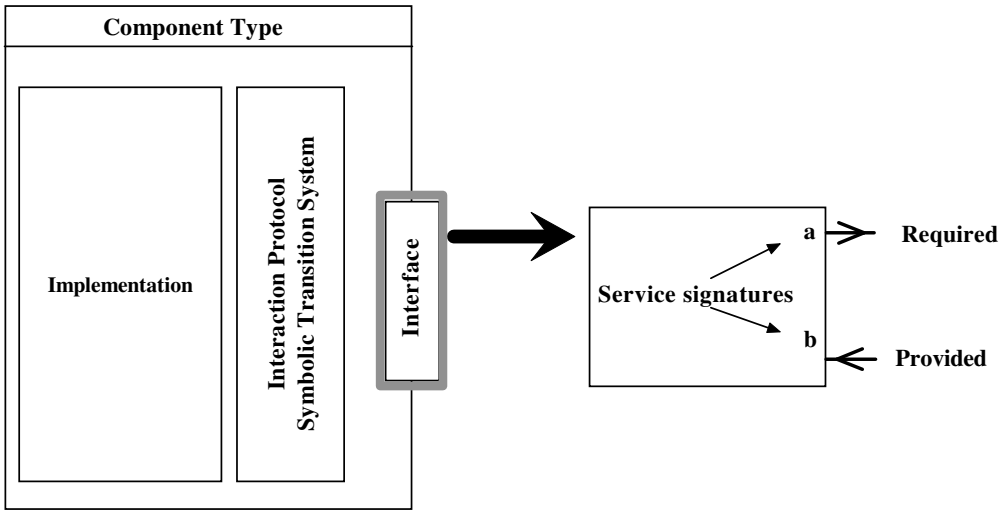


FIGURE 1.1 – Syntaxe graphique d'un composant CwSTS.

du message reçu se fasse en synchrone ou en asynchrone avec l'appel du client. Il peut être exécuté immédiatement après son arrivée (exécution synchrone) et tout en bloquant le client ou ultérieurement à son arrivée (exécution asynchrone) à un moment non spécifié.

Dans le modèle CwSTS, nous parlons des *types de composant* quand nous considérons la définition du composant. L'*implémentation du composant* est représentée par la somme des binaires qui réalisent l'interface spécifiée dans le type du composant. Une *instance du composant* est une instance du type du composant qui est utilisée dans une configuration spécifique à l'exécution.

1.2.1.2 Les interfaces des composants

Comme indiqué dans la figure 1.1, l'interface d'un composant CwSTS décrit les signatures des services qui sont requis ou fournis par le composant (l'interface structurelle). L'interface structurelle correspond au premier niveau des contrats dans la taxonomie proposé par Antoine Beugnard [Beugnard 99]. L'interface comportementale spécifie les instants où un message spécifique peut être reçu par le composant mais aussi les moments où le composant lui-même est susceptible d'envoyer un message. Les protocoles dans le modèle CwSTS correspondent au troisième niveau des contrats de la taxonomie citée ci-dessus.

Les systèmes de transitions symboliques (STS) ont été développés initialement comme une solution aux problèmes d'explosion des états et des transitions dans les algèbres de processus avec passage de valeurs entre les processus. Ce formalisme étend les systèmes de transitions étiquetées avec l'introduction des paramètres et gardes sur les actions des transitions. Nous avons choisi de décrire les protocoles d'interaction au niveau de CwSTS avec une généralisation du formalisme STS. Notre proposition associe un système d'états et transitions symboliques avec une description du type de données qui est représenté par le code d'implémentation du composant. Cette approche n'est pas nouvelle et a prouvé son utilité comme décrit dans [Choppy 00, Royer 03b, Attiogbé 03].

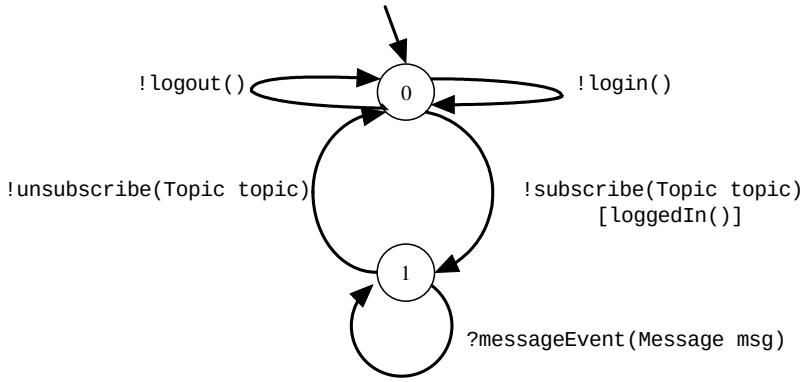


FIGURE 1.2 – Le protocole du composant client du serveur de messagerie.

D'un point de vue opérationnel, l'interface structurelle est une *interface statique*, car les informations qu'elle décrit ne changent pas pendant la phase d'exécution du composant. L'interface comportementale est une *interface dynamique* car elle spécifie les services *disponibles* à des moments précis de l'exécution du composant.

La figure 1.2 présente le protocole d'un composant qui joue le rôle du client d'un serveur de messagerie. Le client va s'identifier auprès du serveur et va s'enregistrer pour recevoir un type spécifique de message. L'interface statique de ce composant est constituée des services requis `login`, `logout`, `subscribe` et `unsubscribe`. Le service fourni `messageEvent` est appelé par le serveur pour signifier l'arrivée d'un message. Le protocole est composé de deux états (les états 0 et 1) qui représentent des états logiques dans l'exécution du composant. Tous les services définis dans l'interface structurelle sont représentés comme des actions forçant la transition du protocole d'un état source à un état destination. La transition qui réalise l'action `subscribe` est gardée par l'évaluation de l'opération booléenne `loggedIn()` qui assure que cette action ne sera pas réalisée tant que la garde n'est pas vraie.

1.2.1.3 La composition

La composition est hiérarchique dans le sens décrit dans le patron de conception *Composite* [Gamma 95]. En composant deux ou plusieurs composants primitifs nous obtenons des composants composites qui à leur tour pourront être utilisés comme des composants dans de futures compositions. Dans notre modèle les composants composites ne contiennent pas de code métier. Ils représentent seulement des agrégats de composants primitifs qui implémentent la logique métier. Ce sont les composants primitifs qui génèrent les messages de sortie ou qui consomment les messages reçus par le composant composite.

La composition est réalisée tant au niveau des interfaces structurelles qu'au niveau des interfaces comportementales. Au niveau structurel, le modèle propose des schémas de composition *binding*, *import* et *export* qui permettent respectivement de connecter deux services correspondants requis/fourni, d'importer un service requis par un sous-composant au niveau de l'interface du composite et d'exporter un service fourni d'un sous-composant au niveau de l'interface du composite. L'interface comportementale du composant compo-

site est représentée par le *produit synchronisé* [Arnold 94, Godefroid 91] des protocoles des sous-composants. Le produit synchronisé est l'automate produit libre qui ne contient que les transitions qui sont authorisées par les règles de synchronisation (et en conformité avec le vecteur de synchronisation). D'une manière générale le produit synchronisé ne contient que les états et les transitions globaux qui sont atteignables dans le cas d'une exécution du protocole du produit. Le vecteur de synchronisation est déterminé par les connexions dans l'architecture (la mise en correspondence des services compatibles conformément à la section 1.2.1.4).

La figure 1.3 présente le composant composite `StatisticsEnabledSever` représentant un serveur de messagerie qui est capable de réaliser des statistiques. Dans cette figure les services fournis et requis du composant composite et les connexions entre sous-composants sont représentées. Le protocole du composant `StatisticsEnabledSever` correspond au produit synchronisé des protocoles de ses sous-composants (leurs protocoles sont représentés dans la figure 1.4 et respectivement dans la figure 1.5 à la page 31).

1.2.1.4 La compatibilité

D'un point de vue informel deux ou plusieurs composants sont compatibles dans une architecture si :

1. Les services qui sont mis en relation sont compatibles (compatibilité structurelle).
2. Les protocoles d'interaction des tous les composants impliqués sont compatibles (compatibilité comportementale).
3. L'architecture est valide.

Deux services sont compatibles s'ils ont le même nom et signature. La compatibilité comportementale fait référence à l'absence de blocage dans le produit synchronisé des protocoles des composants impliqués. Une architecture est considérée comme *valide* si toutes les connexions nécessaires à l'exécution de l'application sont réalisées. En effet, le modèle CwSTS ne requière pas que tous les services des composants soient connectés dans une architecture particulière. Certains services peuvent être connectés dans une architecture alors qu'ils ne le sont pas dans une autre architecture. Le fait qu'il n'y ait pas de dépendance du contexte environnemental est d'une importance majeure au moment de la construction des composants, car ceci assure le fait qu'un composant peut être assemblé et déployé dans plusieurs architectures et environnements différents. Une architecture exécutable est une architecture qui ne requiert pas de services externes, et qui est donc *complète*.

1.2.1.5 Adaptation des composants

Dans le cas où deux ou plusieurs composants ne sont pas compatibles dans une architecture, le modèle CwSTS propose l'utilisation de composants d'adaptation que nous appelons des *adaptateurs*. Au niveau des interfaces structurelles les adaptateurs permettent de changer les noms et signatures des services (ils implémentent le patron de conception *Adapter* [Gamma 95]). Au niveau des comportements, l'adaptateur peut être introduit pour éliminer le blocage qui apparaît au niveau du produit synchronisé. La figure 1.7 présente

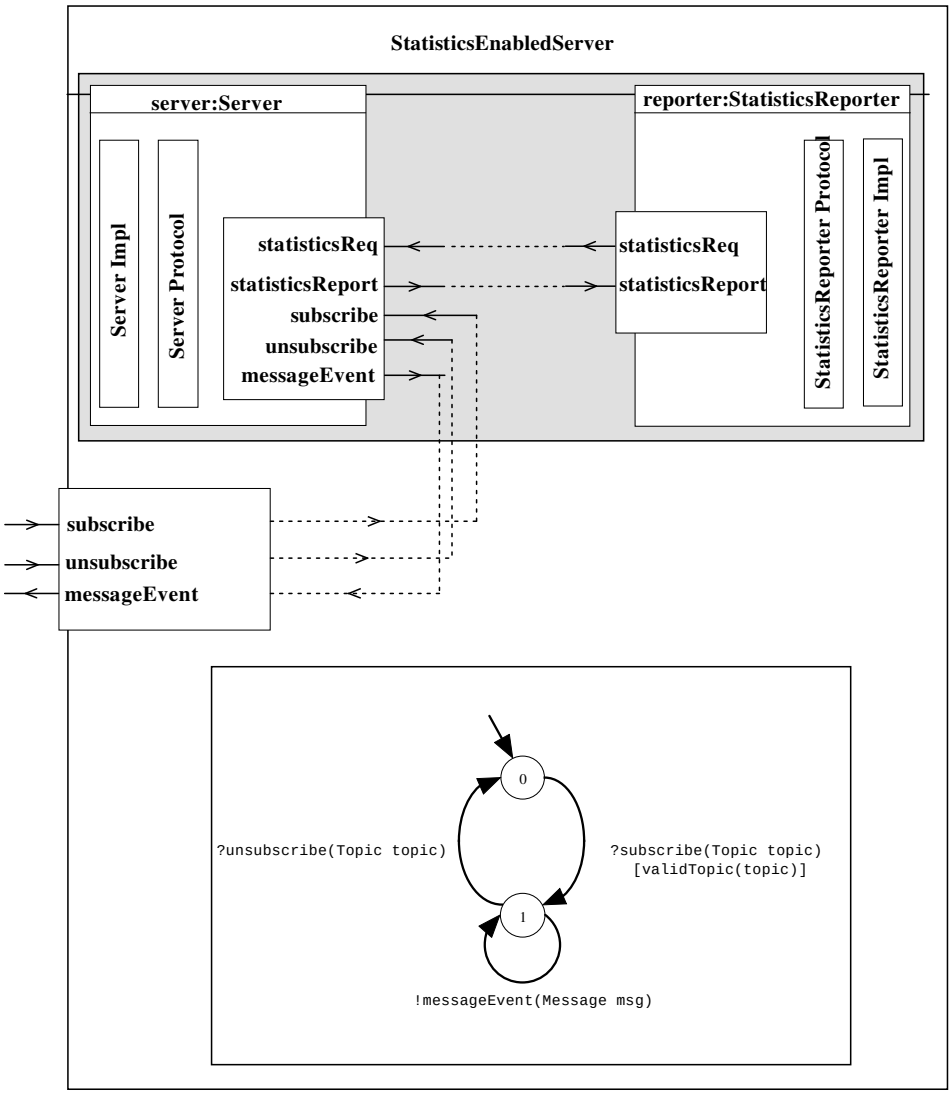


FIGURE 1.3 – Le composant `StatisticsEnabledServer`.

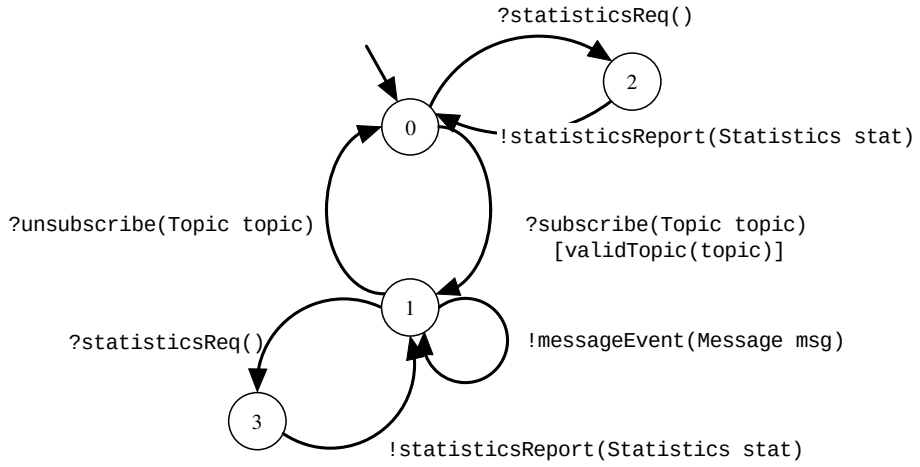


FIGURE 1.4 – Le protocole du composant serveur de messagerie.

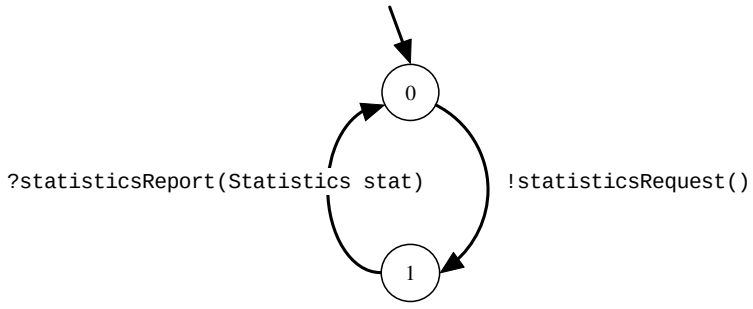


FIGURE 1.5 – Le protocole du composant `StatisticsReporter`.

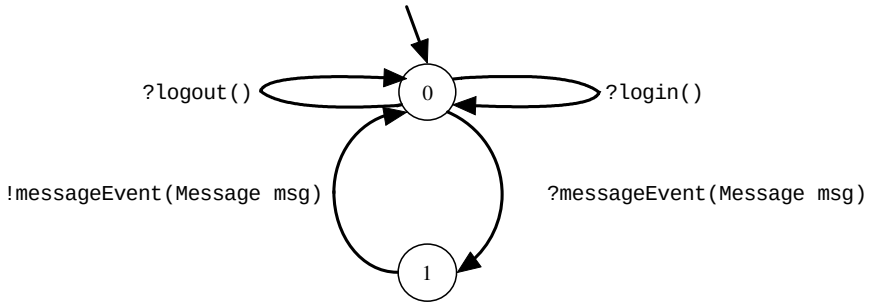


FIGURE 1.6 – Le protocole du composant adaptateur.

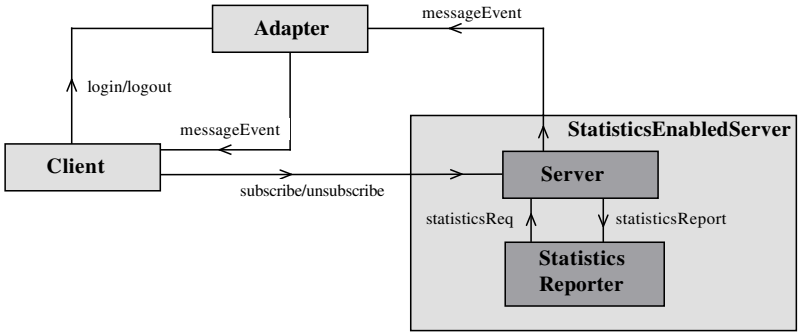


FIGURE 1.7 – Le rôle du composant adaptateur dans l'architecture du système de messagerie.

le cas où un composant adaptateur doit être utilisé pour connecter le client du système de messagerie au serveur de messagerie dû au fait que les interfaces des deux composants ne sont pas compatibles. Le composant client implémente le fait qu'il faut se connecter et se déconnecter avant de souscrire et recevoir des messages de la part du serveur qui lui ne requiert pas ces étapes. Le protocole du composant adaptateur est présenté dans la figure 1.6 et décrit le fait que le client peut faire `login/logout` sans que ces messages arrivent au serveur. Les messages de type `messageEvent` sont transférés du serveur vers le client.

L'adaptateur n'as pas besoin d'intercepter toutes les messages échangés par les composants. Il intercepte que ce que doit être adapté. Dans cet exemple, l'adaptateur doit seulement faire en sorte que les messages `login/logout` soit absorbés sans que le serveur les reçoivent. Évidement, les messages `subscribe/unsubscribe` pourront aussi passer par l'adaptateur mais ceci ne servirait à rien dans ce cas. Avec les adaptateurs nous fournissons un mécanisme d'adaptation très configurable et adaptable aux différentes cas d'utilisation.

1.2.1.6 Substitution des composants

La substitution définit une notion de raffinement sur le type des entités considéré (fonctions, objets ou composants). Les langages de programmation à objets tels que Java

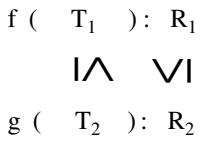


FIGURE 1.8 – Raffinement de la fonction f par la fonction g.

ont déjà abordé ce problème en introduisant une relation de sous-typage (l'héritage). La substitution des composants est plus difficile à réaliser car elle doit considérer les services requis et fournis et aussi les protocoles d'interaction.

Dans le modèle CwSTS, l'intuition que nous avons est que nous pouvons faire une analogie entre la substitution d'un composant et la substitution d'une fonction. En effet, si nous considérons l'ensemble des signatures des services fournis par le composant comme son type d'entrée et l'ensemble des signatures des services requises comme son type de sortie, le parallèle avec le raffinement d'une fonction (comme présenté dans la figure 1.8 où les notions de variance et covariance sont représentées explicitement) est immédiat. Un composant peut remplacer un autre composant au niveau des interfaces structurelles s'il fournit au moins ce que le composant remplacé fournit. De plus, le composant remplaçant requiert au plus ce que le composant remplacé requiert. Nous nous retrouvons dans le cas classique des relations de covariance/contravariance entre les types des deux composants.

Le raffinement comportemental implique le fait que si nous considérons une sémantique des traces associée aux protocoles STS, le protocole du composant remplaçant doit accepter au moins les même traces (envois et réceptions des messages) que le composant remplacé. De plus, le composant remplaçant doit refuser les mêmes requêtes que le composant remplacé. Nierstratz [Nierstrasz 93] définit ce type de substitution en fonction des traces et défaillances. Alors que les traces définissent l'ensemble des actions à exécuter, les défaillances définissent les requêtes qu'un composant ne peut pas accepter après qu'une trace spécifique ait été exécutée.

1.2.1.7 Le langage de description de composants CwSTS-IDL

Nous avons développé un langage de description des interfaces des composants CwSTS. Ce langage permet la description des interfaces structurelles et comportementales. La syntaxe abstraite de ce langage est présentée dans la figure 1.9. Le protocole est défini en utilisant un langage de processus appelé SFSP (*Symbolic Finite State Processes*). SFSP est inspiré de FSP (*Finite State Processes*) [Magee 99] mais retient seulement les transitions, le choix et la récursivité. De plus les actions sont paramétrables avec des types de données quelconques. La syntaxe abstraite du langage SFSP est présentée dans la figure 1.10 à la page 34. Á titre d'exemple, la figure 1.11 à la page 35 présente la description structurelle et celle comportementale, du composant client du système de messagerie.

1.2.2 L'implémentation du modèle

CwSTS-P est l'implémentation préliminaire de notre modèle dans le langage de programmation à objets Java. Le rôle de cette implémentation est de permettre une validation

```
component_def        ::= primitive_def | composite_def
primitive_def        ::= component_type
                         primitive_strct_def
                         guards_def
                         primitive_protocol
primitive_struct_def ::= service_def+
service_def          ::= service_type op_id formal_parameter_list? | op_id formal_parameter_list?
guards_def           ::= op_id formal_parameter_list?
service_type         ::= 'required' | 'provided'
composite_def        ::= component_type
                          subcomponent_decl+
                         composite_structural_def?
                         connection_def+
subcomponent_decl    ::= component_type subcomponent_id
composite_struct_def ::= service_def+
connection_def       ::= bind_exp | export_exp | import_exp
bind_exp             ::= subcomponent_op_id 'to' subcomponent_op_id
export_exp           ::= subcomponent_op_id 'as' op_id
import_exp           ::= op_id 'to' subcomponent_op_id

subcomponent_op_id   ::= subcomponent_id '.' op_id
```

FIGURE 1.9 – Grammaire du langage de description des interfaces CwSTS-IDL.

```
specification      ::= process_def+
process_def        ::= process_id formal_param_list process_body

process_body       ::= choice+
choice             ::= action+ process_inst
process_inst       ::= process_id actual_param_list | special_process

special_process    ::= STOP

action             ::= receive_act | send_act | internal_act
receive_act        ::= action_id formal_param_list
send_act           ::= action_id actual_param_list
internal_act       ::= action_id actual_param_list

formal_param_list  ::= param_type param_id | formal_param_list | ε
actual_param_list  ::= param_id | actual_param_list | ε
```

FIGURE 1.10 – La grammaire abstraite du langage SFSP.

```
primitive component Client {
    interface
        provided messageEvent(Message msg);
        required login();
        required logout();
        required subscribe(Topic topic);
        required unsubscribe();

    guards
        loggedIn();

    protocol
        P=!login -> P | !logout[!loggedIn()] -> P | !subscribe[!loggedIn()] -> Q,
        Q= ?messageEvent -> Q | !unsubscribe() -> P.
}
```

FIGURE 1.11 – La définition du composant client du serveur de messagerie.

des notions conceptuelles décrites dans le modèle de composants CwSTS et de fournir des outils de génération de code, d'analyse des propriétés des architectures (validité, compatibilité) et d'adaptation des composants à l'origine non compatibles tel que décrit dans le modèle CwSTS.
Nous nous basons sur une approche générative dans la construction des composants CwSTS. La figure 1.12 présente les deux scénarios que nous considérons :

- **Scénario A.** Nous partons d'une ou plusieurs descriptions de composants (descriptions des interfaces structurelles et comportementales) qui, avec des instructions d'instrumentation, sont passées à un moteur de génération de code. Le résultat est un composant CwSTS valide pour chaque description de composant fournie en entrée. Cette implémentation représente soit une implémentation par défaut du code métier du composant soit une implémentation orientée code d'adaptation (code d'implémentation du composant adaptateur). Le code métier par défaut ne contient pas de comportement particulier et est préparé pour être personnalisé afin de répondre à des besoins spécifiques. La construction du code personnalisé doit ensuite utiliser un certain nombre de règles simples afin de garantir la conformité de l'implémentation vis-à-vis de la spécification.
  Les *composants adaptateurs* générés sont utilisés dans une architecture qui requiert l'adaptation structurelle et/ou comportementale des composants la constituant. Le code des adaptateurs n'a pas besoin d'être personnalisé pour répondre aux besoins pour lesquels il a été généré.

- **Scénario B.**
  Ce scénario est différent du premier scénario par le fait qu'au moment de la génération, nous fournissons aussi le code représentant le code métier du composant généré. Nous partons donc du code qui a besoin d'être instrumenté pour obtenir

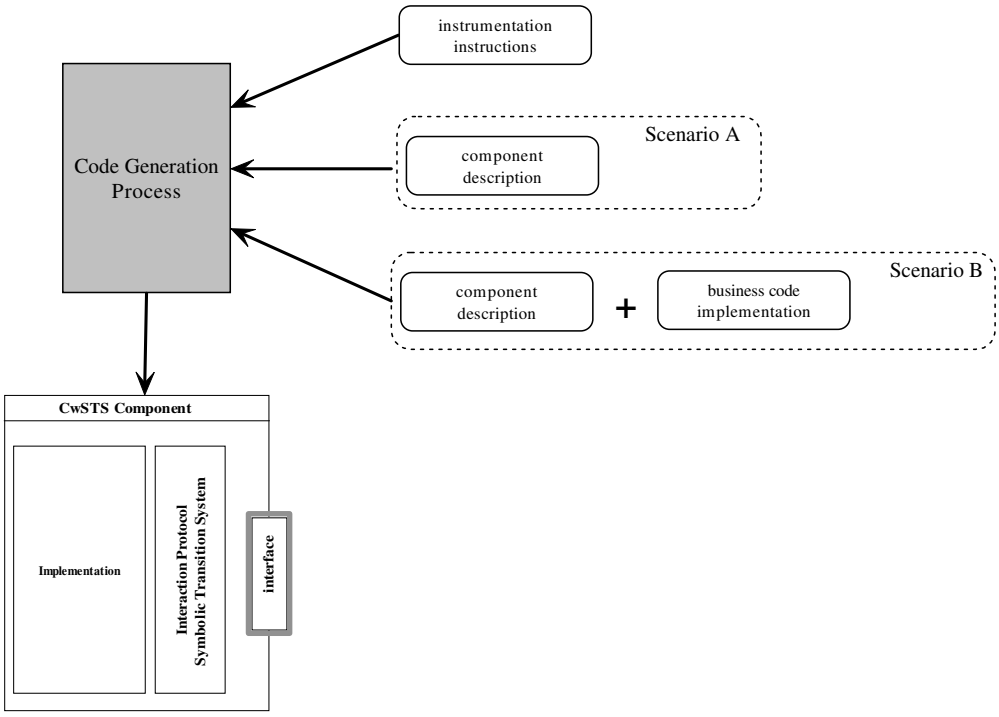


FIGURE 1.12 – Scénarios de génération de composants.

un composant conforme au modèle CwSTS. Dans ce scénario, la description du composant doit être *conforme* au code fourni, et ceci aux niveaux structurel et comportemental.

Notre approche générative répond au besoin d'assurer que l'implémentation est *conforme* aux spécifications. En suivant n'importe lequel des deux scénarios présentés ci-dessus, nous assurons *par construction* que le code d'implémentation du composant CwSTS est conforme tant en terme de structure que de comportement aux spécifications.

Dans ce qui suit nous allons présenter des points significatifs de l'implémentation des composants : l'encapsulation du code Java pour répondre aux exigences du modèle CwSTS, la structure d'un composant (primitif ou composite) et l'implémentation du contrôle du comportement d'un composant.

1.2.2.1 Structure des composants

Dans le prototype CwSTS-P, chaque composant est structuré sous la forme d'un *package* de classes Java classique. Ce *package* contient les classes d'implémentation du composant. La figure 1.13 présente la structure du *package* où les classes et les interfaces constituant le code du composant sont structurées dans des sous-*packages*.

Le nom du *package* est le nom du composant pendant la phase de développement. Le nom est conforme au standard proposé qui est : `fr.emn.CwSTS-P.`*NomDuComposant*. La

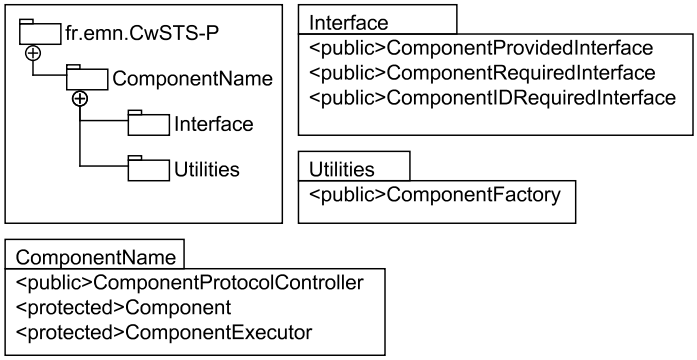


FIGURE 1.13 – Structuration d'un composant sous la forme d'un *package* Java.

racine de ce package contient les classes qui seront instanciées pour obtenir une instance de composant exécutable. Les deux sous-packages *Interface* et *Utilities* contiennent les interfaces déclarées du composant et une classe *factory*. Les interfaces définissent les services fournis et requis par le composant et la *factory* est utilisée pour obtenir de nouvelles instances du composant de ce type.

Notre modèle de composants définit des composants de type boîte noire. Ceci signifie que la structure interne du composant n'est pas visible de l'extérieur du composant à l'exécution. Le seul moyen d'interagir avec le composant est d'utiliser ses interfaces. L'encapsulation du code d'exécution du composant sous la forme d'un package Java va dans la direction de répondre à cette contrainte. Mais le package Java tout seul ne suffit pas. Afin de parfaitement isoler les classes du package de l'environnement d'exécution du composant, les seules entités publiques du package sont les interfaces, le contrôleur de protocole (qui représente le *front-end* du composant) et le *factory* correspondant au composant.

1.2.2.2 Implémentation d'un composant primitif

Un composant primitif CwSTS-P est constitué de trois types d'entités : une ou plusieurs classes contenant le *code métier* du composant, un *exécuteur* du composant et une classe qui joue le rôle de *contrôleur* du protocole implémenté par le composant. La figure 1.14 présente une vue fonctionnelle de la structure du composant.

Le code métier

Le code métier (*business logic* en anglais), est le code qui réalise la fonctionnalité du composant. Il implémente les interfaces fournies par le composant et est donc capable d'exécuter les messages reçus par le composant. C'est aussi cette entité qui génère des messages à destination d'autres composants.

L'exécuteur

L'exécuteur du composant fonctionne comme un mandataire (*proxy* en anglais) au nom du code métier. Son rôle principal est d'intercepter les fils d'exécution (*threads* en anglais) qui appellent les méthodes du code métier et d'exécuter ces appels mais dans des fils

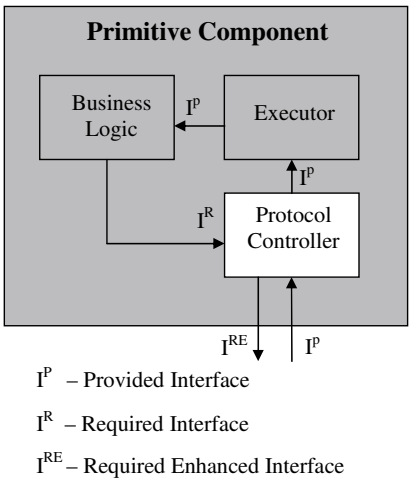


FIGURE 1.14 – Vue fonctionnelle sur la structure d'un composants primitif.

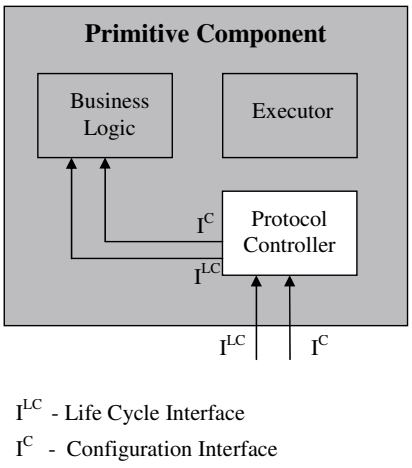


FIGURE 1.15 – Vue de configuration sur la structure d'un composant primitif.

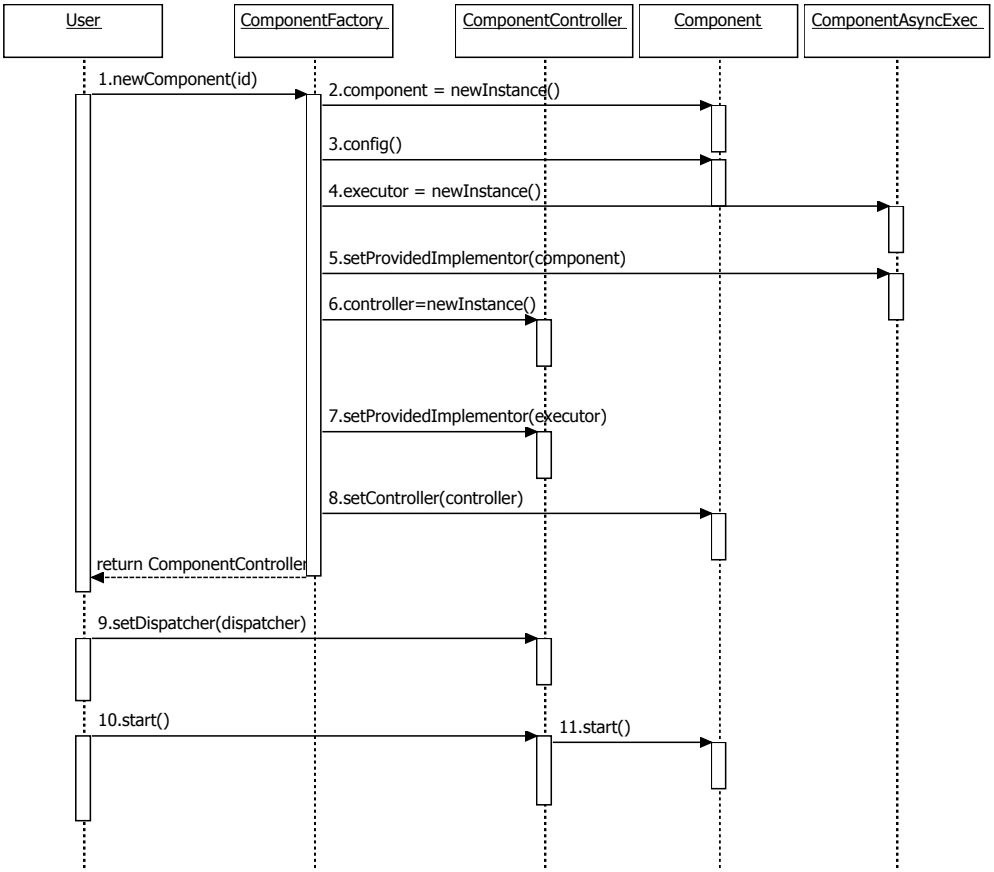


FIGURE 1.16 – Création d'une instance d'un composant primitif.

différents de ceux qui contiennent l'appel d'origine. Ceci est utile pour implémenter les spécifications du modèle CwSTS qui stipule que la réception d'un message est toujours synchrone avec son envoi par un autre composant mais que l'exécution proprement dite du message (de l'appel encapsulant le message) est soit synchrone soit asynchrone avec l'envoi. En effet, le fil d'exécution démarré par l'exécuteur découple l'appel de l'exécution du message.

Le contrôleur

Le contrôleur du protocole du composant contrôle les appels qui entrent et qui sortent du composant. Le protocole implémenté par le composant spécifie la séquence des messages que le composant doit recevoir ou émettre. Le contrôleur assure que l'ordre des appels encapsulant les messages va être conforme au protocole. Le contrôleur implémente l'interface fourni par le composant et l'interface requise augmentée d'une information qui est l'identifiant du composant lui même. Ce mécanisme est nécessaire pour permettre la composition du composant dans une architecture (voir les explications sur le répartiteur à la page 41).

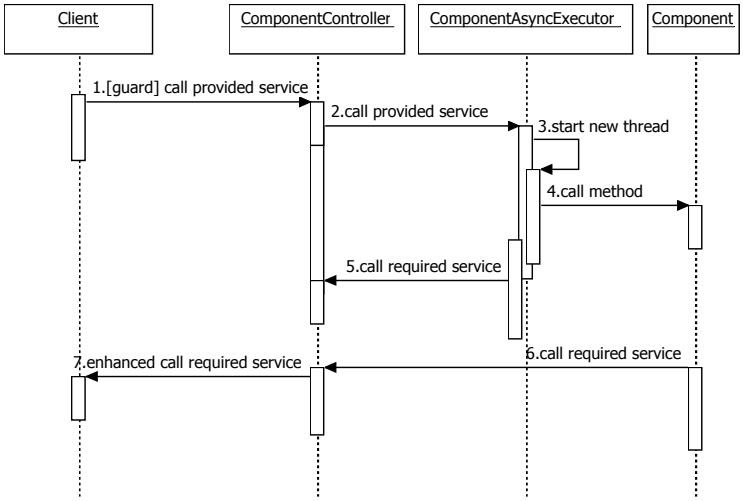


FIGURE 1.17 – Interaction des entités du composant primitif.

La configuration, l'instanciation et l'exécution des composants

La figure 1.15 présente la vue de configuration sur les relations entre les différentes entités d'un composant primitif. Plusieurs phases ont lieu avant l'exécution proprement dite du composant (voir la figure 1.16). Les instances Java du code métier (*Component* dans la figure), du code exécuteur (*ComponentAsyncExec* dans la figure) et du contrôleur (*ComponentController* dans la figure) sont créées. Après une phase de paramétrage des trois entités, la référence au contrôleur du protocole est retournée au client ayant demandé l'instanciation du composant. Le client peut aussi configurer des informations environnementales avant de démarrer l'exécution proprement dite du composant en appelant la méthode `start()`, appel qui sera propagée au code métier du composant.
Une fois démarré, le composant se comporte comme indiqué dans la figure 1.17. La garde sur la ligne 1 du diagramme de séquence indique la présence d'un mécanisme qui en utilisant la notion de monitor Java, permets la synchronisation de l'appel du client avec la réception au niveau du `ComponentController`. Dans le cas d'un appel émis par le composant lui-même (appel provenant du code métier), le contrôleur du composant transfère l'appel après une phase de *décoration* de l'appel avec des informations d'identification du composant. Ces informations sont utilisées par le composant composite (comme présenté ci-dessus) pour diriger l'appel vers son composant destinataire (et ceci conformément aux liens structurels définis dans l'architecture).

1.2.2.3 Implémentation d'un composant composite

Conformément aux spécifications du modèle CwSTS, le composant composite ne contient pas de code métier : la fonctionnalité du composite est réalisée à l'exécution par l'interaction entre les sous-composants qui le composent. De plus, le composant composite comporte un contrôleur (similaire au contrôleur des composants primitifs) et une entité répartiteur (*dispatcher* en anglais). Le répartiteur implémente une stratégie de transfert

d'appels qui réalise à l'exécution les relations structurelles entre les sous-composants du composant composite. La figure 1.18 présente la structure d'un composant composite.

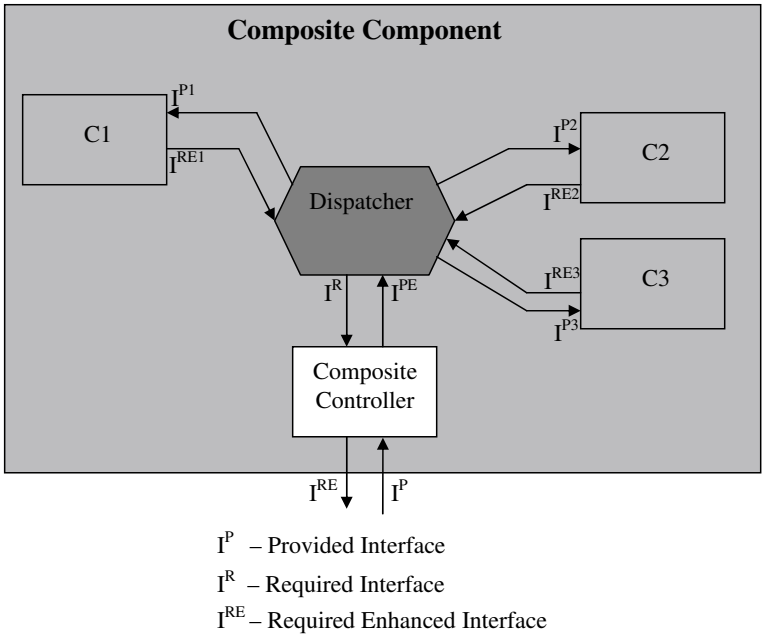


FIGURE 1.18 – Vue fonctionelle sur la structure d'un composant composite.

Sous-composant

Un composant composite contient plusieurs sous-composants. Les sous-composants sont soit des composants primitifs soit des composants composites (car la composition est hiérarchique dans CwSTS). Les sous-composants ne sont pas directement interconnectés dans un composite. Ils sont tous connectés à l'entité répartiteur.

Le répartiteur

Le répartiteur est responsable du transfert des appels qui proviennent des sous-composants vers les composants spécifiés dans l'architecture du composite. Le répartiteur est basé sur une table de routage qui représente à l'exécution les relations structurelles entre les sous-composants du composant composite. Le répartiteur ne contient pas du code de synchronisation Java et donc aucun blocage n'est possible à son niveau. La synchronisation effective entre les composants se réalise au niveau des sous-composants qui contrôlent localement les réceptions et les émissions des appels. Pour réaliser le routage des appels, le répartiteur doit connaître l'identité des composants qui sont la source des appels. Comme spécifié dans la figure 1.18, le répartiteur implémente toutes les interfaces requises des sous-composants du composant composite. De plus, ces interfaces sont personnalisées avec un paramètre qui est l'identificateur du sous-composant à l'origine de l'appel.

Le contrôleur

Le contrôleur est le *front-end* du composant composite. Tous les appels qui sortent ou qui entrent dans le composant composite sont interceptés par cette entité. Dans ce but, le contrôleur implémente les interfaces fournies et requises par le composant composite. Á l'opposé du contrôleur d'un composant primitif, le contrôleur du composite ne réalise pas de vérification du protocole. En effet, le contrôleur ne connaît même pas le protocole du composite, car celui-ci est réalisé à l'exécution par les interactions entre les sous-composants qui réalisent le produit synchronisé de leurs protocoles. C'est le contrôle réalisé au niveau de chaque composant primitif qui permet la réalisation des synchronisations nécessaires au bon comportement du composant composite.

La configuration, l'instanciation et l'exécution des composants

La configuration, l'instanciation et l'exécution des composants composites sont réalisées en utilisant les relations statiques présentées dans la figure 1.19.

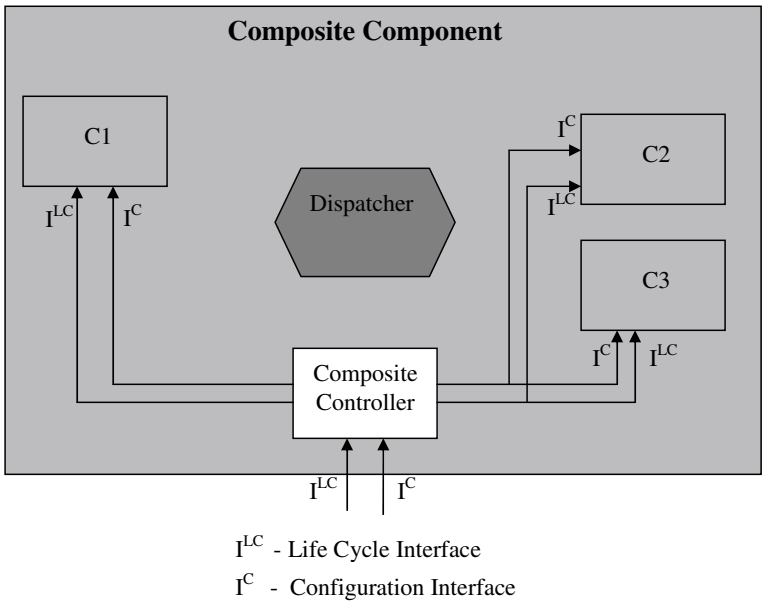


FIGURE 1.19 – Vue de configuration sur la structure d'un composant composite.

La phase de configuration a lieu entre les actions de création d'instances de sous-composants et de démarrage d'exécution (voir figure 1.20). Le client d'un composant composite configure les données nécessaires au composite pour s'exécuter. Le contrôleur du composite se charge de transmettre ces informations au niveau des sous-composants. Habituellement, ces informations contiennent les valeurs de paramètres d'initialisation nécessaires aux composants. D'autres informations qui sont complètement transparentes pour les clients du composite incluent la référence du répartiteur qui est transmise aux sous-composants et les relations structurelles entre les sous-composants. Ces relations structurelles sont codées dans la table de routage au niveau du répartiteur.

Une fois la phase de configuration terminée et les composants démarrés les appels qui arrivent au niveau du composite sont interceptés par le contrôleur qui les transfère au niveau du répartiteur. En fonction de l'identité de l'appelant (les sous-composants ou le contrôleur), le répartiteur route les appels vers leur destinataire. Les appels émis par le composant sont traités de la même manière.

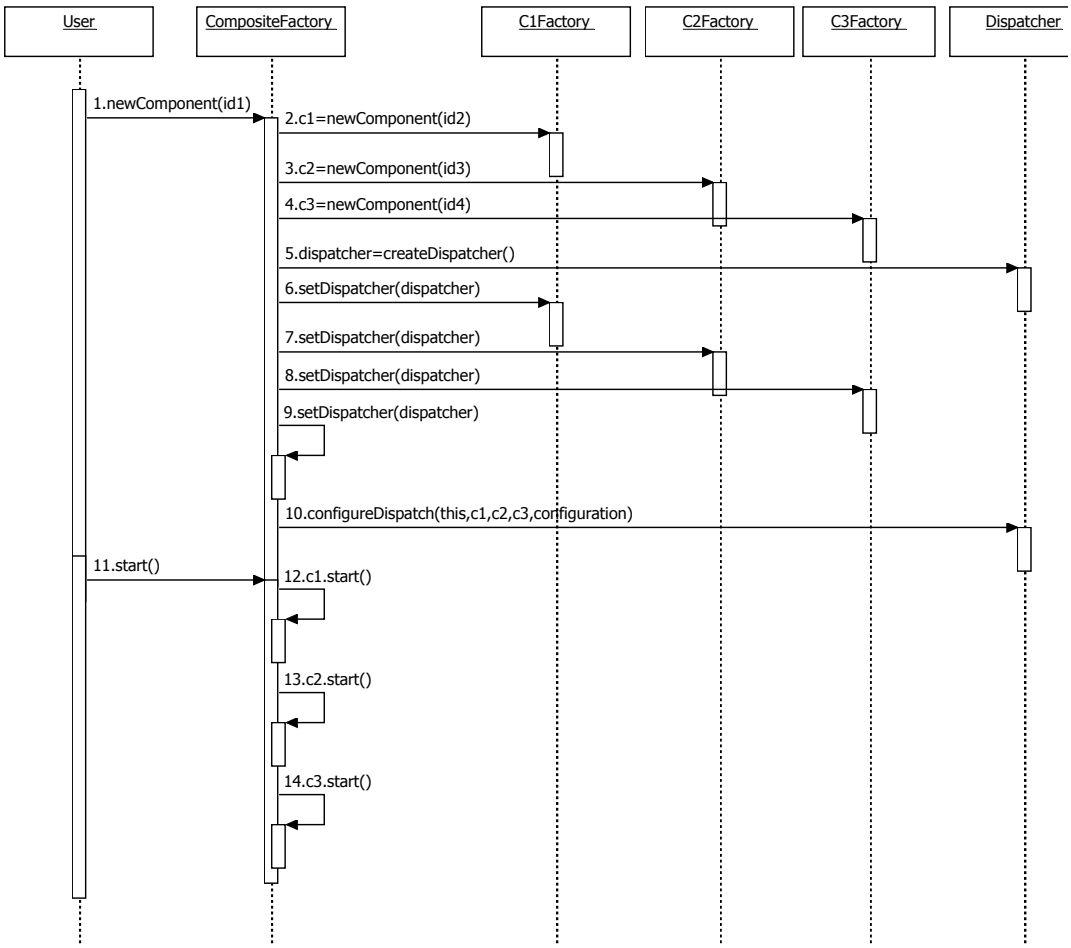


FIGURE 1.20 – Création d'une instance d'un composant composite.

1.2.2.4 La composition comportementale

Comme indiqué auparavant, un composant composite n'implémente pas le protocole représentant le produit synchronisé de ses sous-protocoles. Le produit synchronisé n'est donc pas contrôlé par le contrôleur du composite. C'est l'exécution des sous-composants qui réalise le produit synchronisé global de leurs protocoles respectifs.
Ceci représente une solution dite *distribuée* de contrôle du protocole. Les synchronisations entre l'envoi et la réception des messages encapsulés sous la forme des appels d'un composant envers un autre composant se réalisent au niveau de chaque composant (au

niveau du contrôleur des composants primitifs plus exactement). L'envoi et la réception du message sont exécutés d'une manière *synchrone*, c.à.d. que le composant émetteur est bloqué une fois l'émission démarrée et tant que le composant receveur n'as pas reçu le message. Une fois la réception acquittée, le composant receveur débloque l'émetteur et procède à l'exécution immédiate ou différée du message reçu. C'est dans ce mécanisme que réside l'implémentation de la synchronisation nécessaire à la réalisation du produit synchronisé des composants qui interagissent. Un composant émetteur va émettre un message uniquement si les conditions spécifiées par la garde indiquée dans son protocole d'interaction sont vérifiées. Sinon, aucun message n'est émis. Un composant receveur peut recevoir le message uniquement si, par rapport à son protocole d'interaction, il se trouve dans un état où il peut recevoir ce message. Sinon, le message est mis en attente de réception, ce qui bloque son émetteur, qui n'est pas autorisé à exécuter d'autres actions pendant toute la période de blocage. Au moment où le composant receveur se trouve dans un état permettant la réception du message bloqué, la réception a lieu et l'émetteur est débloqué. Même si le protocole du receveur est dans un état où il peut recevoir un message particulier, la garde spécifique à la transition doit être vraie. Si ce n'est pas le cas, le message n'est pas reçu et mis en attente. Le receveur n'est jamais bloqué par un message en attente d'être reçu ce qui fait que d'autres messages reçus peuvent changer l'état interne du composant en permettant à une nouvelle évaluation des gardes de débloquer un message en attente de réception dûe aux évaluations des gardes. L'implémentation est non déterministe dans le sens où si deux ou plusieurs gardes deviennent franchissables le message reçu n'est pas déterminé d'une manière statique. Ceci dépends des mécanismes de contrôle des fils d'exécution au niveau de la machine virtuelle Java.
Ce mécanisme a prouvé son efficacité dans l'implémentation du produit synchronisé, sans avoir à recourir à un mécanisme dit *centralisé* ou à une entité spécifique (comme par exemple un contrôleur du protocole global d'une application) qui fera le contrôle de chaque paire d'actions émission/réception. Une approche centralisée pose des problèmes importants tant en terme d'efficacité que de passage à l'échelle car complètement contre-indiquée dans un environnement réparti.

Pour l'implémentation effective de notre mécanisme *distribué* de synchronisation nous avons recours au mécanisme de *synchronisation* du langage Java ou les fils d'exécution des applications Java sont instrumentés pour arriver à une exécution chorégraphié. Le prototype CwSTS-P étant implémenté en utilisant la version Java 5, nous avons utilisé des structures de synchronisation très optimisées comme par exemple les files d'attentes, les verrous explicites (les *locks* en anglais), etc pour réaliser la synchronisation des fils d'exécution qui encapsulent les appels d'un composant à un autre.

1.2.3 Perspectives

Nous avons opté pour une approche générative en ce qui concerne la construction des composants. Ceci est en phase avec l'approche MDA actuellement en pleine reconnaissance dans le monde du développement des applications logicielles. La génération permet de construire plus rapidement des applications plus sûres et peut être intégrée avec des

techniques de verification et d'analyse sophistiquées. Les outils proposés avec l'implémentation du modèle CwSTS représentent les briques de base pour pouvoir réaliser des logiciels de génération de code qui intègrent des protocoles d'interaction sous la forme de systèmes de transitions symboliques.

Le modèle actuel prend en charge uniquement des configurations statiques. Une fois les composants de l'architecture configurés et démarrés, aucune modification structurelle ou comportementale des composants n'est plus possible. La tendance actuelle dans le développement des logiciels est de fournir aux applications la possibilité de se reconfigurer rapidement *à chaud*. Ceci signifie qu'une application est capable de se reconfigurer tout en continuant son exécution, ou du moins sans avoir à tout recompiler et redéployer. Une des possibles évolutions du modèle CwSTS est donc de permettre aux composants de se reconfigurer (surtout en termes de connexions à l'intérieur d'un composite) sans avoir à arrêter complètement l'exécution de l'application. Ceci sera possible en concevant des composants qui fonctionnent suivant plusieurs phases (*démarré*, *en configuration*, *en pause*, *en arrêt*, par exemple) après le démarrage de l'application.

Pour des raisons de simplicité, nous n'avons considéré que le cas où les communications entre les composants sont point à point. Ceci signifie que le service fourni par un composant ne peut être appelé qu'à partir d'un seul et unique autre composant pendant toute la vie de l'application. Dans la réalité des applications logicielles d'aujourd'hui, la communication point à point est très restrictive surtout si nous considérons des applications client-serveur où un seul composant serveur peut recevoir des appels d'exécution de services en provenance de plusieurs clients. D'autres schémas d'interaction peuvent être considérés par une extension du modèle CwSTS. Ces schémas peuvent inclure la communication n à 1 ou 1 à n permettant de concevoir des architectures complexes en termes d'interaction entre les composants. Le mécanisme de synchronisation permettant la réalisation du produit synchronisé est, en partie, équipé pour faire face à de tels schémas de communication mais des études complémentaires et leurs implémentations sont à prévoir.

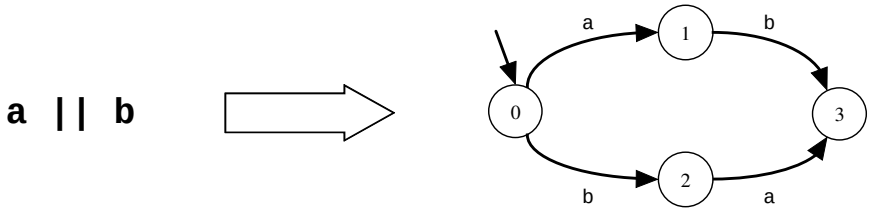


FIGURE 1.21 – Exécution parallèle de deux actions.

Le langage SFSP (basée sur le langage FSP) est un langage de processus qui décrit les protocoles d'interactions des composants. Le langage est très précis dans la description des transitions des protocoles mais pour le cas où nous voulons décrire deux transitions qui peuvent s'exécuter en parallèle, nous sommes obligés de transcrire cette situation comme présenté dans la figure 1.21. Les langages FSP et SFSP décrivent les actions parallèles en les représentant d'une manière séquentielle plutôt qu'en restant *indépendantes* et en faisant abstraction de la façon dont l'exécution des actions aura lieu. Une possible évolution du langage SFSP sera de considérer ce cas précis de description des transitions en parallèle

qui fera abstraction de la manière exacte de l'exécution pourvu que le comportement décrit soit réalisé. Dans un environnement distribué, l'exécution des actions se réalise rarement d'une manière séquentielle ce qui indique une fois de plus l'utilité d'une telle extension.

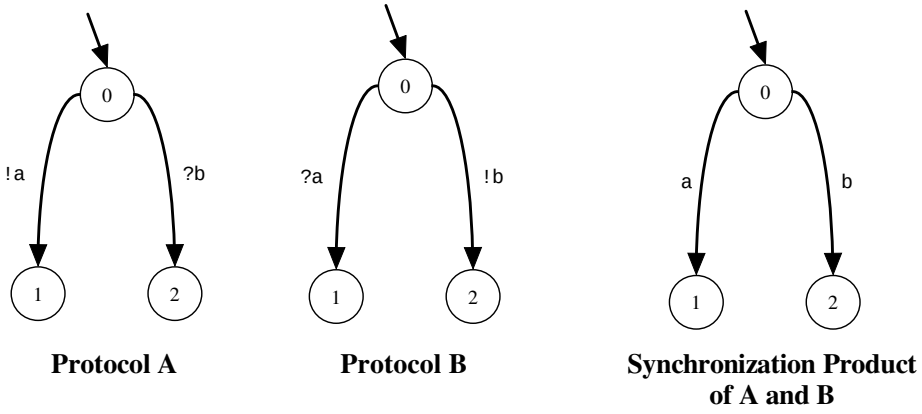


FIGURE 1.22 – États mixtes dans l'interaction entre deux composants.

En ce qui concerne l'implémentation du modèle et plus particulièrement l'implémentation de la réalisation du produit synchronisé à l'exécution, le cas des *états mixtes* [Yellin 97] n'est pas complètement pris en compte. La figure 1.22 présente un exemple de ce cas. Si deux composants qui communiquent entre eux se trouvent chacun dans un état où ils peuvent tous les deux émettre ou recevoir un message, notre implémentation ne garantit pas qu'il n'y aura pas d'interblocage applicatif (dû au fait que les deux composants essaient d'envoyer un message « au même moment »). Comme l'implémentation actuelle bloque l'émetteur tant que le receveur n'a pas reçu le message, chacun des deux composants peut être bloqué en attendant que son partenaire de communication reçoive son message. Une evolution de l'implémentation devrait donc éliminer cette limitation qui, en pratique, est généralement imprévisible et dépend exclusivement des entrelacements des actions des composants. Ceci pourra être basé sur l'intégration de la synchronisation distribué avec une forme de contrôle centralisé (basé sur l'idée de moniteur centralisé présenté dans la section 6.2).

Toujours en rapport avec SFSP, le langage FSP propose aujourd'hui des outils de visualisation et de vérification des propriétés comme par exemple l'interblocage. Une évolution future de l'implémentation de notre modèle pourrait proposer des outils similaires qui permettront une meilleure description et vérification des protocoles d'interaction.

1.3 Structure du mémoire de thèse

Le corps de ce mémoire en anglais est structuré en deux parties. La première partie présente le contexte de notre travail. La deuxième partie de ce document présente la contribution de notre travail : un modèle de composants intégrant un formalisme lisible, expressif et compact basé sur des systèmes de transitions symboliques.

La première partie présente le contexte de notre travail. Dans la section 3.1 nous présentons l'état de l'art des caractéristiques des modèles de composants actuels et nous l'illustrons avec des propositions du domaine de la recherche et de l'industrie. Les langages de description d'architectures [Medvidovic 00b] sont présentés dans la section 3.3.4. Un ADL est défini comme une notation formelle ou informelle, textuelle ou graphique qui permet la spécification des architectures logicielles et qui est accompagnée d'outils spécifiques comme présenté dans [Medvidovic 00a]. Le résultat de l'utilisation d'un langage de description d'architecture est une vision abstraite en termes de composants, connecteurs et configurations sur l'application.

La plupart du temps, la vision concrète de l'implémentation est ignorée dans une approche de type langage de description d'architecture. L'analyse architecturale réalisée avec un ADL peut révéler des propriétés importantes, mais celles-ci ne sont pas en général garanties par l'implémentation. Pour permettre une analyse architecturale au niveau du code, le code doit être conforme à l'implémentation. L'approche de la programmation à base de composants (COP pour *Component-Oriented Programming*) dans la section 3.3.6) propose de développer des logiciels en les programmant directement avec des composants plutôt que des objets.

Le paradigme de coordination est présenté dans la section 3.3.7. La coordination est une approche académique centrée sur la collaboration entre processus. Cette approche considère la programmation des systèmes distribuées ou parallèles comme la combinaison de deux activités distinctes : l'activité de calcul proprement dite, qui contient un nombre des processus impliqués dans la manipulation des données et l'activité de coordination qui est responsable de la communication et de la coopération des processus.

Après cela, nous considérons l'état de l'art dans le domaine de la spécification des protocoles d'interaction. Différents formalismes et modèles de composants intégrant ces formalismes sont présentés. Les algèbres de processus sont présentées dans la section 4.2.1. Le terme algèbre de processus fait référence à une famille de techniques de spécification particulièrement adaptées à la description des systèmes concurrents. Le processus d'interaction est décrit en termes de règles de calcul sur des expressions représentant des processus.

La section 4.2.2 et la section 4.2.3 présentent les types comportementaux et les formalismes basées sur les machines à états finis. Les logiques temporelles sont présentées en dernier dans la section 4.2.4, pour clore cette partie sur les formalismes de description des protocoles d'interaction.

La deuxième partie de ce document présente la contribution de notre travail : un modèle de composants intégrant un formalisme lisible, expressif et compact basé sur des systèmes de transitions symboliques.

Nous commençons par décrire notre modèle de composants d'un point de vue informel dans la section 5.2. Les composants, leurs interfaces, les règles de composition et les détails sur les protocoles STS sont illustrés avec un exemple qui nous sert de fil conducteur.

Les notions de composant, d'interface structurelle et comportementale, ainsi que la compatibilité et la substitution des composants sont définies formellement dans la section 5.3. Le langage de description d'interfaces de notre modèle, CwSTS-IDL (*Components with STS - Interface Description Language*), et est présenté dans la section 5.4. SFSP le

langage de description des protocoles d'interaction de composants CwSTS est détaillé dans la section 5.4.3.

Le prototype (CwSTS-P) de notre modèle et son implémentation sont présentés dans la section 6.1. L'encapsulation de l'implémentation d'un composant sous la forme d'un paquetage Java sur lequel nous imposons des restrictions est présentée dans la section 6.1.2. Dans la section 6.1.3 nous présentons la structure standard d'un composant primitif avec ses entités constituantes et e mécanisme de configuration, d'instanciation et d'exécution. La structure d'un composant composite est indiquée dans la section 6.1.4 et le mécanisme de composition des protocoles d'interaction est présenté dans la section 6.2.

Nous formulons les conclusions et les perspectives de notre travail dans le chapitre 7.

Part II

Work Context in English

Chapter 2

Introduction

By the end of the sixties, it became obvious that the way of developing software needed to be reconsidered. At that time, applications were made on demand and developed from scratch. As the demand was constantly augmenting, the first drawbacks of this approach to software development made their appearance: the software quality was poor and the exploitation and maintenance costs were high. Imposed by the market dynamics, the rapid changes on application development could not be performed as rapidly as required. Time to market and the clients pressure resulted in the fact that the documentation provided with applications was usually incomplete or even incorrect. By this fact, it was very difficult, if not impossible, to reuse parts of an application in another project.

The Component-Based Software Engineering (CBSE) paradigm is trying to overcome these issues by using software component entities at the heart of any software application. The notion of software component was firstly mentioned by McIlroy [McIlroy 68] in his speech to the OTAN conference in 1968 and the idea of software reutilization emerged as a solution to the above-mentioned problems. This would had been realized by the creation of a software industry where applications were created by reusing software components rather than creating them from scratch.

Components relate to modules in that component technology unavoidably leads to modular solutions. While modularity is a prerequisite, rules beyond the traditional modularity criteria are needed to form components rather than just modules. Adopting component technology requires adoption of principles of independence and controlled explicit dependencies.

The Object-Oriented Programming (OOP) credo in reutilization fully revealed its weakness by the end of the nineties. In contrast with the OOP mechanisms, which introduces a strong coupling relation between base classes and derived ones (problem known under the name of Fragile Base Class), the CBSE approach provides a better separation between the implementation of a software component and its interface. Each component is an indivisible black-box entity that can be composed and independently deployed. The construction of applications is realized by assembling components. In this approach, the component interface is of major importance as it defines a contract between the component and its environment. It clearly describes what the component provides to and what it requires

from its environment in order to fulfill its intended purpose in an architecture.

Particular component characteristics are described by models (component models). A component model describes what a component is (by defining its constituent parts) and says how components can be eventually composed (by following some composition rules). It also describes the component life cycle and the roles associated with different actors in the development and exploitation of applications.

One important notion related to a component is its interface. The interface represents the component boundary and offers a view on the component by abstracting on implementation details. Collaborating entities as software components are, often exchange messages in order to coordinate actions or simply exchange data. Components communicate through their interfaces and thus interfaces play an important role in the composition mechanism.

In order to successfully interact, entities (either objects, agents or components) need to conform to a certain form of interaction contract. Interaction protocols, generally, describe the entity behavior in terms of allowed message sequences that must be exchanged between entities in order to perform the system intended global behavior.

CBSE integrates interaction protocols at the component interface level. For many years, component interfaces have only described the signatures of their provided or required services. More recently, interaction protocols have made their appearance in more elaborated component models where the component behavior is also explicitly stated. This allows for more trust in the component behavior and a better integration of a specific component in an architecture.

One important issue when decoupling interfaces and implementation is to ensure that the implementation conforms to the interface. While for a pure syntactical approach where only the service signatures are specified the implementation conformance test is simpler, for the cases where behavioral descriptions are also included, the implementation reveals itself harder to be realized. Many state-of-the-art component models or languages do not address this issue. Others rely on code analysis techniques in order to detect non conformance. Component programming languages address this issue by allowing programming with component entities rather than objects. The code conforms to its interface since the interface description is at the same level as the implementation and the compiler verifies some important properties such as communication integrity. Finally, generative techniques assure by construction that the implementation code conforms to the interface description.

2.1 Objectives and Contributions

Component models and languages presented in the state-of-the-art integrate too many concepts or are too specific. In this context, it is very difficult to analyze the consequences of adding new concepts (like interaction protocols) or functionalities in an already defined, full-fledged component model.

In this thesis, we propose a new, general, component model allowing explicit interaction protocols given under the form of Symbolic Transition Systems (STSs). STSs are an extension of LTSs (Labeled Transition Systems) [Arnold 94] where transitions are symbolic. The interest of using finite-state automata based formalism like LTSs and STSs resides in

the automatic verification techniques and tools already developed. The readability and compactness of this kind of formalisms is also usually considered. Unlike LTSs, in STSs, transitions describe classes of possible operations to be effectively executed. Transitions are parameterized with formal input parameters and can be also guarded with parameterized guards (boolean operations). The main benefit of considering STSs is their readability, compactness and expressiveness. In addition, STSs address the well-known state explosion problem that appears in classical LTSs formalisms.

Interaction protocols, whatever formalism they are based on, have as main purpose a better integration of software components in an architecture. The rules specified by the interaction protocols are used in order to detect possible incompatibilities between interacting components. Once the compatibility property, which takes the interaction protocols into account, is proven, component-based architectures are better trusted especially in specific domains or contexts [Magee 99].

Our interest in software component development is related to previous experiment proposals. In [Pavel 04], we experiment with an existing component programming language (see Section 3.3.6 page 86) called ArchJava [Aldrich 02b]. By applying software product lines, an emerging approach to software development, we propose a generative method in order to develop complete software architectures. Guaranteeing that the implementation conforms to the architecture raises new issues with respect to dynamic configuration. We show how this can be solved in ArchJava by making the components auto-configurable, which corresponds to replacing components by component generators. Such a scheme can be implemented in various ways, in particular with a two-stage generator. This solution goes beyond the initial technical ArchJava issue and complements the standard static generative approach to software product line implementation.

Our proposal presented in [Pavel 05a, Pavel 05b, Noyé 05] considers a component model integrating STS protocols and its implementation in the Java programming language. The implementation is based on the use of component controllers (for the realization of the interaction protocol behavior) and communication channels (for inter-component communication). Both of these entities are considered as component first class entities in the proposed model.

Based on these previous experiments we propose a component model we baptize CwSTS (Components with Symbolic Transition Systems). CwSTS is designed as a simple black-box component model (i.e. communication among components is realized exclusively through their interfaces) integrating only some features like a unique service interface describing services both provided and required by the component. In addition to this description (that we will call *structural interface* as it defines only the signatures of the services), CwSTS also proposes a complementary interface information describing the allowed interaction protocol of the component with its environment. We call this information a *behavioral interface* as it specifies the rules that govern the behavior of the component in terms of message emissions and receipts. Our model is based on a hierarchical view of component composition. Two or more components (also called *primitive components*) can be composed into a unique component (also called *composite component*). Regarding its interfaces and behavior, the composite adheres to the same rules of the component model as the initial primitive ones. Further on, the newly obtained composite component, can be used, as if

a primitive component, in other compositions. This corresponds to the *Composite* design pattern [Gamma 95] approach.

CwSTS-IDL is an interface definition language we propose in order to describe component interfaces. Composite component architectures are also described by using this language and include the declaration of subcomponent type instances and connection directives. CwSTS-IDL include a behavioral IDL for the description of the interaction protocols. This behavioral language called SFSP (Symbolic Finite State Processus) is inspired from the FSP (Finite State Processus) [Magee 99] language but keeps only the transition, choice and recursion constructs. In addition, actions are parameterized with type parameters in SFSP.

We follow a generative approach to constructing CwSTS components. One of the benefits of such an approach is that it ensures that component implementations are consistent (conform to) with their interface descriptions. CwSTS-P is the prototype implementation of our model in the Java programming language. Following two generative scenarios, we present the main features of the implementation of our component model. Important features include the encapsulation of a component entity under the form of a Java package, the implementation of a primitive and composite component in order to conform to the rules of the structural interfaces in our component model and finally the details on how the behavioral implementation is realized both at the primitive and composite component level.

2.2 Document Structure

This document is structured in two main parts The first part presents the context of our work. The second part of this document presents the contribution of our work, a component model definition integrating a readable, expressive and easy-to-use formalism for a component designer or user.

The first part presents the context of our work. Firstly, in Section 3.1 we present state-of-the-art component model characteristics. We exemplify this state-of-the-art with some pertinent component model proposals both from the academic and industrial worlds. We also summarize ADLs (Architecture Description Languages) [Medvidovic 00b] proposals (see Section 3.3.4). An ADL is defined as a formal or informal notation, either textual or graphical, allowing to specify software architectures and accompanied with specific tools [Medvidovic 00a]. The result of using an ADL is mostly an abstract view over an application in terms of *components*, *connectors* and *configurations*, rather than a concrete view over the implementation details.

While architectural analysis in existing ADLs may reveal important architectural properties, these properties are not guaranteed to hold in the implementation. In order to enable architectural reasoning about an implementation, the implementation must conform to its architecture. The Component-Oriented Programming paradigm (see Section 3.3.6) is a recent proposal for developing components by programming directly with component entities rather than objects. Component programming languages (or architectural programming

languages) are presented in order to indicate the research directions of this new trend.

The coordination paradigm is addressed in Section 3.3.7. Coordination is a research area with an interdisciplinary focus. The coordination paradigm implies that programming a distributed or parallel system can be seen as the combination of two distinct activities: the actual *computing* part comprising a number of processes involved in manipulating data and a *coordination* part responsible for the communication and cooperation between the processes.

Secondly, we present the state-of-the-art in the specification of interaction protocols. Different formalisms and component models integrating these formalisms are discussed. Section 4.2.1 presents the process algebra formalisms. The term *process algebra* refers to a family of specification techniques particularly well suited to describing systems of concurrent communicating components. Process algebras describe the process interactions in terms of calculus (an ensemble of rules defined on a set of process expression construction operators).

Next, Section 4.2.2 depicts the formalisms based on behavioral types. Behavioral types specify not only a set of messages (structural interface) to be exchanged between entities in order to communicate but also constraints on acceptable sequences of these messages.

Finite State Machine (FSM) formalisms are presented in Section 4.2.3.FSM-based formalisms are generally assumed to be complete descriptions of system behavior at some level of abstraction. From a component modeling perspective, the system behavior is what an external entity can observe about the interactions of the system with its environment. This is usually the messages the system (black-box component) exchanges with its environment in terms of emissions and receipts. A finite state machine describes the set of all possible traces a component can produce when interacting with its partners.

Finally, temporal logic is depicted in Section 4.2.4. In computer science, temporal logics are used to specify and verify properties like safety, liveness and fairness. Due to the fact that the verification and verification of the specifications are very expensive both in terms of time and space, there are only a small number of usable temporal logics.

The main critics that can be done regarding some of these formalisms is the abstraction level, much too weak to describe higher level entities like components. Another drawback is the fact that some important properties like deadlock are not decidable. Some other formalisms sacrifice expressiveness in order to increase decidability. Developing object models, important notions like substitutability and refinement made their appearance, easing the way to developing satisfactory component models integrating protocols.

The second part of this document presents the contribution of our work, a component model definition integrating a readable, expressive and easy-to-use formalism for a component designer or user. We start by describing our component model from an informal point of view (see Section 5.2). The components, their interfaces, the composition rules, and the details on our STS-based interaction protocols are exemplified by a running example. We formalize our model in Section 5.3. The formal definitions of a component, the structural and behavioral interface are described together with some important properties like the component compatibility and substitutability (see Section 5.3.3). CwSTS-IDL is presented in Section 5.4. SFSP, our interaction protocol description language is presented

in Section 5.4.3.

CwSTS-P, the prototype implementation of our model is presented in Section 6.1. The encapsulation of a component implementation under the form of a restricted Java package is explained in Section 6.1.2. Section 6.1.3 presents the standard structure of a primitive component together with its constituent parts. The configuration, instantiation, and execution mechanism is depicted in Section 6.1.3.2. The structure of a composite component is presented in Section 6.1.4 and the behavioral composition mechanism implementation in Section 6.2.

We conclude and present some possible evolutions related to our work at the end of this second part in Chapter 7.

Chapter 3

Component Models and Languages

In this chapter we present state-of-the-art component models and languages. We start from a history of component-based software engineering and proceed to present component models characteristics along with examples both from the academic and industrial worlds. Next, we present the research fields related to software architecture: software connectors, Architecture Description Languages (ADLs), the emerging paradigm of Component-Oriented Programming (COP), Service-Oriented Architecture (SOA) and Coordination.

Contents

3.1 From Objects and Modules to Software Components

By the end of the sixties, it became obvious that the way of developing software needed to be reconsidered. At that time, the applications were made on demand and developed from scratch. As the demand was constantly augmenting, the first drawbacks of this approach to software development made their appearance: the software quality was poor and the exploitation and maintenance costs were high. In addition, the rapid changes on application development, imposed by the market dynamics, could not be performed as rapidly as required. Time to market and the clients pressure resulted in the fact that the documentation provided with applications was usually incomplete or even incorrect. By this fact, it was very difficult, if not impossible, to reuse parts of the application in other projects.

The notion of software component was first mentioned by McIlroy [McIlroy 68] in his speech to the OTAN conference in 1968 and the idea of software reutilization emerged as a solution to the above-mentioned problems. This would have been realized by the creation of a software industry where applications would have been created by reusing software components rather than creating them from scratch [McIlroy 68].

Important developments in this direction were realized starting from the end of the nineties. By this time, the weaknesses of the object paradigm especially concerning reutilization were fully revealed. In the field of object-oriented development, the main reutilization mechanism is inheritance. This mechanism induces a strong coupling between the base class and the derived classes. This drives to the problem identified under the name of *Fragile Base Class* [Szyperski 96, Mikhajlov 98]. Fragility comes from the fact that one modification at the base class level can result in a problem at the derived class level and is due to a white-box approach when constructing software.

The Component-Based Software Engineering (CBSE) paradigm [Heineman 01a] is trying to overcome the limitations of the object-oriented approach. The CBSE approach proposes a better separation between the implementation of a software component and its interface. The interface defines a contract between the component and its environment. It clearly describes what the component provides to and what it requires from its environment in terms of services. In more elaborated models, the interface also describes the details of the interactions between the component and its partners. This kind of description is generally accepted under the term of *behavioral description*. Each component is an indivisible black-box entity that can be composed and deployed. The construction of applications is realized by assembling components. Adopted in practice, the component paradigm improves the speed of producing reliable, high quality applications.

Besides objects, components also relate to modules [Wirth 77, Mitchell 79] in that the component technology unavoidably leads to modular solutions. Many modularity criteria go back to Parnas (1972) and include the principle of maximizing cohesion of modules while minimizing dependencies between modules. While modularity is a prerequisite, rules beyond the traditional modularity criteria are needed to form components rather than just modules. For example, modules can be built to use global (static) variables to expose observable state, while a component defined as in definition 1 at page 60 does not allow non-abstract observable state. Furthermore, modules tend to statically depend on

implementations in other modules by importing direct interfaces from other modules. For components, such static dependencies are not recommended to enable flexible composition using multiple implementations of the same interface [Szyperski 02].

The CBSE approach encompasses developments in component models definition, Software Architectures, Architecture Description Languages (ADLs), Component-Oriented Programming, etc. However, the current state-of-the-art in component technology does not solve all the problems related to software development. This is in phase with the prediction of The Mythical Man Month [Brooks 75] foretelling that there is no miracle solution to software development as it is intrinsically complex. Further developments still need to be realized to fulfill the CBSE promising results even if some practices and approaches are better than others in achieving those goals.

3.2 Component Models

Particular component characteristics are described by models (component models). A component model describes what a component is (by defining its constituent parts) and says how components can be eventually composed (by following some composition rules). It also describes the component life cycle and the roles associated with different actors in the development and exploitation of applications.

One important notion related to a component is its interface. The interface represents the component boundary and offers a view on the component by abstracting on implementation details. Collaborating entities like software components, often exchange messages in order to coordinate actions or simply exchange data. Components communicate through their interfaces and thus interfaces play an important role in the composition mechanism. There are two categories of component models: academic and industrial.

Academic component models focus on key concepts related to components like the definition of components and their interfaces and on their composition. They also explore different properties like, for example, the substitutability, the compatibility and the dynamic adaptation of their behavior. Industrial approaches focus on the production, distribution and execution of industrial applications. They are successful in providing solutions (like distribution, transactions, persistence and security) in the context of distributed applications.

In this section we start (Section 3.2.1) by presenting the main generally accepted characteristics of a component model. Next, we detail the two component model categories: academic and industrial (Section 3.2.2 and Section 3.2.3).

3.2.1 Component Model Characteristics

3.2.1.1 Component Notion

There is no unique definition of what a component is. Instead a multitude of definitions coexist [Szyperski 02, Brown 98, Heineman 01b, Marvie 02, Larsson 00]. However, it is commonly accepted that a component has at least the following properties:

1. it is a unit of composition,

2. it has no (externally) non-abstract observable state,

3. it is a unit of independent deployment.

One commonly accepted definition of what a component is, was formulated for the first time at the 1996 European Conference on Object-Oriented Programming (ECOOP) as an outcome of the Workshop on Component-Oriented Programming.

Definition 1 (Software Component) *"A software component is a unit of composition with contractually specified interfaces and explicit context dependencies only. A software component can be deployed independently and is subject to composition by third parties." [Szyperski 02]*

This definition has several implications. One important implication is that a component needs to encapsulate its implementation and interact with its environment only by means of well-defined interfaces (black-box approach). Another implication is that the component is prepared to be composed with other components in order to obtain an application (composition rules are required). This definition also implies the existence of a life cycle associated to a component entity, traditionally consisting of at least four phases: creation, assembling, deployment and execution. A less explicit notion implied by this definition is that of an environment where the component can exist at different phases of its life cycle.

Some other definitions of a software component have been given. In fact, different authors gave different definitions and meanings of what a component is. This drove to confusions and the lack of precise discussions around components. In addition, the terms of *component*, *component type*, *component instance* or *component implementation* were used one for the other. Section 3.2.1.4 at page 66 gives a precise classification of the terms we use in this document to designate a software component during its lifetime.

3.2.1.2 Component Interface

In the general field of software systems, an interface defines the communication boundary between two entities, such as pieces of software, hardware devices, or users. It generally refers to an abstraction that an entity provides of itself to the outside world. This interface separates the methods for external communication from internal operations, and allows the entity to be internally modified without affecting the way outside entities interact with it. It also allows the entity to provide multiple abstractions of itself. It may also provide a means of translation between entities which do not speak the same language, such as between a human and a computer.

In object-oriented programming, an *interface* is what objects use to communicate with each other. These are definitions of methods and values which the objects agree upon in order to cooperate. The interface of an object is usually a description of:

1. the messages that are understood by the object,

2. the arguments that these messages may be supplied with, and

3. the types of results that these messages return.

4. the invariants that are preserved despite modifications to the state of the object.

5. the exceptional situations that will be required to be handled by clients of the object.

Based on this definition of interface for objects, the component interface can be defined following two complementary points of view, that is the collection, respectively the contractual, point of view. By the collection point of view we mean the fact that we can see the interface of a component as the set of operations the component provide its partners with. From a contractual point of view, we see a component interface as a contract between the component and its partners. With respect to the contractual point of view, a component interface also integrates what is called *required operations*, in addition to the *provided operations*. While the provided operations define what is offered by the component, a required operation defines what is needed by the component in order to execute. This kind of interface is more adapted to describe the two roles client-server inherently associated to a component. The definition of a component interface integrating both provided and required operations is in conformity with the component definition 1 at page 60, where context dependencies have to be made explicit.

The definition above takes into consideration only operation signatures, either provided or required. Nowadays, this simple definition is considered obsolete. Antoine Beugnard *et al.* [Beugnard 99] propose a taxonomy of contracts related to component interfaces. According to this taxonomy there are four levels of contracts related to the exchanges of a component with its environment:

1. *basic* - this is the syntactic level, as described in the informal definition we gave at the beginning of this subsection. This level considers the syntactical properties of the names, parameters types, return types and exceptions.

2. *behavioral* - this level considers the properties that can be specified using pre-conditions, post-conditions and invariants. These properties are linked to an operation (method in the object world) and specify the parameter types and the dependencies between the values of these parameters. One can also see in these properties the *intent* of the method.

3. *synchronization* - this level deals with properties concerning the components interactions. These properties cannot be expressed only by using pre- and post-conditions on a method. They consider the component interactions with its environment. As these interactions influence the internal behavior of the component, we can describe their execution as a sequence of steps that are to be followed. The indeterminism related to operation calling and the execution concurrency have also to be taken into account when describing these properties.

4. *quantitative* - this level corresponds to all nonfunctional properties as the quality of service, resource management or response time. These contracts are called *Quality of Service* (QoS) contracts. They are the most difficult to express and analyze as they usually depend on the properties to be verified, the environment and the execution context, pieces of information that are usually not known at the moment of architecture description.

All current component models deal with the basic level as it is based uniquely on operation signatures and considers only functional properties. This is realized by using *Interface Description Languages* (IDLs) that let the component designer specify the operations to be performed, the input and output parameters that are required by the component and the possible exceptions that might be raised during execution. This is also the only level of contracts not allowing for negotiation between the interacting partners.

Levels two to four are more or less dealt with in the current state-of-the-art component models. This is due to the complexity in expressing and analyzing this kind of properties.

The behavioral level is closely related to Design By Contract (DbC) [Meyer 92]. This method is based on the theory of abstract data types [Ruane 84] and on the contract metaphor in the juridical sense of the term, binding two parties together. The idea behind DbC is that software entities have responsibilities towards the entities they interact with. These responsibilities are based on formalized rules: functional specifications, or *contracts*, are created even before they are really implemented. The interactions are ruled by these contracts. Work in this domain includes the Eiffel language [Meyer 91], OCL for UML [OCL 03, UML 03], and JML [Leavens 00]. The design by contracts methodology was initially developed for the object-oriented design and programming fields. Some approaches like [Collet 05, Carrez 03] explored the inclusion of contracts into component models.

The synchronization level contracts deal with the sequencing of messages related to component communication. In a distributed concurrent environment, components are usually autonomous processes simultaneously executing and exchanging information. Each component is located on different machines and evolve independently. In order to communicate with another component, a synchronization takes place when exchanging messages. We talk about asynchronous parallelism [Garavel 03]. State-of-the-art component approaches will be given in the following subsection.

The last level of contracts from the taxonomy deal with non-functional properties associated to the notion of QoS. There are many approaches in the direction of specifying quality of service properties at the component interface level. We point out for example [Staehli 03, Hoschka 98, Mercouroff 97, Frølund 98].

Blackbox, whitebox, graybox or glassbox component flavours

According to definition 1 page 60, a component interacts with its environment exclusively through its interfaces. As the interface is abstracting from the implementation, a component communicating exclusively through its interfaces and not revealing its implementation details is said to be of black-box type. Objects communicate through their interfaces (defining their public methods) but can also communicate through their instance variables. Components based on other definitions than definition 1 could also expose implementation details besides their interface descriptions thus breaking encapsulation. For example, in the JavaBeans model, a component allows communication through its interfaces but also through its instance variables. Depending on the degree of implementation details offered by the component, we can speak about a gray-box or a white-box component type [Büchi 97]. While JavaBeans [Hamilton 97] components are of white-box type, other

models and languages like, for example, Oberon [1] describe components in terms of pre- and post-conditions on the input and the output of the services provided by the component [Findler 01]. This kind of approach is usually insufficient to describe the component behavior [Wegner 97]. Some authors further distinguish between white-boxes and glass-boxes, with a white-box allowing for manipulation of the implementation and a glass-box merely allowing study of the implementation.

Operations, Services, Interfaces, Ports

The terminology related to component operations, services, interfaces and ports abounds in the literature. Despite the fact that there is no general consensus on the meaning of these terms, from a collection point of view, we can consider as valid the following definitions [Legond-Aubry 03]:

- An *operation* is a component action. This action is called by a client component and is executed by a server component.
- An *interface* is a set of operations united by the component designer following a coherent point of view. It is considered that an operation cannot be part of more than one interface, that is, interfaces define a partition of the set of operations.
- A *service* is the set of accessible component operations.
- A *port* is the union of some interfaces following a coherent point of view. The subjacent idea related to ports is that if an interface belonging to a port is used by another component it is likely that another interface is also used in the same interaction between the two components. Another idea is that a port is a designation entity, at least at execution time.

The contractual point of view completes these definitions with the point of views associated to the roles of client and server:

- A *required* operation (interface, service, port) characterizes the needs of the client. The required character implies the existence of the specified operation (interface, service, port) in the component environment. In a required interface, service or port, all specified operations are required.
- A *provided* operation (interface, service, port) characterizes the realization proposed by the server. The called operation realizes a computation defined by the specification associated to the provided operation. In a provided interface, service or port, all specified operations are provided.

One particular definition is that of a port in UML 2.0. A UML 2.0 port assembles required and provided interfaces. This definition is relevant as the notions of *network port* or *distributed port*, for example, assemble the input and the output communications. Thus, it is appropriate to introduce a structure allowing the specification of both provided operations and required operations.

1. http://www.oberon.ethz.ch

3.2.1.3 Composition

Composition is a mechanism allowing the construction of complex components (*composite* components) starting from simpler components (*base* components). Figure 3.1 depicts the case of a composite component *Architecture*, obtained by composing components *C1* and *C2*.

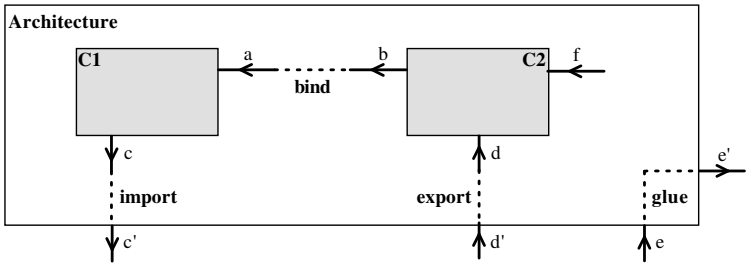


Figure 3.1: Architecture composite component.

Component *C1* provides the *a* operation and requires the *c* operation. *C2* requires the *b* operation and provides the *d* and *f* operations. The obtained composite component *Architecture*, provides the *d'* and *e* operations and requires the *c'* and *e'* operations. The composition is obtained by binding operations provided by one component to operations required by a *partner* component or by delaying the binding of an operation (from the subcomponent level) by making it explicit in the interface of the composite. Note that it is allowed not to bind an unused provided operation (as the provided operation *f* of *C2*).

Composition schemas.

With respect to the composition mechanism we can identify four configuration schemas:

- *binding* a required operation to a provided one - this is the base mechanism associating a required operation in one subcomponent with a provided operation in another subcomponent.
- *exporting* a provided operation from the subcomponent level to the composite level - this mechanism makes it possible to associate an operation provided by one of the subcomponents with a corresponding operation at the composite level. While the subcomponent operation has an implementation, the operation of the composite is merely forwarding calls from its clients to the subcomponent service.
- *importing* a required operation from the composite level to the subcomponent level - instead of binding the required operation to a provided one, this mechanism makes it possible to delay this binding by exposing it (under the identification of an external operation) for further composition schemas.
- *gluing* a provided operation to a corresponding required operation of the same composite - this schema allowsfor passing through a composite component; instead of providing an implementation for the provided operation, the component delegates this responsibility to another component.

Figure 3.1 depicts the case of an architecture where the only operations exposed by the composite correspond to operations defined by subcomponents. In this case, we talk

about composites that only realize configurations. No additional functional code (implementations for operations at the composite interface level) is present in the composite, this is used only to compose subcomponents. Operations *e* and *e'* do not have correspondents inside the composite, they do not correspond to some functional code inside the composite either. Composites implementing both architectures and their own interfaces (without any correspondence with the operations of the subcomponents) belong to particular component models. In both cases, however, this type of composition is called *hierarchical* as it makes it possible to obtain composites starting from base components, composites that are likely to be used in further compositions as if they were base components. This is different from other approaches where the composition is *flat*. For example, the component model presented in [Farias 03] proposes a composition mechanism where components can be obtained by regrouping the interfaces of the base components into a new interface, and by applying regrouping rules to the implementation. The resulting component can then be used by clients by directly calling operations from this interface.

Composition can also be used for *connection-oriented programming* [Szyperski 02]. The connection-oriented programming style is useful when "wiring" prefabricated components or objects provided by such component models. This is in analogy with the integrated circuits (IC) from which the CBSE approach got inspired. The connection-oriented style implies the view of connections as entities of their own. They are symmetrical (interfaces describe both provided and required operations). Safety is of major importance in this approach because of the very late binding of components.

Compatibility.

One important concept that intervenes in the composition mechanism is the compatibility of the operations we want to associate with each other. Indeed, in order to allow a valid configuration, the operations we bind, export, import or glue together must be *compatible*. With respect to the taxonomy presented in Section 3.2.1.2 and the *binding* schema, different component models define compatibility in function of the level of conformance to the taxonomy. The first-level contracts, the syntactical ones, consider that a provided operation is compatible with a required operation if they have the same *name* and *signature*, where *signature* is the type and name of input and output parameters and the return type of the operation. In addition to this constraint, interface contracts of the second level, consider that we can connect two corresponding operations only if the descriptions (given in terms of pre-, post-conditions and invariants) match. More evolved models also define compatibility of behavioral descriptions [Yellin 97]. From the QoS level point of view, the compatibility is harder to define and verify as it depends on properties that are possibly not known at the moment of composition [Ozanne 07].

One important concept related to that of compatibility is subtyping. Subtyping makes it possible to substitute a type (interface type) with a subtype without breaking the compatibility rules. Usually, in order to allow for substitution, the type-subtype relationship must adhere to the variant-covariant specifications. Less strict definitions of compatibility also consider the type-subtype relationship when connecting corresponding operations.

Incompatible operations cannot be directly connected. Instead, an adaptation can be realized under certain conditions. The incompatibility is mediated by an *adaptor* that takes

in charge the adaptation of interfaces (both syntactical and behavioral, when present). The adaptor is usually implemented as *glue code* [Cherinka 98] and, in certain cases, needs more time to be developed than the component itself. ADLs address this issue by including the adaptation code into connector entities used to actually connect the two components.

Models that regroup interfaces into ports, define the compatibility at the port level following the same guidelines as previously presented.

3.2.1.4 Life Cycle

The component life cycle defines the steps to follow when constructing and executing an application based on components. Traditionally, the life cycle consists of four phases: *creation*, *assembly*, *deployment*, and *execution*.

1. Creation. During this phase, component interfaces are defined and an implementation conforming to the interfaces is realized. Once ready, all the component constituents parts (interface descriptions and implementation entities) are packaged to be distributed to tiers.

2. Assembly. An application based on components is obtained by assembling existing components into a final architecture. The assembly phase also consists in defining the assembling configurations needed in the deployment phase. When assembling components, each individual component can also be configured to fit the specific architecture.

3. Deployment. The deployment of an application takes in consideration a configuration specifying details like, for example, the initial values for instances (component instances).

4. Execution. During this phase, the application is installed by the administrator and is used by clients.

Depending on implementations, the third and fourth phases also include instantiation of components. In case of EJBs (see below) component instantiation takes place after the application is deployed and before the first use of the components.

The life cycle makes it possible to define roles associated to each of its phases. For example, in the EnterpriseJavaBeans component model, we can find the following roles:

- Entreprise Bean Provider
- Application Assembler
- Deployer
- EJB Server Provider
- EJB Container Provider
- System Administrator

While the *Entreprise Bean Provider* and the *Application Assembler* roles relate respectively to the creation and the assembly phase, the next roles transgress the deployment and execution phases. In EJB, the deployment takes place into an environment called container. The container is integrated into an application server and the administrator is in charge of surveying the execution of the application.

Software Components, Component Types and Component Instances. The life cycle phases are generally the same in all component models. However, there are some differences in the definition of each phase depending on proposed models [Brown 96, Morisio 00]. One potential source of misunderstandings is related to the use of the term *software component*. In fact, this term is used through all the phases of the life cycle. It, naturally, takes different forms. In this document we will talk about *component type* to denote the information related to the description of the required and provided services and, in case of a composite, to denote a configuration. We call *component implementation* the set of sources and/or binaries allowing the realization of a component. Finally, we talk about *component instances* to denote an existing entity executing in a specific environment. A component instance is deployed and is characterized by a unique reference, a component type and an implementation of this type.

3.2.2 Academic Models

Among current propositions of academic component models we can cite [Kenney 95, Bellissard 95, Magee 95, Medvidovic 96, Plašil 98, Flatt 98, Siegel 00, van Ommering 00, Cardone 00, Choppy 01, McDirmid 01, Seco 02, Sreedhar 02, Aldrich 02b, Coupaye 02]. Many of these models also integrate an ADL (see Section 3.3.4 page 79) or interaction protocols (see Chapter 4 page 93). In this section we will discuss Fractal an extensible component model and SOFA, a component model integrating interaction protocols. Fractal also integrates an ADL, but our focus is to present its main characteristics and discuss its results.

The choice of these two models is sustained by the fact that both of them are general purposed component representative for the current state-of-the-art in component models. They have many things in common but they also differ in some significant points: static versus dynamic architectures, use of explicit connectors or not, etc.

3.2.2.1 Fractal

The Fractal component model is developed by the ObjectWeb [2] consortium (led by France Telecom R&D) and by INRIA (*l'Institut National de Recherche en Informatique et en Automatique*). Fractal goals are to authorize the definition, configuration and dynamic reconfiguration of an architecture based on components and to provide a clear separation of functional concerns from the non-functional ones. Fractal is a also an extensible component model allowing the developer to add structural and behavioral elements, and go beyond the possibilities defined by the base model.

2. http://www.objectweb.org

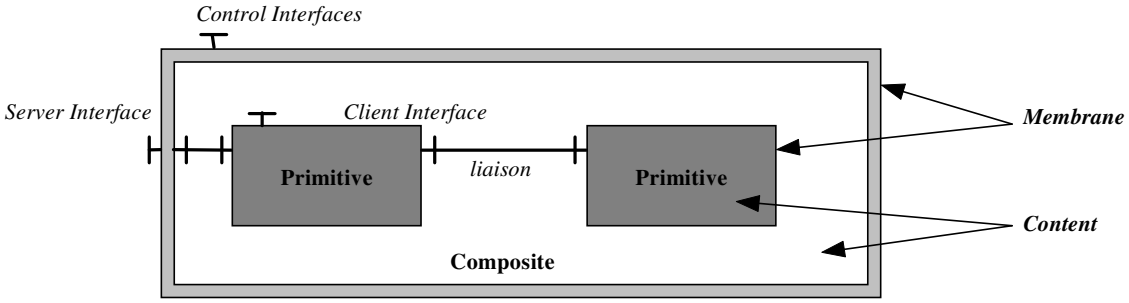


Figure 3.2: Fractal Architecture.

Fractal is defined on two levels of abstraction (and two corresponding models). The general model describes the notion of a component system from an abstract point of view. This model describes the notion of cell having a content (plasm) and a membrane. The model is inspired by a metaphor between a component and a biological cell. The concrete model specializes the general one to obtain a more operational model. This specialization introduces a type system inspired from the target programming language.

Components, interfaces and composition.

A component in Fractal is made of a membrane and a content (Figure 3.2). The membrane is constituted by the a set of interfaces defining the operations required and provided by the component. The content makes it possible to distinguish two types of components: primitives and composites. The content of a primitive component is represented by a software module realizing the component services. The content of a composite component is an assemblage of primitive components satisfying the services defined by the composite interfaces.

Fractal interfaces are of two types: server interfaces and control interfaces (the membrane corresponds to the entire set of control interfaces). An interface is composed of a name, a signature and a type. While the business interfaces represent the points of interaction with the environment of the component, the control interfaces deal with non applicative properties like the management of the life cycle or the connections of the component.

The composition is possible by using what the authors call *liaisons*. A *liaison* is a connection between a client interface and a server interface (Figure 3.2). Because of the strongly typed system, a server interface can be connected to a client interface only if its type is identical or if it its type is a subtype of the client interface.

The membrane allows the definition of both external and internal interfaces. Internal interfaces are accessible only from the inside of the component (the content). An internal interface exists symmetrically with an external interface and is used to create pass-through links from the membrane to the content level without losing the *liaison* semantics.

The Fractal model makes it possible to construct hierarchical architectures. However, the specificity of a Fractal architecture is that a primitive component can be *shared* by multiple composite components.

Life cycle and dynamic reconfigurations.

The phases of creation and deployment are realized in a traditional way (Section 3.2.1.4 page 66). At execution time, the API provided by Fractal allows the navigation and introspection of the structure of components: to discover component interfaces and content, and navigate along the *liaisons* between interfaces.

In addition it is possible to dynamically modify the structure of an application by modifying the content of composite components and creating or destroying *liaisons*. Component instances can be created at execution time either using a *Factory* or by using *Templates*. A component instance is active as long as there is at least one liaison between its interfaces and the environment.

Fractal implementations.

There are many implementations of Fractal. Julia [Bruneton 04] is the reference implementation of Fractal in the Java programming language. Think [Fassino 02] is a C implementation targeting the development of embedded applications and extensible OS cores. FracNet [Escoffier 05] is an implementation for the .NET framework. AOKELL [Seinturier 05], PLASMA [Layaïda 05] and FracTalk [3] implement Fractal in AspectJ, C++ and SmallTalk, respectively.

Behavioral descriptions in Fractal.

Fractal is an extensible component model. It makes it possible to extend the model in order to add structural or behavioral elements. Different approaches target to integrate "contracts" at the interface level. ConFract [Collet 05] proposes the use of assertions at the interface level. Barros [Barros 05] also proposes to use LTS [4] to describe the component behavior.

3.2.2.2 SOFA

SOFA [Plašil 98] (SOFtware Appliances) is a component model natively allowing the specification of behavior protocol at interface level. A SOFA component can be of either *atomic* or *compound* type. While an atomic component relies on an implementation language, a compound component solely defines a component hierarchy (architecture). A compound component is specified by using an interface description and an architecture description. The interface description gives a black-box view over the component by specifying only the required and provided services. The architecture description gives a grey-box view over the component by specifying the subcomponents and their interconnections. Interconnections are realized by employing four types of connectors: *binding*, *delegating*, *subsuming* and *exempting*. SOFA provides a CDL (Component Definition Language) in order to describe interfaces and architectures.

Interfaces in SOFA allow the definition of behavior protocols. A behavior protocol describes all the service traces a component can accept and are specified by using a formalism

3. http://csl.ensm-douai.fr/FracTalk
4. Labeled Transition System

based on regular expressions. The model provides means to verify the coherence of the architecture but not the coherence of the implementation with its specification.

A *SOFAnode* represents a deployment and execution environment for SOFA components. A network of SOFAnodes constitutes a *SOFAnet* representing the runnable configuration of an application. SOFA model also allows for the dynamic update of a component, mechanisms to distributed deployment and version control.
In order to allow for a dynamic update of a component, SOFA defines a component type called *DCUP* [Plašil 98]. This type of component comprises two parts. A permanent part which is specific to each component version and a replaceable part being renewed at each component update.

3.2.2.3 Fractal and SOFA Evaluation

Fractal is a general-purpose component model. It uses a hierarchical component model without connectors. Connectors can be simulated using "normal" components (the Fractal specification even instructs to do so), however this results into rather unclean and difficult to comprehend architectures mixing different levels of abstraction. Fractal separates components functional and non-functional (control) parts. The non-functional part is managed using controllers, which are from the architectural point of view provided interfaces. Fractal also introduces the concept of shared components, i.e. a single subcomponent instance shared by several composite components. Such an approach easily allows for runtime changes of an architecture, but it breaks a component encapsulation hierarchy and can result in clumsy and uncontrollable architectures. By itself, Fractal is just a specification defining a set of component features and standard interfaces, and it has a number of implementations.

The SOFA component model is like Fractal a general-purpose component model. It also uses a hierarchical component model but with connectors (and therefore with multiple communications styles). In addition, these connectors allow for transparently distributed applications. Component behavior can be described using behavior protocol and these can be subsequently used to verify component composition and communication. For describing components and architectures, SOFA uses its own ADL. Similarly to Fractal, SOFA components also exist and may be instantiated at runtime. The weak points of SOFA comprise no support for dynamic changes of an architecture (it just supports a dynamic update of a single component), not clearly separated and non-extensible control part of components, and a limited set of communication styles.

Both SOFA and all implementations of Fractal create a component platform over the Java platform; in fact they are Java libraries. At runtime, instances of components exist but they are mapped to a set of Java classes. As components in SOFA and Fractal are implemented in pure Java, they can be much more easily integrated with other legacy systems.

3.2.3 Industrial Models

Industrial component models focus on the implementation of system services for the programming of distributed application. Development of applications in the large, is sustained by the adoption of a *three-tier* architecture design approach and the use of system services like the distribution, transactions, persistence and security. There are three main actors in the industrial world proposing industrial component models: OMG [5] with the CCM [6] component model, SUN [7] with the EJB [Matena 06] component model and Microsoft with the couple COM+ (component model)/.NET architecture.

Unlike academic models, industrial propositions generally define all the component model characteristics as presented in Section 3.2.1 page 59. The specifications related to these approaches describe component types, the execution environment and deployment strategies.

3.2.3.1 CORBA Component Model (CCM)

OMG (acronym for Object Management Group) is a consortium of more than 800 member companies in the computing industry. OMG original aim was the standardization of "whatever it takes" to achieve interoperability at all levels of an open market for "objects". The main result of their efforts is the definition of CORBA (Common Object Request Broker Architecture) and related standards. From the very beginning, the goal behind CORBA was to enable open interconnection of a wide variety of languages, implementations and platforms. The downside of OMG approach is that individual CORBA-compliant products cannot interoperate on an efficient binary level. Instead, they must engage in costly high-level protocols.

At the very heart of CORBA stands the ORB (acronym for Object Request Broker) which is essentially a high level, transparent, remote method invocation service. The most common use of ORBs in industry is to replace sockets and remote procedure calls in applications spanning several server machines. Beside ORBs, a set of invocation interfaces and a set of adapters represent the core of CORBA. An Interface Description Language (OMG IDL) is used to describe object interfaces and a compiler generates stubs/skeletons starting from these descriptions. In addition, language bindings from OMG IDL to general programming languages like Java, C, C++, Smalltalk, etc. are also provided. Stubs and skeletons are good solutions when dealing with regular method invocations. CORBA also provides a dynamic invocation interface (DII) and a dynamic skeleton interface (DSI) to allow the dynamic selection of the operation to be invoked at execution time either the client's end (DII) or at the server's end (DSI). With CORBA 3, OMA (acronym for Object Management Architecture) also adds three new areas of standardization: a set of common object service specifications (CORBAservices), a set of common facility specifications (CORBAfacilities), a set of application object specifications and the CCM (CORBA Component Model).

5. Object Management Group - http://www.omg.org
6. Corba Component Model
7. http://www.sun.com/

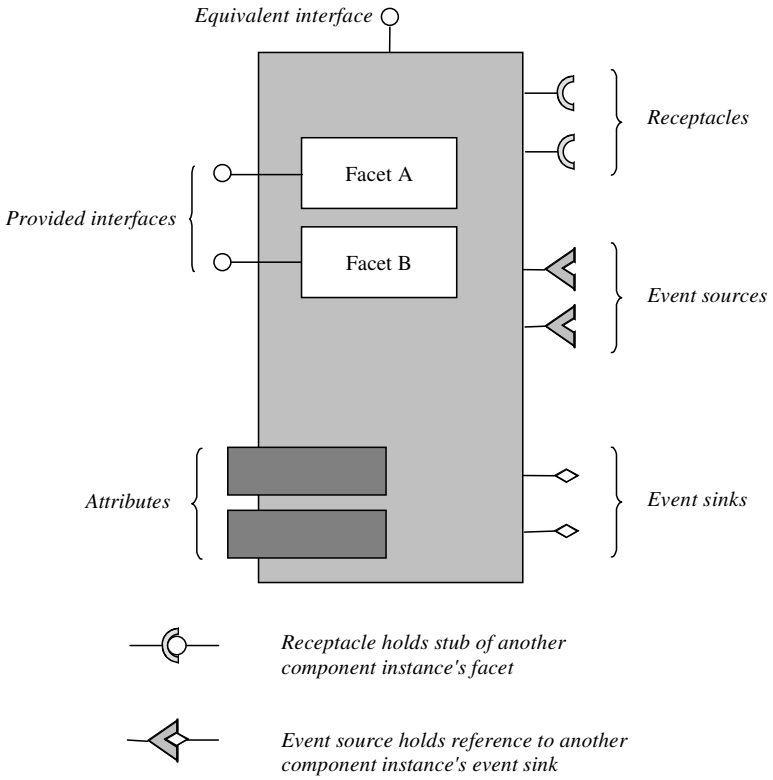


Figure 3.3: CCM Component.

The CORBA Component Model is an extension of Entreprise JavaBeans (see Section 3.2.3.2). It introduces several novel features, promises a fully compatible embedding of existing EJB solutions and aims to maintain the original CORBA goal of being both language and platform independent. A CCM application is an assembly of CCM components, each of which may be custom-built or off-the-shelf, in-house or acquired. Entreprise JavaBeans components and CCM components can also be combined in a single application.

A CCM component is characterized by a number of features like (Figure 3.3):

- ports (facets, receptacles, event sources and event sinks) representing the provided (facets) and required (receptacles) interfaces. Event sources and sinks are similar to facets and receptacles but they are connected to event channels instead of to each other.
- primary keys, which are values used to allow client identification of the instances.
- attributes and configurations, which are named values exposed via accessors and mutators.
- home interfaces providing factory functionality to create new instances.

A special facet (interface) of a CCM component is the equivalent interface enabling the

navigation between the different facets of a CCM component. A CCM component can also be classified into one of the fourth categories:

- *service* - are instantiated per incoming call and cannot maintain state across calls.
- *session* - instances maintain state for the duration of a transactional session.
- *entity* - they have persistent instances, correspond to entities in database and can be accessed by presenting the database entity's primary key.
- *process* - instances lifetime corresponds to the lifetime of some processes they are servicing and have persistent state.

In CCM as in any industrial approach, a component unit is an archive file containing the component definition (compiled files) and a deployment descriptor. More specifically, in CCM, a component unit contains:

- the *component implementation*.
- the corresponding *IDL description*.
- the *component descriptor* (CCD, CORBA Component Descriptor) specifying the services (like persistence, transactions, security, etc.) the container must provide.
- a *property descriptor* (CPF, Component Property File) defining the component attributes default values.
- a *packaging descriptor* (CSD, Corba Software Descriptor) giving informations like author, licence, etc.

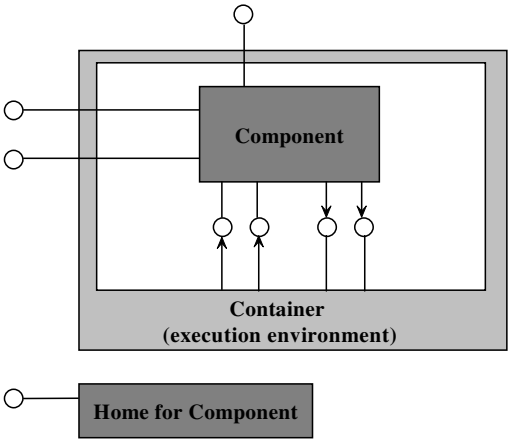


Figure 3.4: CORBA Component Model Execution Environment.

Each component instance is placed inside an execution environment (see Figure 3.4). In the case of CCM this environment is represented by a CCM container. The container is

responsible for providing the component with the required services (mainly transactions, security, persistence and notification services). Containers are themselves coordinated by an application server.

The standards described by CORBA 3 are very numerous and the overall architecture tends to be very complex. These are, probably, the reasons for not having many CORBA-compliant implementations available at this time. Vendors like BEA, IBM IONA or Borland tried to provide developers with CORBA-compliant application servers, containers and IDEs (Integrated Development Environment) but the variety and, some times, the rapid changes in standards specifications made that they are not fully compliant. Concerning CCM, at this time even fewer compliant server-side implementations (like the K2 Component Model [8]) exist.

3.2.3.2 Component Models in the Java World

At the end of the nineties, SUN Microsystems started elaborating several component models designed to address the development of distributed applications. These models are all based on the Java [Arnold 98] programming language. Currently, the Java universe counts five component models: applets and JavaBeans (parts of J2SE [9]), Enterprise JavaBeans, servlets/JSPs and application client components (parts of J2EE [10]). While applets target developing client-side applications, J2EE define standards for component models targeting server-side development.

Applets were the first Java component model, aiming at downloadable lightweight components that would augment websites displayed in a browser. The security model related to applets was too tight and later developments in browser standards and technologies made the applet technology obsolete.

The JavaBeans models is rather an object model than a fully fledged component model (according to characteristics defined in Section 3.2.1 at page 59). JavaBeans focus on supporting connection-oriented programming (Chapter 10 in [Szyperski 02]) and is useful on both clients and servers. The proposed model is of white-box type as the communication between JavaBeans objects can be realized also by fields and events notifications in addition to interfaces (public methods). The event mechanism adheres to the publish/subscribe paradigm [Eugster 01, Wang 02]. JavaBeans support two types of variables representing the source of events: *bound properties* and *constraint properties* and the event mechanism is used as a composition mechanism.

Enterprise JavaBeans focuses on container-integrated services supporting EJB beans (components) that request services using declarative attributes and deployment descriptors. EJB follows an entirely different path than JavaBeans. There is no provisions for connection-oriented programming at all (adding this is one of the main improvements of CCM over EJB). EJB deals with the *contextual composition*. Contextual composition is about the automatic composition of component instances with appropriate services (like transactions and security policies) and resources. An EJB container (representing the bean

8. http://www.icmgworld.com/corp/k2/k2.overview.asp
9. Java 2 Standard Edition
10. Java 2 Enterprise Edition

execution environment) configures services to match the needs of contained beans. These needs are expressed declaratively in a bean's deployment descriptor or, more recently, using annotations. Currently, there are four kinds of beans: stateless, stateful, entity and message-driven. They are all united by a common top-level contract between beans and containers and their use of deployment descriptors. Message-driven beans appeared in EJB 2.0 and are different from entity and session (stateless or stateful) beans. Entity and session beans share the design of EJB Object and EJB Home interface (non-functional interface). Message-driven beans behave as asynchronous event consumers [Haase 02]. They are stateless, not persistent but can use the transaction service. In EJB an application passes through all the phases of the life cycle defined in Section 3.2.1.4 that is: creation, assembly, deployment and execution. At creation time, interfaces, implementation classes and deployment descriptors related to a component are gathered into a package. At assembly time, components can be composed and corresponding packages for composite components created. The phase of deployment sees the beans being instantiated and initialized depending on descriptor specifications and configuration files. The execution phase consists in running the application. The container (the execution environment) controls the execution of the deployed beans by, among others, providing them with the non-functional services they require.

Java servlets pick up the spirit of applets but live on a server and are (usually) lightweight components instantiated by a web server processing, typically, web pages. Java ServerPages (JSP) can be used to declaratively define web pages to be generated. JSPs are then compiled to servlets.

J2EE introduces application client components. These are essentially unconstrained Java applications that reside on clients. A client component uses the JNDI (Java Naming Directory Interface) enterprise naming context to access environment properties, EJBs and resources on J2EE servers (for example, access to e-mail through JavaMail or databases via JDBC).

Among the panoply of "component" models in the Java world, the Enterprise JavaBeans seems to conform the most to the definitions of components, interfaces, composition and life cycle defined in Section 3.2.1 at page 59.

3.2.3.3 Component Object Model (COM)

The COM+ component model extends the COM[11] (Component Object Model) model of Microsoft with transactional, service browsing and request instrumentation services. COM was introduced with the operation systems family on 32 bits (Windows 95 and Windows NT 3.1). Later, DCOM (Distributed COM) [Eddon 98] was proposed to allow the construction of distributed applications.

The COM model allows the definition of black-box components implementing multiple interfaces and that can be dynamically connected to other components, possibly provided by different vendors [Brockschmidt 95]. Component interfaces specify both the required and the provided operations and implementations in different programming languages are possible.

11. www.microsoft.com/com

.NET [Löwy 01] is a newer component model proposed by Microsoft. .NET can be seen as an update of the COM+ model but without replacing COM+ components: each .NET component is also a COM+ component. .NET components evolves inside a common language execution environment dealing with execution aspects and providing services like memory management, distribution and security.

3.3 From Components to Software Architecture

3.3.1 Software Architecture Definition

Definition 2 *The Software Architecture of a program or computing system is the structure or structures of the system, which comprise software components, the externally visible properties of those components and the relationships among them [Bass 98].*

According to the ANSI/IEEE 1471 2000-standard, software architecture is: "The fundamental organization of a system embodied in its components, their relationships to each other and to the environment and the principles guiding its design and evolution" [Shaw 96b, Szyperski 02].

More clearly, the software architecture [Garlan 94, Bass 98, Garlan 94] is the organization of a software system as a collection of components, connections between the components, and constraints on how the components interact.

3.3.2 Software Connectors

Among the first-class entities (as suggested in [Shaw 96a]) of software architectures, *connectors* play an important role in component interactions. They establish the rules of interactions and indicate all the auxiliary mechanisms needed in theses interactions [Shaw 96b].

Definition 3 *A connector can be defined as a* reusable design element that supports a particular style of component interaction. *By extension, the corresponding implementation elements are also called connectors.*

3.3.2.1 Categories of Connectors

Connectors are very diverse and can be categorized according to the services they provide and the way the provide these services.

Categories of Services - The categories of services provided by connectors as described in [Aldrich 02c] are presented below:

***Communication*:** Communication connectors support transmission of data among components.

***Coordination*:** Coordination connectors support transfer of control among components[12]. Components interact by passing the thread of execution to each other. Function calls and method invocations are examples of coordination connectors[13]. Higher-level connectors, such as signals and load balancing connectors, provide richer, more complex interactions built around coordination services.

***Conversion*:** These connectors convert the interaction provided by one component to that required by another. Conversion services allow components that have not been specifically tailored for each other to establish and conduct interactions. Conversion of data formats and wrappers for legacy components are examples of connectors providing this interaction service.

***Facilitation*:** Facilitation connectors mediate and streamline component interaction. Even when heterogeneous components have been designed to interoperate with each other, there is a need to provide mechanisms for facilitating and optimizing their interactions. Mechanisms like load balancing, scheduling services, and concurrency control are required to meet certain extra-functional system requirements and to reduce coupling between components.

Every connector provides services that belong to at least one of these categories. It is also possible to have multi-category connectors to satisfy the need for a richer set of interaction services. For example, it is possible to have a connector that provides both communication and coordination services.

Connector Types - There is a need to classify connectors into different types based on the way in which they realize interaction services. These types of connectors, as described in [Aldrich 02c], are listed below:

***Procedure Call*:** Procedure call connectors model the flow of control among components through various invocation techniques (coordination). They also perform transfer of data among the interacting components through the use of parameters (communication).

***Event*[14]:** Event connectors are similar to procedure call connectors in that they model the flow of control among components (coordination). Once the event connector learns about the occurrence of an event, it generates messages for all interested parties and yields control to the components for processing these messages. Messages can be generated upon the occurrence of a single event or a specific pattern of events. The contents of an event can be structured to contain more information about the event, such as the time and place of occurrence, and other application-specific information (communication). Virtual connectors are formed between components interested in the same event topics. Event connectors

12. *Transfer* of control from component A to component B does not necessitate the *loss* of control by A.

13. Note that function calls and method invocations provide communication services in addition to coordination services.

14. An event can be defined as "the instantaneous effect of the (normal or abnormal) termination of the invocation of an operation on an object, and it occurs at that object's location" [Rosenblum 97].

are found in distributed applications that require asynchronous communication.

***Data Access*:** Data access connectors allow components to access data maintained by a data store component (communication). Data access often requires preparation of the data store before and clean-up after access has been completed. In case there is a difference in the format of the required data and the format in which data is stored and provided, data access connectors may perform translation of the information being accessed (conversion). The data can be stored either persistently or temporarily, in which case the data access mechanisms will vary.

***Linkage*:** Linkage connectors are used to tie the system components together and hold them in such a state during their operation. Linkage connectors enable the establishment of ducts, the channels for communication and coordination, which are then used by higher-order connectors to enforce interaction semantics (facilitation).

***Stream*:** Streams are used to perform transfers of large amounts of data between autonomous processes (communication). Streams can be combined with other connector types, such as data access connectors, to provide composite connectors for performing database and file storage access, and event connectors, to multiplex the delivery of a large number of events.

***Arbitrator*:** When components are aware of the presence of other components but cannot make assumptions about their needs and state, arbitrators streamline system operation and resolve any conflicts (facilitation), and redirect the flow of control (coordination). Arbitrators can provide facilities to negotiate service levels and mediate interactions requiring guarantees for isolation levels, reliability, and atomicity. They also provide scheduling and load balancing services. Arbitrators can ensure system trustworthiness by providing crucial support for dependability in the form of reliability, safety, and security.

***Adaptor*:** Adaptor connectors provide facilities to support interaction between components that have not been designed to interoperate. Adaptors involve matching communication policies and interaction protocols among components (conversion). These connectors are necessary for interoperation of components in heterogeneous environments, such as different programming languages or computing platforms.

***Distributor*:** Distributor connectors perform the identification of interaction paths and subsequent routing of communication and coordination information among components along these paths (facilitation). They never exist by themselves, but provide assistance to other connectors, such as streams or procedure calls.

3.3.3 Software Architecture Use

An architecture may be used for three main purposes (as specified in [Bosch 00]): for an individual software system, as a product-line architecture or as a standard architecture

used for a public component model.

The software architecture for an individual software system is part of the normal development cycle, preceded by requirement extraction and specification and followed by detail design, implementation, validation and deployment.

The use of a software architecture as a product-line architecture is, in the opinion of many authors, the most promising technique for achieving increased productivity, time-to-market and software quality. A software product line (SPL [15]) is a set of software-intensive systems that share a common, managed set of features satisfying the specific needs of a particular market segment or mission and that are developed from a common set of core assets in a prescribed way.

The third type of software architectural use is the standardization for a particular domain. It may be used by component developers and users as the means to agree on functionality covered by components, provided and required interfaces and dependencies between components. In [Szyperski 02], this type of architecture is referred to as a *component framework*.

When using Software Architectures, one systematically targets to:

- reduce the **development cost**,
- improve the **quality** in terms of reliability, maintainability and resource efficiency,
- reduce **time-to-market**,
- reduce **maintenance cost**.

These goals become more specific when using particular paradigms or approaches. See for example the benefits of using a Service-Oriented Architecture in Section 3.3.5.2.

3.3.4 Architecture Description Languages (ADLs)

The ADLs represent one of the results of the research done in the field of Software Architectures. Their objectives is to help application architects structure and compose their software parts in order to obtain valid applications. An ADL is defined as a formal or informal notation, either textual or graphical, making it possible to specify software architectures and that are accompanied with specific tools [Medvidovic 00a]. The result of using an ADL is mostly an abstract view over an application in terms of *components*, *connectors* and *configurations*, rather than a concrete view over the implementation details.

In this context, a component is a process or a data storage unit. A connector put components in relation and, in some ADLs, models their interaction. A configuration is a graph of components and connectors, defining the architecture structure. In order to exemplify ADLs, in the sequel, we present the Darwin [Magee 95] ADL and Tracta [Giannakopoulou 99] its extension allowing to add a behavioral description to each Darwin component.

15. http://www.sei.cmu.edu/productlines/

3.3.4.1 Darwin

Darwin was developed by the Distributed Software Engineering Group of the London Imperial College. It proposes an ADL for the construction of distributed applications. Its particularity is that it allows the specification of the dynamics of an application in terms of component creation all along its lifetime.

Darwin components.
A Darwin component is defined by an interface describing the required and provided services. The semantics associated to a component is the process, each component corresponds to a created process.

Darwin connectors.
The interaction between components is realized through provided and required services. The concept of connector is not explicitly present in Darwin. Instead, each interaction is represented by the binding of a required services in one component to a provided service in another component. Services have no functional connotation, they only designate the type of the communication object used to and authorized to use a component function. This type of objects are defined by the Darwin's distributed execution environment called *Regis*. Among these type of objects, the *port* is the most used.

Darwin configuration.
In order to offer the possibility of a hierarchical architecture, Darwin allows the description of two types of components:

- the *primitive components* are entities of encapsulation of functions and data. They are defined by their name and interface declaring the required and provided services of the component business logic.
- the *composite component* is defined as a configuration unit based on primitive and composite components. It describes the components interactions. Thus, the final structure of an application is represented by a composite component.

By default, before the actual binding of services, Darwin carries out a simple equivalence test concerning the name and types of the services. If the required service is compatible with the provided one, the configuration is considered valid.

Regis: an execution platform for Darwin.
Regis provides a C++ coded execution environment to construct and execute distributed programs specified by Darwin. The primitive components are C++ classes inheriting from the `Process` class. For the communication between components, Regis provides the programmers with several types of communication objects like ports or message multicasting.

Component deployment.
Darwin provides a deployment model related to its execution platform Regis. Each component is associated to a deployment site of the Darwin environment. The component creation site is determined in an absolute manner. This is realized at instantiation time by specifying a number designating the creation site. This number corresponds to an available site in the execution environment.

Expression of the dynamics.
An important characteristics of Darwin is the fact that it can describe the dynamic creation of components. This represents a notable advancement because the architecture is no longer considered as a collection of components and connections specified once for all at the conceptual phase. The mechanisms used in Darwin to describe dynamic structures are:

- *lazy instantiation* is a pre-declaration of instances being effectively created not in the phase of initialisation but when the first call to the instance is issued. Lazy instantiation allows the description of the potential configuration at execution time by the specification of dynamical structures at conception time. Unfortunately, this mechanism allows the instantiation of a unique component by clause of interconnection.

- *dynamic instantiation* allows the description of a component instance creation by providing it with initialization parameters. In a general manner, the dynamical creation of a component is realized inside a composite component. The access to this dynamically created component is managed by the composite component. All these declarations are realized by using the configuration language provided by Darwin.

Behavioral descriptions.
Tracta [Giannakopoulou 99], an extension of Darwin, makes it possible to add a behavioral description to each component. In Tracta, an architect adds a behavioral description to each primitive component. In addition, a set of properties that the system has to satisfy is also provided. Darwin uses FSP [16] [Magee 99], a high level language derived from CSP, to describe the components behavior. FSP provides a concise syntax for the definition of LTSs modeling the behavior of a primitive component. The behavior of the composite components is obtained from the descriptions of the primitive ones.

The association of the definition of services, of the dynamics of the architecture and of the components behavior makes of Darwin an important approach for the specification and analysis of distributed applications.

The behavioral descriptions allows for the detection of deadlocks and the verification of safety and liveness properties. From a liveness perspective, Darwin defines the notion of the *progression property*. In the case of a process scheduling priority policy, this property guarantees that for an infinite executions of a process we can be sure that at least one action defined by the property will be executed an infinite time. This type of property

16. Finite State Processes

makes it possible to prove, in some cases, that some actions would never have the chance to be executed.

Darwin evaluation.

Darwin proposes a clear and functional view of a software architecture. The major benefit of Darwin over the existing ADLs is the consideration of dynamic software architectures. By using Tracta, Darwin allows the modeling of system behavior by using the process algebra language FSP. Darwin, also allows the description of interfaces of level three according to the taxonomy proposed by Beugnard *et al.* [Beugnard 99].

Among the inconveniences of Darwin, the fact that a component can be associated only to a process semantics can be seen as a major drawback as a component cannot express another element like a file of the shared memory, for example. The environment utilization code is not transparent for the programmer. There is a unique implementation of Darwin, the application developer cannot make the choice of the execution platform and the explicit use of ports is required to perform communication. Finally, the description of the application dynamics remains limited. For example, it is not possible to dynamically destroy components or to allow a primitive component to communicate with dynamically created ones.

3.3.4.2 ADLs Evaluation

An ADL provides means to model and analyse both the static and dynamic properties of a Software Architecture. Medvidovic and Taylor [Medvidovic 00b] give an extensive classification and comparison framework of existing ADLs. Examples include Wright and Rapide ADLs.

The main advantage of Wright is to provide a formal language (CSP) to specify the components and connectors. Thus, the architecture can be analyzed. Wright is one of the first ADLs providing an interface description according to the third level proposed by the taxonomy of Beugnard *et al.* [Beugnard 99]. Wright is the result of research in the field of formal specifications. Important efforts were realized in order to allow the analysis of assemblies, but the problem of code generation was not sufficiently taken into consideration.

Unlike Wright, Rapide is a concurrent object language used to design distributed application architectures. An architecture is defined in Rapide as a set of *modules*, *module interfaces*, *connection rules* and many *formal constraints*. An interface defines provided and required services but also reactive executable rules. A reactive executable rule is defined like an event schema executed in an event model called *poset* [Luckham 95b]. A connector defines data synchronous or asynchronous communication between interfaces. A formal constraint specifies restrictions on the interfaces and connectors, restrictions related to the communication order and data.

While ADLs like Darwin, Wright and Rapide are used in the analysis and design steps of the classical development phases, the implementation step, is at best, only supported by code generation facilities. The result is that the implementation tends to loose its connection to the intended architectural structure during the maintenance steps. The same

result for a generated code when not strictly adhering to the model-driven discipline. The result is "architectural erosion" [Baumeister 06] mainly represented by the fact that implementation languages cannot guarantee that the implementation code obey architectural constraints. In order to deal with this problem, a new class of languages, namely Component Programming Languages (also referred as Architectual Programming Languages) tent to counter architectural erosion by the inclusion of architectural notions into general-purpose programming languages.

3.3.5 Service-Oriented Architectures (SOA)

Service-Oriented Architecture (SOA) is a relatively new approach to software architecture where functionality is grouped around business processes and packaged as interoperable services. SOA is above all a design and a way of thinking about building systems using heterogeneous network addressable software components and is often seen as an evolution of distributed computing based on the request/reply design paradigm for synchronous and asynchronous applications. A SOA architecture is made up of components (implementing services) and interconnections that stress interoperability and location transparency through the use of a *service layer*.

A service is a behavior that is provided by a component for use by any other component-based only on the interface contract. The key to services is their loosely coupled nature: the interface is independent of implementation and application developers or system integrators can build applications by composing services without even knowing the underlying implementation of the services. The contractualization is a critical activity in a SOA and the difference between a *public* interface and a *published* interface comes into play. While a public interface is an interface that can be used by components within a system, a published interface is one that is exposed to the network.

The services in the business logic layer have the ability to be invoked over a network. The technologies used to invoke the interface of the services stress interoperability. The services in the service layer also stress location transparency so they may be discovered and used dynamically by using a third party mechanism. Consequently, hard coding of a machine location is not consistent with a service-oriented approach.

As a summary, specific architectural principles for design and service definition focus on specific themes that influence the intrinsic behavior of a system and the style of its design. These principles include:

- service encapsulation - many webservices are consolidated to be used under the SOA architecture. Often such services have not been planned to be under SOA.
- service loose coupling - services maintain a relationship that minimizes dependencies and only requires that they maintain an awareness of each other.
- service contract - services adhere to a communications agreement, as defined collectively by one or more service description documents.
- service abstraction - beyond what is described in the service contract, services hide logic from the outside world.

- service reusability - logic is divided into services with the intention of promoting reuse.
- service composability - collections of services can be coordinated and assembled to form composite services.
- service autonomy - services have control over the logic they encapsulate.
- service optimization - all else equal, high-quality services are generally considered preferable to low-quality ones.
- service discoverability - services are designed to be outwardly descriptive so that they can be found and assessed via available discovery mechanisms.

3.3.5.1 SOA implementation

A SOA is commonly built using Web services [17] standards that have gained broad industry acceptance. These standards (also referred to as Web Service specifications) also provide greater interoperability and some protection from lock-in to proprietary vendor software. One can, however, implement SOA using any service-based technology, such as Jini [18], CORBA, DCOM, WCF [19] or REST [20] [Fielding 00].

High-level languages such as BPEL [21] and specifications such as WS-CDL [22] and WS-Coordination extend the service concept by providing a method for defining and supporting orchestration of fine grained services into more coarse-grained business services, which in turn can be incorporated into workflows and business processes implemented in composite applications or portals.

An emerging approach in implementing SOA is Service Component Architecture (SCA) [23]. The value proposition of SCA, is to offer the flexibility for true composite applications, flexibly incorporating reusable components in an SOA programming style. The overhead of business logic programmer concerns regarding platforms, infrastructure, plumbing, policies and protocols are removed, enabling a high degree of programmer productivity.

3.3.5.2 Benefits of using a SOA approach

While the SOA approach is fundamentally not new, it differs from existing distributed technologies in that most vendors accept it and have an application or platform suite that enables SOA. SOA, with a ubiquitous set of standards, brings better reusability of existing assets or investments in the enterprise and allows for the creation of applications that can be built on top of new and existing applications. SOA enables changes to applications while keeping clients or service consumers isolated from evolutionary changes that happen in the

17. http://www.w3.org/2002/ws/
18. http://www.sun.com/software/jini/
19. Windows Communication Foundation
20. Representational State Transfer
21. Web Services Business Process Execution Language
22. Web Services Choreography Description Language
23. http://www.osoa.org/display/Main/Service+Component+Architecture+Home

service implementation. SOA enables upgrading individual services or services consumers; it is not necessary to completely rewrite an application or keep an existing system that no longer addresses the new business requirements. Finally, SOA provides enterprises better flexibility in building applications and business processes in an agile manner by leveraging existing application infrastructure to compose new services. SOA benefits as seen from an industrial point of view can be summarized in the followings:

- better return on investment - as the architecture design makes possible the creation of services that can be reused in many contexts,
- code mobility - as the location transparency is a key property of the service layer,
- better parallelism in development and focused developer roles - as the architecture design forces an application to have multiple layers and thus the need to clearly specify roles in the development,
- better testing/fewer defects - as the modular implementation of the services allow for extensive unit testing without involving the rest of the application,
- support for multiple client types - as the data format exchanged between the client and the server is standard,
- more reuse and service assembly - as services are meant to be composed in many different contexts,
- better maintainability and better scalability,
- higher availability - as due to the location transparency, multiple servers may have multiple instances of a service running on them.

3.3.5.3 Challenges in using SOA approach

While the importance and benefits of using a SOA approach are obvious, some challenges to pass from theory to practice also exist.

One obvious and common challenge faced is managing services descriptions (metadata). SOA-based environments can include many services which exchange messages to perform tasks. Depending on the design, a single application may generate an important number of messages. Managing and providing information on how services interact is a complicated task.

Another challenge is providing appropriate levels of security. Security models built into an application may no longer be appropriate when the capabilities of the application are exposed as services that can be used by other applications. That is, application-managed security is not the right model for securing services. A number of new technologies and standards are emerging to provide more appropriate models for security in SOA.

As SOA and the WS specifications are constantly being expanded, updated and refined, there is a shortage of skilled people to work on SOA based systems, including the integration of services and construction of services infrastructure.

There is significant vendor hype concerning SOA that can create expectations that may not be fulfilled. SOA does not automatically guarantee reduced IT costs, improved systems agility or faster time to market. Successful SOA implementations may realize some or all of these benefits depending on the quality and relevance of the system architecture and design.

3.3.6 Component-Oriented Programming (COP)

While architectural analysis in existing ADLs may reveal important architectural properties, these properties are not guaranteed to hold in the implementation. In order to enable architectural reasoning about an implementation, the implementation must conform to its architecture. Luckham and Vera [Luckham 95a] identify three criteria for architectural conformance:

- *decomposition* (each component in the architecture has a corresponding component in the implementation),
- *interface conformance* (each implementation component must conform to its architectural interface)
- *communication integrity* (each implementation component may only communicate directly with the components to which it is connected in the architecture)

In order to deal with these three requirements, a new approach was born, that is programming with components. A new class of programming languages integrating architectural abstractions into a general-purposed language like Java. Below we present some of the actual so called *component programming languages* (others use the term of *architectural programming languages* [Baumeister 06]) used to support the paradigm of Component-Oriented Programming.

3.3.6.1 ArchJava

ArchJava [Aldrich 02a, Aldrich 02b] is a small, backwards-compatible extension of Java that integrates software architecture with Java implementation code. ArchJava supports a flexible object-oriented programming style, allowing data sharing and supporting dynamic architectures where components are created and connected at run time. An important feature of ArchJava is a type system that guarantees communication integrity between an architecture and its implementation, even in the presence of shared objects and runtime architecture configuration [Aldrich 02a].

Architectural reasoning is supported in ArchJava by using entities like components, ports and connections inside Java code. A component is a special kind of object that communicates with other components in a structured way. Components are instances of component classes (described with a slightly modified Java language). Components can only communicate with other components at their level in the architecture through explicitly declared ports. Regular method calls between components are not allowed. A port

represents a logical communication channel between a component and one or more components that it is connected to. Ports declare required and provided methods, thus conforming to the connection-oriented programming style [Szyperski 02]. Hierarchical composition is expressed with composite components, which are made up of interconnected subcomponents. Connections interconnect two or more ports by binding each required method to a provided method with the same name and signature. Inheritance is also supported in ArchJava, that is component classes can inherit from other component classes. One restriction is that component subclasses may not specify new required methods because this could break subtype substitutability.

Dynamic component creation is supported by the `new` syntax used to create ordinary objects. However, communication integrity places restrictions on the ways component instances can be used [Aldrich 02a]. In addition to the `new` operator, connect patterns/expressions are used to connect together component instances at runtime. Multiplicity is supported by declaring port interfaces. Port interfaces can be seen as port types that are to be instantiated at run-time for each particular connection that will be realized between component instances.

Component and connection instances are garbage collected as in Java, when they are no longer reachable through direct references, running threads or architectural connections.

Limitations of the ArchJava language include the fact that it cannot be used in a distributed environment, as ArchJava programs must be run in the same JVM. ArchJava's definition of communication integrity supports reasoning about communication through method calls between components; however components may still use shared data to communicate in ways that are not directly expressed in the architecture. Finally, ArchJava lacks port protocols or other abstract means to specify component behavior. Thus, reasoning about communication integrity is limited to analysis on the interface level.

3.3.6.2 Java/A

Java/A [Baumeister 06] extends Java by providing support for architectural concepts introducing port, required and provided interfaces, simple and composite component and assembly keywords. Unlike ArchJava, Java/A also includes port protocol descriptions as UML state machines and allows a developer to deal with behavioral descriptions at port level.

The component interfaces are bound to ports that regulate message exchange by protocols and ports can be linked by connectors establishing a communication channel between their owning components. Thus, safe communication can be specified and verified. The number of component in an assembly (composite component), the number of ports a component offers, the linkage of ports between components can vary dynamically, providing basic means for dynamic reconfiguration. Java/A is supported by a compiler which translates Java/A programs to Java classes and includes the possibility to check port protocols for compatibility (i.e. that two connected ports will only exchange messages the communication partner understands, and the trace of messages will not lead to a deadlock). This kind of verification is done at compile time by employing the HUGO model checker encapsulated within the JAVA/A compiler.

The Java/A language resides on an abstract component model formalized using interface automate design (for behavior) and *states-as-algebras* approach (representing the internals of components and assemblies).

3.3.6.3 Jiazzi

Jiazzi [McDirmid 01] does not extend Java. Instead, it provides separate compilation in order to obtain components out of Java code (externally linked code modules called units). Jiazzi components (*atoms* or *compounds*) can be thought of as generalizations of Java packages with added support for external linking and separate compilation. Jiazzi components are practical because they are constructed out of standard Java source code. Jiazzi requires neither extensions to the Java language nor special conventions for writing Java source code that will go inside a component. Jiazzi components are expressive because Jiazzi supports cyclic component linking and mixins, which are used together in an *open class pattern* that enables the modular addition of new features to existing classes. Unlike Java/A, Jiazzi does not provide any support for behavioral descriptions.

Thanks to Jiazzi architecture, units can act as effective "aspect" constructs with the ability to separate crosscutting concern code in a non-invasive and safe way [McDirmid]. Unit linking provides a convenient way for programmers to explicitly control the inclusion and configuration of code that implements a concern, while separate compilation of units enhances the independent development and deployment of the concern. The expressiveness of concern separation is enhanced by units in two ways. First, classes can be made open to the addition of new behavior, fields, and methods after they are initially defined, which enables the direct modularization of concerns whose code crosscut object boundaries. Second, the signatures of methods and classes can also be made open to refinement, which permits more aggressive modularization by isolating the naming and calling requirements of a concern implementation.

The actual implementation of Jiazzi consists of a stub generator and linker. Stubs are generated for imported classes to ensure they are used correctly in classes that the atom contains. The linker ensures that the atom classes are consistent with the atom unit signature. For compounds, the linker ensure that the linking of units within the compound is consistent with the compound unit signature. Thus, the linker performs type checking ensuring that architectural constraints hold in implementation.

3.3.6.4 ComponentJ

A similar approach to those of ArchJava and Java/A is proposed by ComponentJ [Seco 02]. ComponentJ is a programming language for the Java platform which favors code reuse by composition instead of implementation inheritance. Components and objects are the main ingredients of a ComponentJ program. ComponentJ is based on a component core calculus presented in [Seco 00] that formally presents the basic concepts of the language and develops a type system that ensures the safety of those constructions.

ComponentJ integrates well with Java. The compiler takes ComponentJ type declarations and produces two kinds of gadgets that help integrate ComponentJ and Java code.

From a component type one can produce skeleton classes that allow one to naively program a component. On the other hand, one can produce stub classes that allow Java programs to use ComponentJ components.

The compiler works by translating ComponentJ code to Java. ComponentJ source files are type checked and transformed into a set of Java classes and interfaces. The resulting code is then processed by a standard Java compiler to produce executable bytecode. The packaging and deployment of compiled components is still undefined, hence a ComponentJ component is here represented by a set of class files.

A different approach is based on incremental development of applications by using layers. JavaLayers [Cardone 00] extends Java by implementing a software component model in which applications are constructed incrementally in layers. Applications are built by composing components and are changed by specifying new compositions.

3.3.7 The Coordination Paradigm

A new class of models, formalisms and mechanisms has evolved for describing concurrent and distributed computations based on the concept of "coordination" (a concept by no means limited to computer science). Malone and Crowston [Malone 94] characterize coordination as an emerging research area with an interdisciplinary focus, playing a key issue in many disciplines such as economics and operational research, organization theory and biology. Consequently, there are many definitions of what coordination is. In the area of programming languages, *coordination is the process of building programs by gluing together active pieces* [Gelernter 92]. From this perspective, a *coordination model* is the glue that binds separate activities into an ensemble. Furthermore, a *coordination language* is the linguistic embodiment of a coordination model, offering facilities for controlling synchronization, communication, creation and termination of computational activities.

The coordination paradigm implies that programming a distributed or parallel system can be seen as the combination of two distinct activities: the actual *computing* part comprising a number of processes involved in manipulating data and a *coordination* part responsible for the communication and cooperation between the processes.

Coordination is closely related to the concepts of multilinguality and heterogeneity. Since the coordination component is separate from the computational one, the former views the processes comprising the latter as blackboxes; hence, the actual programming languages used to write computational code play no important role in setting up the coordination apparatus. Furthermore, since the coordination component offers a homogeneous way for interprocess communication and abstracts away the machine-dependent details, coordination encourages the use of heterogeneous ensembles of architectures.

The purpose of a coordination model and associated language is to provide a means of integrating a number of possibly heterogeneous components together, by interfacing with each component in such a way that the collective set forms a single application that can execute on and take advantage of parallel and distributed systems. Almost all of these models share the same intent, to provide a framework which enhances modularity, reuse of existing (sequential or parallel) components, portability and language interoperability. However, they differ in how they precisely define the notion of coordination, what exactly

is being coordinated, how coordination is achieved and what are the relevant metaphors that must be used.

Papadopoulos and Arbab [Papadopoulos 98] argue that the existing coordination models fall into two major categories, namely either *data-driven* or *control-driven*. In a data-driven approach, the evolution of computation is driven by the types and properties of data involved in the coordination activities. In control-driven (or process-oriented) models, changes in the coordination processes are triggered by events signifying (among other things) changes in the states of their coordinated processes.

Configuration and architectural description are closely related to coordination. Configuration and architecture description languages share the same principles with coordination languages. From a slightly liberal point of view, one can include configuration and ADLs in the category of coordination languages.

In the sequel, we detail the characteristics of coordination models and languages, where by *coordination* we also mean *configuration* and *architectural description*. For a more exhaustive comparison of existing coordination models and languages, the technical rapport presented by Papadopoulos and Arbab [Papadopoulos 98] is considered as a reference.

3.3.7.1 Data-Driven Coordination

In a data-driven approach, the state of the computation at any moment in time is defined in terms of both the values of the data being received or sent and the actual configuration of the coordinated components. This means that, at least stylistically or linguistically, there exists a mixture of coordination and computation code within a process definition. The immediate consequence is that processes cannot be easily distinguished as either coordination of computational processes. It is usually up to the programmer to design her/his program in such a way that the coordination and the computational concerns are clearly separated and are made the responsibility of different processes. However, most of the time such a clear separation is not enforced at the syntactic level by the coordination model.

The data-driven category tends to be used mostly for parallelizing computational problems. Almost all data-driven coordination models have evolved around the notion of *shared dataspace* [Roman 90], that is a common, content-addressable data structure. All processes involved in some computation can communicate (by posting, broadcasting or retrieving information) among themselves only indirectly via this medium. Historically, Linda [Ahuja 86, Carreiro 89] is the first genuine member of the family of coordination languages. It is based on the so-called *generative communication* paradigm: if two processes wish to exchange some data, then the sender generates a new data object (referred as a *tuple*) and places it in some shared dataspace (a *tuple space*) from which the receiver can retrieve it. Linda is in fact not a fully-fledged coordination language but a set of some simple coordination primitives that are completely independent of the host language. Thus, it is possible to derive natural Linda variants of almost any programming language (like C, Modula, Pascal, Ada, Prolog, Lisp, Eiffel or Java) or paradigm (imperative, logic, functional, object-oriented).

Based on the concepts originally proposed by Linda, different coordination models and languages have been designed [Papadopoulos 98].

3.3.7.2 Control-Driven Coordination

In a control-driven approach, the state of computation at any moment in time is defined in terms of only the coordinated patterns that the processes involved in some computation adhere to. The actual values of the data being manipulated by the processes are almost never involved. This means that the coordination component is almost completely separated from the computational component. The control-driven category tends to be used primarily for modelling systems. In such systems, processes communicate with their environment by means of clearly defined interfaces, usually referred to as *input* or *output ports*. Producer-consumer relationships are formed by means of setting up *stream* or *channel* connections between output ports of producers and input ports of consumers. In addition to using ports, processes often send out to their environment *control messages* or *events* with the purpose of letting other interested processes know in which *state* they are of informing of any *state change*.

Considering these properties of control-driven (or process-oriented) coordination models, it is easy to observe that some of the ADLs fall into this category. For example, Darwin/Regis [Magee 92] and Rapide[Shaw 95] are considered to be evolutions of the Conic [Kramer 90] model. Conic is a programming language which is a variant of Pascal enhanced with message-passing primitives, plus a configuration language featuring logical nodes configured together by means of links established among their input/output ports. PCL (*Proteus Configuration Language*) [Sommerville 94] is a language designed to model architectures of multiple versions of computer-based systems. Coordination is viewed as a configuration where the unit of configuration is a family entity, representing one or more versions of a logical component or system. Some other examples of control-driven coordination approaches are: Durra [Barbacci 90], CSDL [DePaoli 94, DePaoli 93], POLYLITH [Purtilo 94], ConCoord [Holzbacher 96] and MANIFOLD [Arbab 93].

3.4 Conclusions

In this chapter we have presented the current state-of-the-art related to components, components models, software architectures, ADLs, Service-Oriented Architecture, component programming languages (also referred as architectural programming languages), and coordination. All of these are part of the ongoing work in CBSE to provide efficient methods, languages and tools for the development of component-based applications.

Particular component characteristics are described by component models. A component model describes what a component is (by defining its constituent parts) and says how components can be eventually composed (by following some composition rules). It also describes the component life cycle and the roles associated with different actors in the development and exploitation of applications.

One important notion related to a component is its interface. The interface represents the component boundary and offers a view on the component by abstracting from implementation details. Collaborating entities, as software components are, often exchange messages in order to coordinate actions or simply exchange data. Components communicate through their interfaces and thus interfaces play an important role in the composition

mechanism.

The Software architecture [Garlan 94, Bass 98, Garlan 94] is the organization of a software system as a collection of components, connections between the components, and constraints on how the components interact. When using Software architectures, one systematically targets to: reduce the development cost, improve the quality in terms of reliability, maintainability and resource efficiency, reduce time-to-market and reduce maintenance costs.

The ADLs represent one of the results of the research done in the field of Software architectures. Their objectives is to aid application architects structure and compose their software parts in order to obtain valid applications. An ADL is defined as a formal or informal notation, either textual or graphical, allowing to specify software architectures and that are accompanied with specific tools [Medvidovic 00a]. The result of using an ADL is mostly an abstract view over an application in terms of *components*, *connectors* and *configurations*, rather than a concrete view over the implementation details.

Service-Oriented Architecture (SOA) is a relatively new approach to software architecture where functionality is grouped around business processes and packaged as interoperable services. SOA is above all a design and a way of thinking about building systems using heterogeneous network addressable software components and is often seen as an evolution of distributed computing based on the request/reply design paradigm for synchronous and asynchronous applications. A SOA architecture is made up of components (implementing services) and interconnections that stress interoperability and location transparency through the *service layer* use.

While architectural analysis in existing ADLs may reveal important architectural properties, these properties are not guaranteed to hold in the implementation. In order to enable architectural reasoning about an implementation, the implementation must conform to its architecture. Luckham and Vera [Luckham 95a] identify three criteria for architectural conformance: decomposition, interface conformance and communication integrity. In order to deal with these three requirements, a new approach was born, that is programming with components. A new class of programming languages integrating architectural abstractions into a general-purpose language like Java. Examples include ArchJava [Aldrich 02a, Aldrich 02b], Java/A [Baumeister 06] and Jiazzi [McDirmid 01].

A new class of models, formalisms and mechanisms has evolved for describing concurrent and distributed computations based on the concept of *coordination* (a concept by no means limited to computer science). The purpose of a coordination model and its associated language is to provide a means of integrating a number of possibly heterogeneous components together, by interfacing with each component in such a way that the collective set forms a single application that can execute on and take advantage of parallel and distributed systems. Configuration and architectural description are closely related to coordination. Configuration and architecture description languages share the same principles with coordination languages. From a slightly liberal point of view, one can include configuration and ADLs in the category of coordination languages.

In the next chapter we present formalisms usually employed in order to describe and verify interaction (behavioral) protocols. We also present different component models and languages currently integrating interaction protocols expressed by these formalisms.

Chapter 4

Interaction Protocols

In this chapter we start by presenting state-of-the-art formalisms usually employed to describe and verify interaction protocols. Different categories of formalisms like process algebras, behavioral types, finites state machines or temporal logic are presented. Next, we discuss the integration of these formalisms in existing component models and finally conclude.

Contents

4.1 Introduction

Collaborating entities often exchange messages in order to coordinate actions or simply exchange data. In order to successfully interact, entities (either objects, agents or components) need to conform to a certain form of interaction contract. Interaction protocols,

generally, describe the entity behavior in terms of allowed message sequences that must be exchanged between entities in order to perform the system intended global behavior.

Multiple approaches coexist in order to formalize, analyze and implement interactions between entities composing a larger system. In the following, we start by presenting the mainstream research directions that focus on providing the software community with methods and tools that allow the description, analyze and implementation of interaction protocols. However, an in-depth, formal comparison of all the presented formalisms is out of the scope of this thesis.

In the second part of this chapter, we present some component models integrating different forms of interaction protocols.

4.2 Formalisms

4.2.1 Process Algebra

The term *process algebra* refers to a family of specification techniques particularly well suited to describing systems of concurrent communicating components. Process algebras describe the process interactions in terms of calculus (an ensemble of rules defined around a small set of process expression construction operators). There are many process algebra formalisms issued from the research field. The most important ones are CCS [Milner 89] and CSP [Hoare 85] as they stand at the base of other algebras like the PI-calculus [Milner 92, Milner 99] and the mobile agents [Cardelli 98], integrating primitives for the expression of distribution and mobility.

At first glance, CCS, CSP and LOTOS [Brinksma 87], a process algebra derived from the first two ones, are very similar in their basic concepts. They all begin with the notions of processes composed from atomic actions, have operational semantics and include many operators in common. However, they also differ in their philosophy and area of application. Some syntactically identical operators have very different semantics and some semantic requirements are expressed differently from one language to another.

In the remaining of this section, we present some details on CSP, CCS and LOTOS as they stand as the basis for other process algebra languages and as they have proven valuable in the specification and design of distributed systems [Logrippo 92], for formal reasoning [Milner 89] and for rapid prototyping [Loureiro 91]. We also give a short comparison in Section 4.2.1.4 at page 96.

4.2.1.1 CSP

CSP [Hoare 85] allows the description of an application like an ensemble of parallel processes communicating through communication channels and using events. The communication among processes is unidirectional and synchronous. CSP proposes a set of operators in order to define processes: the operators for deterministic and non deterministic choices are an example. The composition operator allows the composition of independent (no communication between processes) or dependent processes.

CSP is based on the mathematical theory of *traces* and *failures*. The process behavior consists of all the traces that can be realized at execution. A trace is a finite sequence of events realized up to a specific moment in time. A failure defines the set of events that cannot be executed after a certain trace. The specificity of CSP is that it distinguishes between events representing the sending and receiving of messages.

CSP is a very expressive language allowing, among others, the modeling of operation systems. Also, CSP provides interesting abstractions for the description of communicating processes behavior but they lack in abstractions allowing the definition of data structures.

4.2.1.2 CCS

CCS [Milner 89] was developed in order to model concurrent processes communicating through signals. A process is defined like an ensemble of agents interacting through two unique actions: the sending and receiving of signals through defined ports. An agent is constructed from operators like the prefix, the choice, the composition and the repetition/recursion. The choice operator, for example, allows the environment to choose between different action alternatives provided by an agent. The Agent composition allows two processes to communicate.

All the actions an agent can realize generate the so called *derivation tree*. The behavior of an agent is defined like a property of its derivation tree, obtained from the application of derivation rules defining the CCS semantics. The observation of processes is supported by equivalence relations allowing the comparison of processes behavior. Among the equivalence relations there are the strong equivalence and the weak equivalence (also called *bissimulations*).

CCS is a general-purpose language providing abstractions that allow the definition of interaction protocols between processes. Meanwhile, the absence of support for the definition of data structures and the low-level encapsulation prevent the direct application of this formalism to a component model.

4.2.1.3 LOTOS

LOTOS [Brinksma 87] consists of an algebra specification language for defining data and a language of communicating processes developed for the formal description of the OSI [1] architecture. While inheriting its main characteristics from CCS and CSP, its semantics is very close to that of CCS. In LOTOS, an application is viewed as a set of processes interacting and exchanging data among themselves and their environment. Tools like CADP [2] [Fernandez 92] can be used to analyze and execute architectures defined in LOTOS.

Like CSP and CCS, LOTOS is a general-purpose language offering abstractions that are too low-level to be used in a component model. Some work using LOTOS in order to define an architecture description language is presented in [Heisel 97].

1. Open Systems Interconnect
2. Caesar/Aldebaran Distribution Package

4.2.1.4 Process Algebra Evaluation

The process algebras CSP, CCS and LOTOS are all very similar in their basic concepts. Each formalism is described in terms of a set of operators over the same basic domain: processes and actions. However, they differ in their philosophy and area of application.

With regards to syntax and semantics of operators, it can be seen that CSP and LOTOS are quite similar and useful as practical specification languages, while CCS is more suited to small theoretical investigations. Indeed, CCS was defined with the intention to have a minimal set of operators which allow the semantics of the language to be more easily explored. On the other hand, CSP and LOTOS in particular were designed for large communication systems and therefore have a lot of operators which make it easy to build a large specification out of smaller parts.

The three formalisms, although based on a common semantic model (CSP can also be defined in terms of labeled transition systems (LTS)), have slight differences in the way in which that model is interpreted. While, for example, communication in CSP and LOTOS is based on multi-way synchronization, CCS restricts communication to two parties.

Another difference is the use of distinguished actions. CSP has one distinguished action which signifies successful termination. Although the hiding operator in CCS, as in LOTOS, produces internal, invisible, actions, CSP has no special notation for this. CCS, on the other hand, relies quite heavily on the internal action for the result of communication, and for modeling nondeterminism.

Finally, when considering implementation of processes, CSP is closely related to the language Occam[3], CCS is incorporated in the language LCS [Berthomieu 89], and tools exist which translate LOTOS into C.

4.2.2 Behavioral Types

Behavior models are precise, abstract descriptions of the intended behavior of a system. Behavior models have solid mathematical foundations that can be used to support rigorous analysis and mathematical verification of properties. Effective techniques and tools have been developed for this purpose and have shown that behavior modeling and analysis are successful in uncovering the subtle errors that can appear when designing concurrent and distributed systems.

According to type theory, a type system defines how a programming language classifies values and expressions into types, how it can manipulate those types and how they interact. A type indicates a set of values that have the same sort of generic meaning or intended purpose (although some types, such as abstract types and function types, might not be represented as values in the running computer program). Type systems vary significantly between languages with, perhaps, the most important variations being their compile-time syntactic and run-time operational implementations.

Much of the work on developing type-theoretic foundations for programming languages has its roots in typed lambda calculus. In such approaches, an instance of a type is viewed as a record of functions together with a hidden representation type [Cardelli 85]. We talk

3. http://www.wotug.org/occam/

about *service types* when referring to such an instance as it only describes the signatures of the services an entity (either object or component) provides to or requires from its communication partners.

More recent work ([Nierstrasz 93, Puntigam 96]) advocate for the extension of classical service types in order to allow the description of dynamic properties of objects (and also components) and their composition. This kind of types are called *behavioral types* as they specify not only a set of messages to be exchanged between entities in order to communicate but also constraints on acceptable sequences of these messages.

4.2.2.1 Regular types and non-regular types

O. Nierstratz proposes behavioral types for active objects [Nierstrasz 93]. While *service types* describe types of service execution request and reply, the author also defines *regular types* that express the abstract states in which services are available and when transitions between abstract states may take place. Regular types are used to express non uniform service availability especially in concurrent object-oriented languages, where *active objects* may have their own thread of control and may delay the servicing of certain requests according to synchronization constraints. We talk about an object protocol. Protocols are described as *regular processes*, that is processes with a finite number of states or behaviors [Bergstra 84, E.M. Clarke 83, Graf 86, Milner 84]. By process, it is meant an abstract machine communicating by passing messages along named channels as in CCS.

Regular types may be refined according to a subtype relation called *request substitutability* specifying that services may be refined as long as the original promises are still upheld [Wegner 88]. That is, an object that *conforms* to the protocol of another object is substitutable for the second object, in the sense that clients expecting that protocol to be supported will receive no unpleasant surprises [Nierstrasz 93].

Puntigam [Puntigam 96] proposes *process types* in order to deal with the behavioral properties of objects. In his approach, process types specify sequences of acceptable messages. Even if the set of acceptable messages changes dynamically, a type checker can statically ensure that only acceptable messages are exchanged. In [Puntigam 99], Putigam proposes to increase the expressiveness of process types in order to specify non-regular message sequences (i.e. non deterministic message sequences). This proposal assures that type equivalence and subtyping are decidable, sound and complete even for non-regular process types, provided that these relations conform to an extensibility criterion.

4.2.2.2 Similar approaches on behavioral types

The proposal of Najim and Nimour [Najm 99] has a similar purpose as process types of Putigam. A limitation of their proposal compared to that of Putigam, however, is that at any time only one client is allowed to interact with a server through an interface specifying acceptable message changes.

A similar definition of subtyping as that proposed by Nierstratz was given by Bowman *et al.* [Bowman 97]. The proposal of Nielson and Nielson [Nielson 93] can deal with constraints on message ordering. Their system cannot ensure that all messages are understood, but

subtyping is supported.

Liskov *et al.* [Liskov 94] also propose a behavioral notion of subtyping. They define a new notion of subtype relation based on the semantic properties of the subtype and supertype. An object type determines both a set of legal values and an interface with its environment (through calls on its methods). The interest is in preserving properties about supertype values and methods when designing a subtype. They require that a subtype preserve the behavior of the supertype methods and also all invariants and history properties of its super type.

Largely outside the object-oriented or component-based design community, type systems that capture various dynamic properties of programs have been studied under various settings, such as concurrent ML, actor-based languages, π-calculus, the CNAM formalism, etc.

4.2.3 Finite State Machines

A finite state machine (FSM) is an abstract machine that is used to study and design systems that recognize and identify patterns. The idea of a finite state machine comes from an interdisciplinary branch of mathematics called formal language theory. Its roots are also in computer science and linguistics.

Whenever pattern recognition is vital to scientific inquiry, finite state machines may play an important role. Although a finite state machine cannot identify and recognize all types of patterns, it is still very powerful. It is the simplest and most basic pattern-recognizer and pattern-describer used in computer science.

State machine based formalisms are generally assumed to be complete descriptions of system behavior at some level of abstraction. From a component modeling perspective, the system behavior is what an external entity can observe about the system's interaction with its environment. This is usually the messages the system (black-box component) exchanges with its environment in terms of emissions and receipts. A finite state machine is describing the set of all possible traces a component can produce when interacting with its partners.

4.2.3.1 LTS

Labelled Transition Systems (LTS) [Keller 76, Arnold 94] represent one of the FSM based formalisms employed in order to describe process behavior. An LTS consists of a finite set of states and a corresponding set of transitions between states.

Formally, an LTS is a tuple $\langle S, A, \Delta \rangle$ where S is a set (of states), A is a set (of labels) and $\Delta \subseteq S \times A \times S$ is a ternary relation.
If $p, q \in S$ and $\alpha \in A$, then $\langle p, \alpha, q \rangle \in \Delta$ is written as $p \xrightarrow{\alpha} q$.
This represents the fact that there is a transition from state p to q with label α. A label can represent different things depending on the language of interest. Typical use of labels include representing input, conditions that must be true to trigger the transition, or actions performed during the transition.
LTS states generally represent "logical states" in the entity execution. Logical states

represent the set of values of all the variables of an entity at a specific moment.

The composition of two or more LTSs results in the *synchronous product* of the ensemble. The synchronous product represents the global behavior of all the interacting entities. Formally, the synchronous product of two LTSs ($\|$) is defined as:

$P\|Q = \langle S_1 \times S_2, A_1 \cup A_2, \Delta \rangle$ where Δ is the smallest relation satisfying the rules:

$$\frac{P \xrightarrow{\alpha} P'}{P\|Q \xrightarrow{\alpha} P'\|Q}\ \alpha \notin A_2,\ \frac{Q \xrightarrow{\alpha} Q'}{P\|Q \xrightarrow{\alpha} P\|Q'}\ \alpha \notin A_1,\ \frac{P \xrightarrow{\alpha} P',\ Q \xrightarrow{\alpha} Q'}{P\|Q \xrightarrow{\tau} P'\|\ Q'}$$

specifying that transitions (on different LTSs) with the same labels are executed synchronously (in the same) from a compositional point of view where $P = \langle S_1, A_1, \Delta_1 \rangle$ and $Q = \langle S_2, A_2, \Delta_2 \rangle$ are two distinct LTSs.

The Labelled Transition System Analyzer [Magee 99] provides model checking and animation functionality over behavior models written in the Finite State Processes (FSP) process algebra [Magee 99].

Number of LTS extensions were proposed. For example, PLTSs (Partial LTSs) [Uchitel 03] extend LTSs by explicitly modeling in each state the set of actions that must not occur, i.e. the set of proscribed actions at each state. Indeed, when considering pure LTSs, there is no possibility to make a distinction between proscribed behavior and behavior that has not yet been defined especially during the development phase. By explicitly modeling the aspects of system behavior that are unknown, PLTS makes it possible to generate meaningful feedback to users leading to more comprehensive descriptions of the system behavior [Uchitel 03].

4.2.3.2 I/O Automata

I/O Automata provide an appropriate model for discrete event systems consisting of concurrently-operating components. The *input/output automaton* model has been defined in [Lynch 87] as a tool for modeling this kind of systems. Such systems are often characterized by the fact that they continuously receive input from and react to their environment. One characteristic of this model is that I/O automata are unable to block inputs or eliminate undesirable inputs. In other words, if the environment behaves correctly, then the automaton behaves correctly. Instead of allowing the automaton to block bad inputs, the model permit the automaton to exhibit arbitrary behavior when they do (usually by error messages). I/O automata may be nondeterministic and this is an important characteristic of the model descriptive power. One important notion in the I/O automata model is *fair executions*. When I/O automata are run, they generate executions (alternating sequences of states and actions). Fair executions are those that permit each of the automaton primitive components to have infinitely many chances to perform output or internal actions.

The I/O automata model is especially helpful when describing the interfaces between system components and it provides a clean composition model for fair composition. Although I/O automata can be used to model synchronous systems, they are best suited for modeling systems in which the components operate asynchronously.

A number of extensions to this initial model exist. In [Merritt 91], the authors propose to augment the I/O automata model with a notion of time in order to reason about time

in concurrent systems. The *Interface Automata* proposal in [de Alfaro 01], starts from the initial I/O automata model but takes a rather *optimistic* approach when considering components in their environment. While the *pessimistic* approach considers two components compatible if they can be used together in all systems, under the optimistic approach two components are compatible if they can be used together in at least one design. By not accepting certain inputs, the interface automaton expresses the assumption that the environment never generates these inputs. In this way, environment assumptions can be used to encode restrictions on the order of method calls, and on the types of return values and exceptions.

4.2.3.3 STS - Symbolic Transition Systems

LTSs suffer from a very important drawback when it comes to model infinite systems, big systems with data, complex conditions or input/output informations. In order to deal with this issue, STSs extend LTSs by using *symbolic transitions*. Unlike LTSs, in STSs transitions describe *classes* of possible operations to be effectively executed. One transition defines an operation that can be parameterized with input and output parameters. In addition a transition can be guarded, allowing the execution of the operation only if the condition of the guard is true. Where an LTS explicitly describes all the traces that can be realized at execution, an STS abstracts on the possible traces. Thus, an STS description is much more readable, compact and expressive than a classical LTS description.

4.2.3.4 UML State Machine Diagrams

State machine diagrams, formerly named state charts were introduced by [Rumbaugh 90] and adopted in UML [UML 03]. A state machine diagram is a visual formalism used to describe concurrent and reactive systems. This formalism is an extension of the approaches based on state diagrams [Harel 87]. A state machine is defined by a sequential automata where the transitions are guarded by conditions that must be satisfied before the action can be realized. It can be composed by using simple, parallel or hierarchical operators to describe more complex systems. In order to interact with each other, state machine automata must designate the messages that must be exchanged in order to modify their behavior. Some work like that presented in [Krüger 99] propose techniques to translate architecture description languages to state diagrams.

4.2.4 Temporal Logics

In logic, the term temporal logic is used to describe any system of rules and symbolism for representing, and reasoning about, propositions qualified in terms of time. Initially introduced by the the notable computer scientist Amir Pnueli [Pnueli 77], temporal logics are used to specify and verify properties like safety, liveness, and fairness. Among the numerous temporal logics, the most known are: Linear Temporal Logic (PLTL) [Clarke 81] and the Arborescent Temporal Logic (CTL) [Pnueli 81]. The biggest advantage of using temporal logics is that they allow a concise definition of different properties.

The expression of interaction protocols is realized directly in temporal logic by using the base operators defined by [Clarke 00]. Due to the fact that the verification and execution of the specifications is very expensive both in terms of time and space, there are only a small number of executable temporal logics.

4.2.5 Other Approaches

Some other research efforts were directed towards developing object-oriented languages that natively integrate interaction protocols. One example of this kind of language is PROCOL [van den Bos 89]. PROCOL (for PROtocol-constrained Concurrent Object Language) is an OOL with strong support for explicit parallelism. It also integrates explicit protocols to control access by communication. The protocols specify the object interaction but only from a server point of view, that is the provided interface services. Transitions in protocols contain guards written in the C language. PROCOL also provides an execution environment controlling the object interactions in order for the protocols to be correctly executed. One important drawback of this language, however, is the lack of a formal model allowing the analysis and verification of important properties.

The *reactive programming* allows the design of independent applications communicating through synchronous or asynchronous events. A reactive application specifies the actions to be realized in concordance with the already treated events traces. In other words, a reactive application executes an interaction protocol. The development stages of a reactive application include the specification, prototyping, simulation and validation. The specification consists in defining the actions to be taken in function of external events. Examples of reactive languages include Esterel [Berry 92] and the proposal presented in [André 96]. However, reactive languages are generally too low level and are not providing enough high-level abstractions in order to allow the design of software components.

4.3 Component Models and Interaction Protocols

In the previous section we have presented the formalisms used in order to describe interaction protocols. In this section we present different component models and languages currently integrating interaction protocols described by either of the formalisms previously presented.

4.3.1 Automata-Based Models

Darwin (see Section 3.3.4.1 at page 80) addresses the issue of component behavior. Behavior is specified in terms of LTSs and properties verification is realized by using the Finite State Process (FSP) algebra. Properties are separated into two classes: safety and liveness. For a complete description of the behavioral model see Section 3.3.4.1 at page 81.

Barros *et al.* [Barros 05] propose a Fractal (see Section 3.2.2.1 at page 67) extension in order to introduce behavioral descriptions at interface level. Their model is an adaptation of the *symbolic transition graphs with assignment* of [Lin 96] into the *synchronization networks* of [Arnold 94]: they extend the general notion of Labeled Transition Systems (LTS) and

hierarchical networks of communication systems (synchronization networks) by adding parameters to the communication events in the spirit of [Lin 96].

A parameterized LTS is a LTS with parameterized actions and with a set of parameters (defining a family of similar LTSs) and variables attached to each state. Parameters and variables types are simple. Additionally, the transitions can be guarded and have a resulting expression which assigns the variables associated to the target state.

They describe both the functional behavior and the non-functional features (life-cycle management) of components in terms of synchronized transition systems. They define a notion of correct component composition and they show how to prove temporal properties of a component system using compositional model-checking techniques. Reconfigurations of a system, for example replacement of a sub-component, are expressed as transformations of its behavioral semantics, allowing to prove preservation of some properties, or the validity of new properties after transformation. The concept of *Correctness* (with respect to behavior) covers the absence of dead-locks and general safety and liveness properties. After reconfiguration, the preservation of some properties valid before transformation and the satisfaction of a new set of properties, corresponding to features added by the transformation. These proofs take into account the intricate interplay between functional and non-functional actions during transformation, like the management of the internal state of components.

A Java tool has been developed that automatically and incrementally generates the synchronization files for a component system from its description and the CADP [4] tool is used to calculate the synchronous product, minimize the systems, and model-check the formulas.

4.3.2 Regular Types

CwEP (acronym for Components with Explicit Protocol) [Farias 03] is a component model integrating explicit interaction protocols. CwEP also specifies a notion of *identity* at interface level. A protocol defines the availability of provided services, specifies the component interactions and supports point-to-point communication between components. The identity is used in order to identify component collaborators.

In addition to the enhanced interface definition, the CwEP model also defines a notion of protocol refinement (based on the notion of substituability defined by Nierstratz, Section 4.2.2.1 at page 97) and coherence between interface description and implementation, and some compositional operators for protocols and components.

One particularity of CwEP is that it is not hierarchical in the structural sense. A CwEP application is viewed as a flat hierarchy of interacting components that ask for service execution to their partners. The composition of two or more components is another component simply regrouping the provided services of the constituting components. The communication is of synchronous type and realized exclusively through component interfaces (one component actually implements only one interface). Protocol refinement defines a notion of refinement of regular types and is specified in function of traces and failures (see Section 4.2.2.1 at page 97). The notion of coherence specifies that a protocol and the

4. http://www.inrialpes.fr/vasy/cadp/

component implementation are coherent if the protocol specifies the requests realized by the component and vice-versa. Its verification is realized by extracting a protocol starting from the implementation and is compared to the protocol defined at interface level. If the implementation protocol can be substituted to the interface protocol and vice-versa the conclusion is that the implementation is coherent with the specification. At a more formal level of explanation, the two protocols must have the same traces and failures in order to be substitutable.

Yellin and Strom [Yellin 97] studied the integration of behavioral protocols with component interfaces. They do not propose a component model but define a notion of interface compatibility where the typed interfaces define the components. The protocols, similar to the regular types of Nierstratz (see Section 4.2.2.1 at page 97), deterministically describe the component interactions in terms of emissions and receipts. The interactions take place under a synchronous semantics: the two protocols (one for each component) must be in a state where one component sends a message and the other component receives the same message. Notions of compatibility and substitutability are proposed. They both consider the request/response directions of message passing. A theorem guarantees that if a protocol p is compatible with another protocol r, then the protocol q, subtype of p is also compatible with the protocol r. Based on these notions of substitutability, Yellin and Strom propose a methodology allowing the generation of component adaptors. Thus, incompatible components get the chance to be connected in an architecture.

Cyril Carrez [Carrez 03] defines a behavioral interface type language constituting a behavioral contract. This language is inspired from the regular types by following a message passing semantics. The notions of *allowed* and *mandatory* actions (inspired from the deontic logic [Lomuscio 04]) are considered, imposing restrictions both on the component itself and on its environment. The language specifies behavioral contracts used to realize two types of verifications:

- that the component respects its interface contract: a component semantic and rules on how to verify the coherence between component internal behavior and interface behavior are defined
- at composition time, that interconnected interfaces are compatible

These kinds of verifications continue the trend started by [Abadi 93, Abadi 95] and guarantee that each sent message will be consumed and that there will be no deadlock between components.

4.3.3 Coordination-Based Models

Arbab *et al.* [Arbab 02] propose a coordination (see Section 3.3.7 at page 89) model for component-based software systems based on the notion of *mobile channels*. Channels allow anonymous and point-to-point communication among components, while mobility allows dynamic reconfiguration of channel connections in a system. From a software development point of view, mobile channels provide a highly expressive data-flow architecture for the construction of complex coordination schemes, independent of the computation parts

of components. This enhances the reusability of systems: components developed for one system can easily be reused in other systems with different coordination schemes. Coordination schemes include *messaging*, *events*, *shared data spaces*, and *channels* [Arbab 02]. Channel employment also decouples the component updates from that of channels updates.

In the presented model, a channel is called *mobile* when the identities of its channel-ends can be passed on through channels to other components in the system. Furthermore, in distributed systems the ends of a mobile channel can physically move from one location to another, where location is a *logical address space* where component execute. Because the communication via channels is *anonymous*, when a channel-end moves, the component at its other end is not affected.

The model can be implemented in any modern programming language. The authors chose to describe the implementation guidelines in the Java programming language.

The coordination model based on mobile channels opens the possibility to apply more powerful coordination paradigms. One example of paradigm is $P\epsilon\omega$ [Arbab 01] supporting composition of channels into complex connectors whose semantics are independent of the components they connect to.

4.3.4 Other Approaches

SOFA component (see Section 3.2.2.2 at page 69) behavior is specified by a formalism close to the *regular expressions*. In SOFA, each method call and return is realized as an atomic event. Each event is represented by a particular symbol in order to distinguish between emission and receipt of calls and returns. Sequences of symbols define traces and the behavior of a SOFA component is defined as the set of all traces that can be produced. The CDL compiler in SOFA is capable of verifying the coherence between a behavioral specification and an implementation.

PACOSUITE [Vanderperren 03] is a visual environment used in component composition. PACOSUITE provides a language employed in order to document behavioral specifications called *scenarios*. A scenario is specified as an message sequence graph (under the form of a UML state diagram) and describes the possible interactions between a specific component and other abstract entities (component descriptions).
A composition pattern defines the interactions among multiple abstract components and due to its abstract nature it can be easily reused. While trying to compose components in terms of behavior, adaptors are automatically proposed in order to deal with incompatibilities. However, the PACOSUITE approach is reduced to a mechanism of component documentation (compound components included) but no coherence between specification and implementation is guaranteed.

MIDAS [Pryce 98] is a specification language for interaction protocols among concurrent components in a distributed environment. MIDAS descriptions are annotated with formal specifications of protocols and verifications over these descriptions are realized in the conceptual phase of the component development.
A component encapsulates a state and realizes a behavior defined by its roles. A role can specify a set of provided or required services and consists of a name, a type and final interaction point. Interactions are described like finite state machines and are asynchronous.

They are also bidirectional, concerning only two interacting components. Like the other approaches presented in this section, MIDAS does not provide tools in order to verify the coherence between specification and implementation.

4.4 Conclusions

In order to successfully interact, components need to conform to a certain form of interaction contract (interaction protocol based on the notion of behavior protocol). Interaction protocols describe the entity behavior in terms of sequences of messages exchanged between a component and its environment. In this chapter we have presented the formalisms usually employed to specify this kind of component interaction contracts and we illustrated the integration of some types of formalisms inside fully fledged component models or less developed models but with great impact on component software development (see for example the work of Yellin and Strom, Section 4.3.2 at page 103).

The main criticisms that can be made regarding some of these formalisms is the abstraction level, much too weaker to describe higher level entities like components. Some other formalisms sacrifice expressiveness in order to increase decidability. While developing object models, important notions like substitutability and refinement made their appearance, easing the way to developing satisfactory component models integrating protocols.

In component models, interaction protocols usually specify component externally visible behavior. The specification is done at interface level and enhances purely structural interfaces with behavioral descriptions. Practically, none of the presented component models consider the coherence between the specification and the implementation. Excepting for CwEP (see Section 4.3.2 at page 102), the other component models have a rather evasive approach when considering implementation issues. However, CwEP is only flat hierarchical component model and structural hierarchical composition is not considered at all.

What misses is a component model integrating a form of interaction protocol that is readable, expressive and easy-to-use by an average component designer. The implementation has to be part of the main concerns when developing the component model as the coherence between specification and implementation is essential when applying the model in a real software development. Generative techniques, either by using an MDA [5] approach or a rather empirical one, represent an appealing research field.

The second part of this thesis presents our proposal: a component model based on Symbolic Transition Systems (STS) focusing on the description of interaction protocols and on the component implementation by using a generative approach, guaranteeing thus that the implementation is coherent with the specification.

5. Model Driven Architecture

Part III

Contribution in English

Chapter 5

A Component Model with Explicit Interaction Protocols

In this chapter we present CwSTS (Components with STS), a simple, yet general component model that we propose in order to explicitly integrate interaction protocols. Following a generative approach, we present the component model details in an informal fashion. Components, interfaces, compatibility and adaptation are all exemplified by a guideline example. Next, we present a formalized view over CwSTS and we end up by CwSTS-IDL which is our component model interface description language.

Contents

5.1 Introduction

In this chapter we present a simple, yet general, component model we baptize CwSTS (acronym for *Components with Symbolic Transition Systems*). Component models and languages presented in the state of the art integrate too many concepts or are too specific. In this context, it is very difficult to analyze the consequences of adding new concepts (like interaction protocols) or functionalities in an already defined, fully-fledged component model. We make the choice of proposing a new, general, model integrating some minimal features. CwSTS proposes, in particular, interaction protocols both at the conceptual and at the implementation level.

CwSTS is designed as a simple black-box component model (for example, the communication among components is realized exclusively through their interfaces) integrating only some features like a unique interface. This interface consists of two parts: a *structural interface* and a *behavioral interface*. While the structural interface describes the signatures of the services provided and required by the component, the behavioral interface (given under the form of an interaction protocol) describes the rules that govern the behavior of the component in terms of message emissions and receipts.

The interaction (communication) between components is point to point (binary) and is realized at the level of individual services from the interface. This means that communication 1 to 1 (from a component defining a required service to a component defining a corresponding provided service) and oriented (one way, no rendez-vous as in process algebras [Bergstra 01, Turner 93]). Our model considers only static architectures, that is, once created, an architecture cannot be modified at execution time. Components cannot be dynamically created and bound in the architecture, nor can they be eliminated.

We voluntarily restrict the set of features of the CwSTS model (only one interface, communication is 1 to 1, only static architectures) in order to simplify the model and the implementation and easily analyse the integration of interaction protocols at the level of component interfaces. However, the model can be extended in order to provide communication multiplicity for example. Future work related to our component model is suggested in Section 7.1 at page 163.

Our model is based on a hierarchical view of component composition. Two or many components (also called *primitive components*) can be composed in a unique component (also called *composite component*). Regarding its interfaces and behavior, the composite adheres to the same rules of the component model as the initial primitive ones. Further on, the newly obtained composite component, can be used, as if a primitive component, in other compositions. This corresponds to the *Composite* design pattern [Gamma 95] approach.

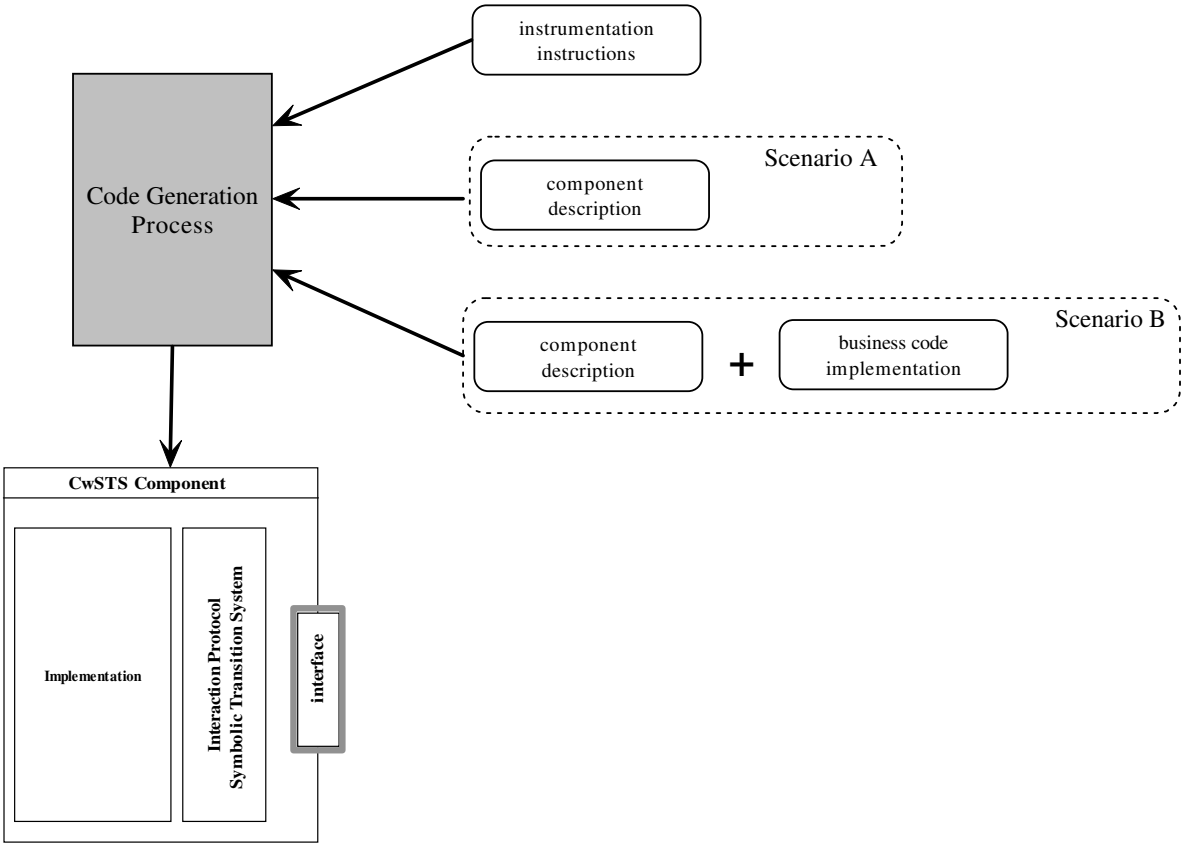


Figure 5.1: A Generative Approach to Components.

5.1.1 Components, a Generative Approach

We follow a generative approach to constructing CwSTS components. One of the benefits of such an approach is that it ensures that components implementations are consistent with their interface descriptions. This concern is of major importance in CBSE as most of the component state-of-the-art models and languages decouple component specifications activity from the implementation stage in a way that makes it hard to ensure that implementation conforms to specification.

Beside the component model definition, our interest is in providing associated tools allowing the maximum of code generation for the given CwSTS component descriptions. More specifically, we base our proposal on a two-scenario approach when automatically generating component implementation (see Figure 5.1). Naturally, the characteristics of the CwSTS model we propose are adapted towards the realization of the two scenarios:

- **Scenario A:** We start from one or many component descriptions (i.e. both structural and behavioral descriptions) and together with some instrumentation instructions we pass them to a generation engine that creates one CwSTS valid implementation per

provided component description. This implementation comes either with a default business implementation or an adaptor-oriented implementation. The default business implementation contains no business code and it is suited to be extended in order to address the specificities of the desired implementation.
Adaptor components (i.e. structural and behavioral adaptation) generated by following this scenario, are used in an architecture where existing components need adaptation code in order to be successfully interconnected. The adaptor code needs no further modification in order to be inserted in an architecture.

- **Scenario B** is different from Scenario A in that we start from existing code that need to be instrumented in order to obtain a valid CwSTS component implementation rather than starting from component definitions. In this scenario (as the business code is provided independently of the generated code), the component description must be *consistent* with the characteristics of the provided implementation code (both structural and behavioral). We can ensure this constraint by applying analysis techniques and tools (although we do not address this kind of concern in this thesis). This scenario allows the creation of component implementations from already existing business code and guarantees that the specification is consistent with the generated component implementation code.

For the remaining of this chapter, we begin (Section 5.2) by giving an informal presentation of our component model. We describe what a component is, how we compose it in an architecture and what is its life cycle. Next, we continue with a more formal definition of the component model (Section 5.3 at page 122). We present the use of Symbolic Transition System protocols in our model in Section 5.3.2 at page 123. The composition properties are discussed in Section 5.2.3 at page 116 (informal) and Section 5.3.3 at page 125 (formal) and the associated component language (CwSTS-IDL) is presented in Section 5.4 at page 128. Finally, Section 5.5 at page 133 concludes on the proposed model.

5.2 Informal Presentation

CwSTS is a simple (yet general) black-box, hierarchical component model explicitly integrating interaction protocols in the component interface descriptions. As a specificity of our proposal and for simplicity reasons, a CwSTS component implements only one interface. This interface consists of two parts: a *structural interface* and a *behavioral interface*. While the structural interface describes the signatures of the services provided and required by the component, the behavioral interface (given under the form of an interaction protocol) describes the rules that govern the behavior of the component in terms of message emissions and receipts.

As in any other model where we explicitly consider both provided and required service descriptions, a component plays a double role: the server and the client role. When treating calls to its provided services, a component conforms to the role of a server. On the other hand, in order to provide the specified service, a component may rely on other services

provided by other components. When calling services on other components, the component plays the role of a client.

In order to clarify things, when we talk about a *message emission* event we consider the encapsulation of the service call in a message that passes over the communication medium from the caller to reach the corresponding component really implementing the service. A *message receipt* event does not automatically mean message treatment (i.e. service execution on the receiving component). It only means that the message (the call of a service) was received (passed the component interface frontier) but not necessarily executed. The actual execution of the message (service) is realized depending on the implementation of the component.

In other words, when looking at a component interface, we make no assumption on the actual execution of the messages passing through the interface level of a component. The execution can be synchronous (the execution takes place as soon as the message was received) or asynchronous (the execution takes place at another non specified time after the message receipt). We consider CwSTS components as active entities (running one or many execution threads, depending on implementation) and all we require in this model is that the component supplies an implementation for each provided service in the interface and that at execution time, the business logic encapsulated in the component conforms to the behavioral description given by the interaction protocol.

We consider a black-box, hierarchical model where components can communicate only through their interface and where the composition of two components (also called *primitive components*) results in a third component (also called *composite component*) that can be later used in further compositions. This composition schema conforms to the *Composite* design pattern [Gamma 95] but in addition to the pure structural properties of the composition, our model also considers composition at the level of behavioral description (interaction protocols). As a specificity of our model, the composite components contain only architecture descriptions (no business code), the actual computation being provided by the interaction of its subcomponents. A composite component also guarantees that its behavior conforms to the synchronous product of the protocols of its subcomponents.

The interaction (communication) between components is point to point (binary) and is realized at the level of individual services from the interface. This means communication is 1 to 1 (from a component defining a required service to a component defining a corresponding provided service) and oriented (one way, no rendez-vous as in process algebras [Bergstra 01, Turner 93]). Our model considers only static architectures, that is, once created, an architecture cannot be modified at execution time. Components cannot be dynamically created and bounded in the architecture, nor they can be eliminated.

Another particularity of our approach is that we voluntarily restricted the set of features of the CwSTS model (only one interface, communication is 1 to 1, only static architectures) in order to simplify things and easily analyse the integration of interaction protocols at the level of component interfaces. However, the model can be extended in order to provide communication multiplicity for example. Future work related to our component model is suggested in Section 7.1 at page 163.

We also propose a component language associated to our CwSTS model (see Section 5.4 at page 128). This language (CwSTS-IDL) is an *Interface Description Language* allowing

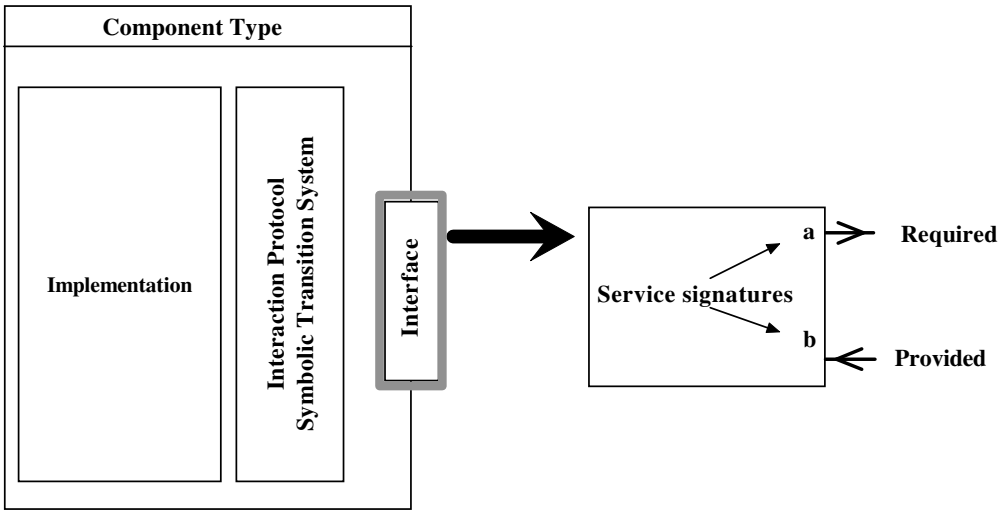


Figure 5.2: Graphical Component Syntax.

for the description of component interfaces (including a Behavioral IDL for the interaction protocol descriptions). Composite component architectures are also described by using this language and include the declaration of subcomponent type instances and connection directives.

5.2.1 Components

A CwSTS *component* is a software unit implementing a unique *interface*. The interface is composed of a set of service signatures representing the services the component provides to or requires from the environment and of an interaction protocol description specifying the rules ordering the message exchange between the component and its environment (the graphical syntax of a CwSTS component is given in Figure. 5.2)

The interaction between two components is realized by sending a service execution request message from the component requiring the service to the component actually providing (and implementing) the specified service. From an operational point of view, the component requiring a service execution sends a message that will be received and eventually executed by the component providing the service.

In order to deal with the difference between a *component type*, a *component implementation* and a *component instance*, we will further talk about component types when considering component definitions. The component implementation is given by the set of binaries or sources realizing the interface specified in the component type. A component instance is an instance of a component type used in a specific configuration, at runtime.

5.2.2 Interfaces

The interface of a CwSTS component type consists of two parts: a structural interface and a behavioral interface.

The structural interface describes the signatures of services either provided or required by the component. The structural interface corresponds to the first level of contracts in the taxonomy of Beugnard *et al.* [Beugnard 99]. The behavioral interface is specified under the form of an interaction protocol. The protocol plays an important role in the component interaction as it, on the one hand, specifies if a request to its services can be accepted at a certain moment or not, and, on the other hand, it also specifies the moments when the component itself is susceptible to generate a service request to another component.

From an informal point of view a protocol can be defined as:

Definition 4 (Protocol) *A component interaction protocol describes, at any moment in the execution of a component instance, the availability of a service request event (client role) or service receipt event (server role) depending on the actions performed before.*

Protocols are often depicted by using finite state automata [Hopcroft 01]. A protocol defines states and transitions. We use a FSM [1]-based approach in order to integrate protocols. In our model, a state in the protocol represents an *abstract state* in the component instance execution. A transition represents an action executed by the component. Actions can be of three types: message reception, message emission and internal action. Message receipt/emission is in correspondence with the service request. The internal action is the execution of an operation with no externally visible effect. Transitions can be guarded by boolean expressions. This means that a specific transition (and its corresponding action) cannot be realized while the corresponding guard is not true. Guard operations are restricted to not introduce side effects like, for example, an operation execution in order to change component state.

Protocols in CwSTS correspond to the third level in the taxonomy of Beugnard and al [Beugnard 99]. They are based on the *Symbolic Transition System* (STS) formalism. Thus, interaction protocols in the CwSTS model consider parameterized actions and parameterized boolean operations guarding transitions.

From an operational point of view, the structural interface is a *static interface* as it does not change during the time of the execution, while, from a liberal point of view, the behavioral interface can be seen as a *dynamic interface* as it describes the services that are available at specific moments in time in the execution of the component.

The interaction protocol depicted in Figure 5.3 describes the behavioral interface of a component playing the role of a typical `Client` component registering to a messaging server in order to receive particular message types. The structural interface of the `Client` component includes the required services `login`, `logout`, `subscribe` and `unsubscribe`, each of them with their corresponding formal parameters. The provided service `messageEvent` is also part of the static interface.

The typical execution of this component is that it logs in to a messaging server and subscribes to one *topic* (i.e. of interest to the component). After subscribing, the *Client* component is waiting in order to receive messages of the specified topic from the server (the *messageEvent* action). At any time after subscribing to a particular topic, the component can unsubscribe to that topic in order to subscribe to another one or simply log out and stop.

1. Finite State Machine

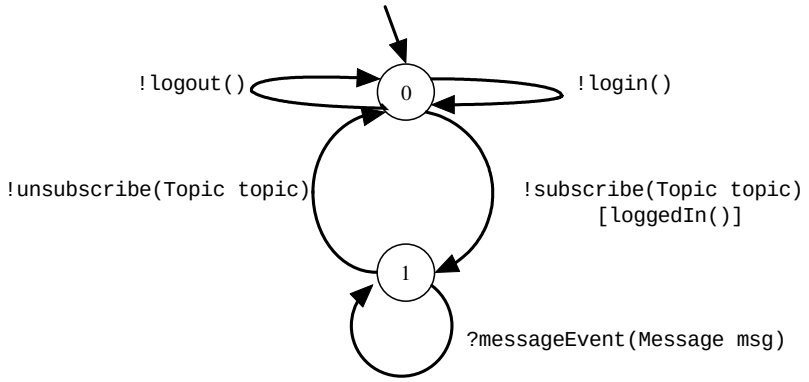


Figure 5.3: Message Client Protocol.

The protocol is composed of two abstract states 0 and 1. All the services defined in the structural interface are represented as actions forcing the transition from one state of the protocol to another one. The transition realizing the action is also guarded by the evaluation to true of the internal boolean action `loggedIn()` assuring that this action (the corresponding message emission) cannot occur while the guard action does not evaluate to true. In this case, the `Client` component is not allowed to subscribe to a topic before being logged in. As a particularity of our model, guards can be parameterized with values of the actual parameters of the transition actions (as shown in Figure 5.7 at page 119).

5.2.3 Composition

Composition in CwSTS is hierarchical in the sense that it adheres to the composite design pattern [Gamma 95]. When composing two or more components (also called primitive components) we obtain a composite component that can be used in further compositions as if it were a primitive one. Composite components do not contain any business logic, they simply constitute an aggregate of primitive components that are actually implementing the business logic dealing with input messages or generating output messages.

The composition is realized both at the structural and the behavioral interfaces levels. At a structural level, CwSTS includes the *binding*, *import* and *export* composition schemas described in Section 3.2.1.3 at page 64. The resulting static interface of the composite depends on the schema used to compose the primitive components. The behavioral interface of the composite is given by the *synchronous product* [Godefroid 91] of the primitive component interaction protocols.

Figure 5.4 depicts the composition of two components `c1` and `c2`. From a structural point of view, the resulting composite component only *exports* the same service `a` as the `c1` component. The requested service `b` from `c2` is *bind* to the corresponding service `b` from `c1`. From a behavioral point of view, the interaction protocol of the `Composite` component is the resulting synchronous product (see Figure 5.5) between the protocols of the `c1` (where the `b` action is possible only after the `a` action was realized) and `c2` components where the internal `b` action is hidden as it is not relevant, in terms of emission or reception, from the

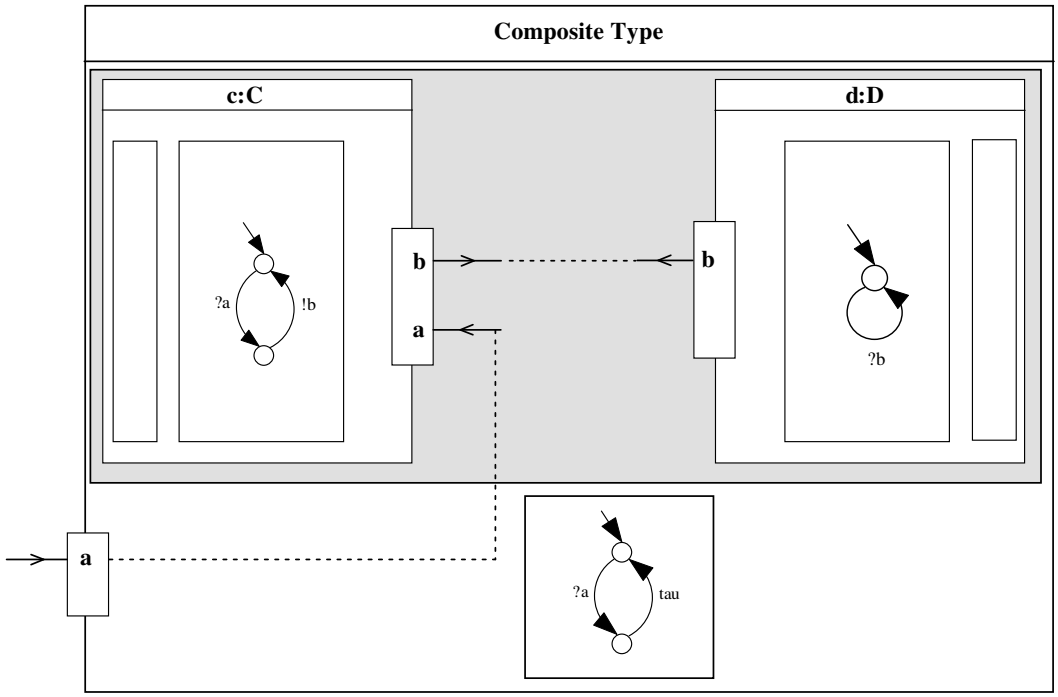


Figure 5.4: Composite Component Example.

exterior of the composite component.

As a practical example let us consider the composition of two components *Server* and *StatisticsReporter* that we integrate in order to obtain a composite component actually implementing the messaging server. Figure 5.6 depicts this composite component waiting for topic subscriptions from clients and logging statistics about messages sent to those clients. The protocol illustrated in this figure represents the synchronous product of the two protocols of the *Server* and *StatisticsReporter* components (see Figure 5.7 and Figure 5.8 at page 119) but the internal actions are hidden as they are not relevant to the exterior of the composite component.

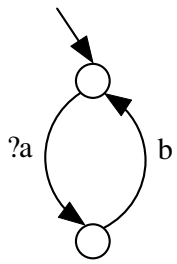


Figure 5.5: The Synchronous Product.

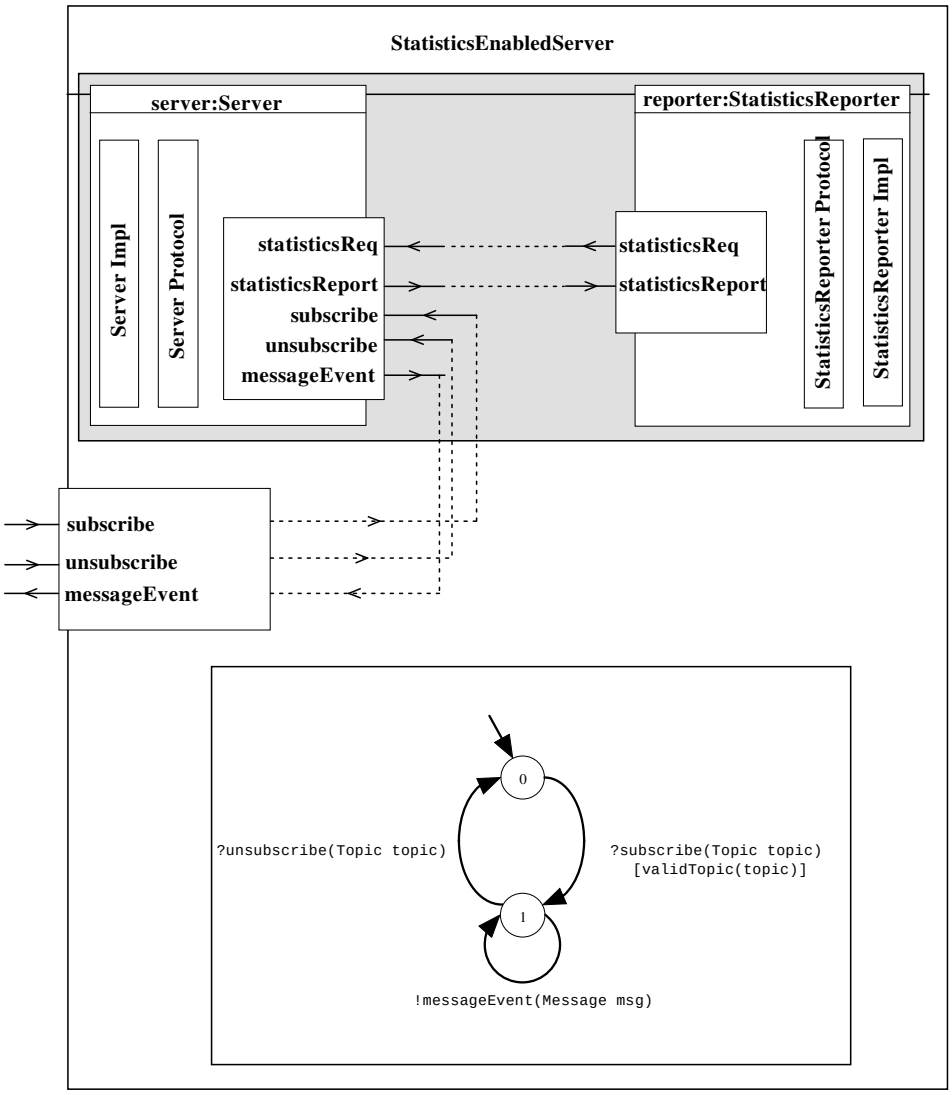


Figure 5.6: StatisticsEnabledServer Protocol Representing the Synchronous Product of the Server and StatisticsReporter Protocols with Hidden Internal Actions.

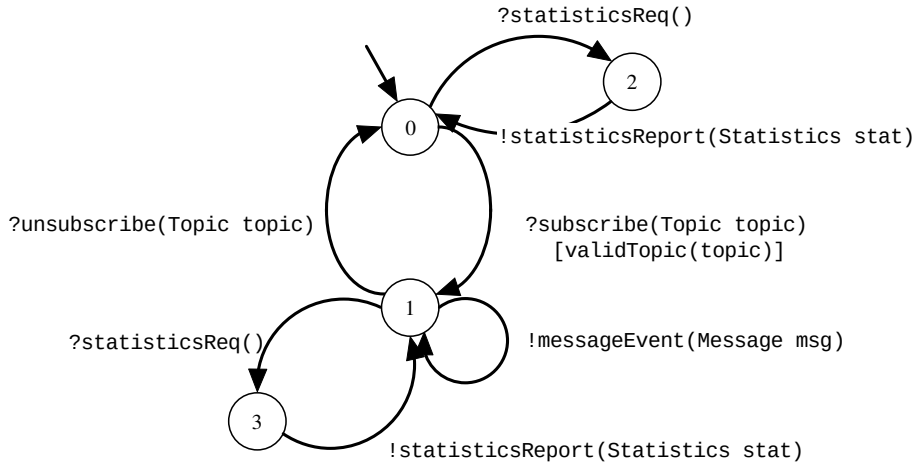


Figure 5.7: The Server Protocol.

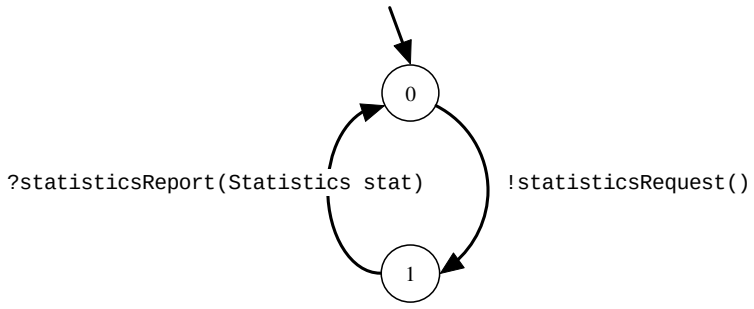


Figure 5.8: The StatisticsReporter Protocol.

5.2.3.1 Compatibility

When composing primitive components in an architecture, the components interfaces compatibility is of major importance. From a structural point of view, the *bind* between two services that are not compatible is not possible. From a behavioral point of view, the protocols of the components involved in the composition need also to be compatible. If this condition is not satisfied, the global behavior of the composite component is, in the best case, not satisfying the safety properties (i.e. no deadlock) that are usually expected in such a case.

From an informal point of view, two or more components are compatible in an architecture if:

1. the bind services between any two components are compatible (structural compatibility),
2. the interaction protocols of all the components involved in the composition are also compatible (behavioral compatibility),
3. the architecture is valid.

In CwSTS, we consider that two services are compatible if they have the same name and signature (i.e. formal parameter list). This might seem like a severe limitation as it might be unacceptable not to be able to bind two services that are different only in their name or the names of their parameters. For the sake of simplicity in CwSTS, we propose to consider services structurally compatible if their name and signature match but we address the problem of incompatibilities by employing adaptors (as explained in Section 5.2.3.2 at page 121). This also solves the issue of component substitutability: when replacing a component in an architecture with another one the architecture must remain compatible. Adaptors are of big importance in this case of situations.

Protocol compatibility refers to the absence of any deadlock in the resulting synchronous product of the involved protocols. A deadlock (the execution of the composite component would result in a deadlock) signifies that the subcomponents cannot be successfully integrated in an architecture.

Valid architectures refer to the fact that, in an CwSTS architecture, not all component services must be bound. The actual services that are required to be bound is determined by the architecture itself. Some services could be connected in one architecture while they are not in another completely different one. Environmental context abstraction is of major importance when constructing components as it makes it possible that a component can be assembled and deployed in many different architectures and environments.

Checking architecture validity consists in removing the states and transitions corresponding to the services not connected in an architecture from the synchronous product of the component protocols and test for deadlock. If a deadlock occurs, it means that there is at least a service that must be connected in the given architecture. The states and transitions corresponding to the services not connected are those that cannot be reached in executing the synchronous product of the component protocols.

A valid architecture does not automatically mean a complete executable one. A composite component that exports or imports services from or to its subcomponents is usually not executable as its execution depends on the effective connection of these services in a larger composite. We consider a complete architecture (and so it is executable) a valid architecture where the largest composite component (actually representing the complete architecture) does not import any services.

5.2.3.2 Adaptation

In case where two components are not compatible (both structural and behavioral) as is, the CwSTS model proposes the use of adaptor entities (actually adaptor components) that can be employed in order to adapt components interfaces. At the structural interface level, an adaptor can act in the sense of changing services names, parameters order or parameters instrumentation (combination of two parameters in a single one, etc.) thus implementing the Adaptor design pattern [Gamma 95]. In this way, two services that are initially not compatible can be finally connected by introducing a level of indirection (the adaptor component itself). At the behavioral level, an adaptor component interaction protocol can be used to prevent deadlock from the resulting synchronous product and, thus, enable the interconnection of the given initial components.

Adaptor components are first class entities in CwSTS. Due to their function in an architecture, their implementation code can be automatically generated from their structural and behavioral interface description.

Let us reconsider our running example where a *Client* component needs to login to a messaging *Server* component in order to subscribe a topic and receive messages. The problem is that the *Client* component cannot directly connect all its services to those of the *Server* or even *StatisticsEnabledServer* components. This is due to the fact that the *Server* component does not require a `login` and `logout` behavior. In order to interconnect the client and the server an adaptor component is used. Figure 5.9 depicts the *Adaptor* component protocol and the complete architecture of our example is shown in Figure 5.10 where we indicate only the components involved in the architecture and the sense of message passing.

In this specific example the adaptor entity intercepts only the interactions that cannot be directly realized by the interconnected components. The adaptor component could have also intercepted the `subscribe` and `unsubscribe` messages but this is not necessary because these messages can be directly exchanged between the `Client` and `StatisticsEnabledServer` components respectively and because the global order of message exchange is not influenced by the fact that they are not intercepted by the `Adaptor` component.

5.2.4 Life Cycle

The lifecycle of a CwSTS component consists of four phases:

- *Creation.* The component is defined by describing the interface (both structural and behavioral parts) and providing the implementation of this interface. In this phase

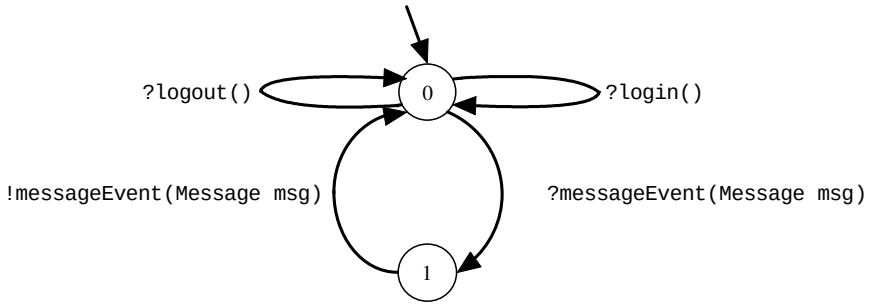


Figure 5.9: Adaptor Component Protocol.

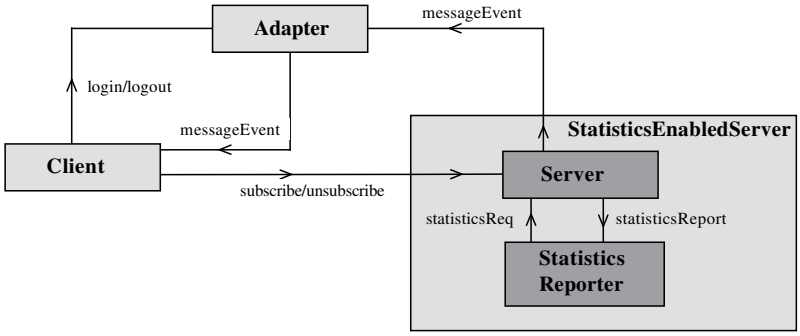


Figure 5.10: Complete Architecture of the Case Study.

we talk about component types.

- *Assembly.* The components are composed in order to obtain composite components and finally a complete architecture that can be deployed and then executed as is. Component types are combined in order to obtain another component types.
- *Deployment.* Component types are instantiated (at this level we talk about component instances).
- *Execution.* Component instances are initialized and then executed. At runtime, from an external point of view (reception/emission of messages), each component behaves as described by the interaction protocol (its behavioral interface). At the architecture level, the global execution of the system conforms to the resulting synchronous product of all the component instances behavioral interfaces.

5.3 Model Definition

In this section we give a more formalized definition of the notions of component, interface, interaction protocol and operational semantics.

5.3.1 Components

Definition 5 (Component Definition) *A component c is described as* $c = (I^c, Impl^c)$, *where* $I^c = \langle i^c, p^c \rangle$ *is the interface of c defined by a structural interface* i^c *and a protocol* p^c *and* $Impl^c$ *represents the implementation code realizing* I^c.

In this definition:

- i^c is given as a set of service definitions (name and formal parameter list).
  $i^c = s^- \oplus s^+$ where s^- is a required service and s^+ is a provided service.
  If we define n^c as the set of non connected services $n^c \subseteq i^c$, then $i^c \backslash n^c$ is the complementary of n^c in i^c,
- p^c is given under the form of an STS defining an automaton were the alphabet is constituted from the parameterized actions encoding either message receipt or emission (see Section 5.3.2),
- $Impl^c$ is either the source or the binaries of the code realizing the component interface I^c.

5.3.2 Interaction Protocols

Symbolic Transition Systems (STS) [Calder 02, Ingolfsdottir 01] have initially been developed as a solution to the state and transition explosion problem in value-passing process algebras using substitutions associated to states and symbolic values in transition. This formalism extends the LTSs (Labeled Transition Systems) formalism with parameters and explicit guards on transitions.

We choose to describe the interaction protocols in CwSTS by using a generalization of the STS formalism. Our proposition associates a symbolic state and transition system with a data type description given as a set of Java classes (the implementation of a CwSTS component). This kind of association is not new. Several authors proposed to give the data type description under the form of an algebraic specification [Choppy 00, Royer 03a] or under the form of a model-oriented specification [Attiogbé 03]

In the following, we give the formal definitions of Parameterized Actions, STSs and Composition and an introduction to the graphical language associated to the STSs.

5.3.2.1 Parameterized Actions

Our STS proposition considers parameterized actions. The two actions that are related to transitions in the symbolic system are the actions representing the reception of a call to a certain service in the interface or the actions representing a call to a specific service on another component.

Each action is parameterized with a set of formal parameters (see for example the *Server* component in Figure 5.7 at page 119 where the `subscribe` transition represents an action with the `Topic topic` formal parameter). As in our model we forbid component types in interface descriptions (in order to preserve the integrity property [Aldrich 02b]) we

allow any type from the implementation target language (Java in our case) to be present in the definition of the parameters.

Definition 6 (Parameterized actions) *Parameterized actions are* $?s(\vec{x})$ *encoding the reception of a call to the service* s *(*$\vec{x}$ *will be affected by the arguments of the call) and* $!m(\vec{e})$ *encoding a call to the service* m *of another component with the arguments* $\vec{e}$*.*

In Definition 6, $\vec{x}$ and $\vec{e}$ represent the list of the formal parameters of the s and m services, respectively. These formal parameters will be affected by the arguments of the service call at execution.

We say that two actions are *complementary* if they are defined in different components and have the same name and formal parameter lists. These actions are the operational correspondent of the provided and required services defined in the static interface. Complementary actions are intended to be considered as unitary at execution.

5.3.2.2 STS Definition

Definition 7 (STS) *A Symbolic Transition System (STS) is a tuple* $\langle S, A, \Delta, s_0 \rangle$ *where:*

- S *is a finite set of states,*
- A *is the set of* parameterized guarded actions *(*pga*) where* pga $= (g(\vec{x}), a(\vec{x}))$*:*
  - g *is the boolean parameterized action representing the transition guard, and*
  - a *is a parameterized action.*
- $\Delta \subseteq S \times A \times S$*, denotes a transition relation that maps from a state and an action onto another state,*
- $s_0 \in S$ *is the initial state. In CwSTS we consider only one initial state representing the fact that the execution already starts with that particular state.*

Next we give the definition of a transition from a particular state in the STS to another state (considered as the initial state of a subprotocol of the initial protocol).

Definition 8 (STS Transition) $P \xrightarrow{[g]a} P'$ *denotes the guarded transition from STS* P *to STS* P' *where:*

$P = \langle S, A, \Delta, s \rangle$*, and*

$P' = \langle S', A', \Delta', s' \rangle$*, and*

$(g, a) \in A$ *and* $(s, (g, a),\ s') \in \Delta$*.*

The transition from one state to another state can represent either a message emission, a message receipt or an internal action. We already stated that CwSTS protocols do not specify if the message receipt also means message execution. It only specifies that a message was received. The actual component implementation can consider a synchronous or asynchronous execution of the message that was received.

Another important remark is that guards block the receipt or emission of a message. The complementary emission and receipt are realized in the same logical time so the guard actually blocks the two complementary actions (emission and receipt) on the two collaborating components.

5.3.3 Composition

Definition 9 (Component Composition) *If* $c=(\langle i^c, p^c\rangle, Impl^c)$ *and* $d=(\langle i^d, p^d\rangle, Impl^d)$ *and* $+_c$ *is the component composition operator then:*
$c +_c d = (\langle i^c +_i i^d, p^c +_p p^d\rangle, Impl^c +_{||} Impl^d))$.

In this definition:

- the $+_i$ operator denotes the composition of the static interfaces of c and d,
- the $+_p$ operator denotes the composition of the protocols of c and d,
- the $+_{||}$ operator denotes the parallel execution of the implementations code it takes as arguments.

5.3.3.1 Static interface composition

The result of the $i^c +_i i^d$ operation is the set representing all the services defined by the two interfaces but without those services that are actually bind in the architecture. Implicitly, two complementary (one required and the other provided) services with the same name and signature are bind in the architecture. In order to simplify things in this definition, we consider there are no two services (either provided or required) with the same name and signature defined by different components and that could introduce confusion when using this operator. In practice, this kind of situation is allowed.

With this restriction $i^c +_i i^d = (i^c \cup i^d)$ - $\{s_m, s_n \mid s_m \in i^c$ and $s_n \in i^d$ and s_m has the same name and signature as s_n, s_m and s_n represent either a required service s^- or a provided service s^+ and s_m and s_n are connected in the architecture$\}$.

Definition 9 considers the composition of only two components. Below we give the definition for the composition of n static interfaces where each static interface can be restricted (by using the \ operator) to the services that are actually connected in the specific composition architecture.

Definition 10 (Static Interface Composition) *The static interface composition operator is defined as:*

$$+_{i_{m=1}^N}(i^m \backslash S) = \bigcup_{m=1}^N i^m \text{ - } \{s | s = s^- \text{ or } s = s^+ \text{ and } s \text{ is bind in the architecture}\}.$$

5.3.3.2 Behavioral interface composition

$+_p$ represents the composition of the interaction protocols of the components in the configuration. The semantics of the composition of two STSs (Definition 11) is given by the synchronous product (SP) of the automata where complementary actions (shared actions) are synchronized in the sense that they are considered to be synchronously executed.

Definition 11 (STS Composition) *For $P = \langle S_1, A_1, \Delta_1, s_1\rangle$ and $Q = \langle S_2, A_2, \Delta_2, s_2\rangle$, the composition of P and Q is represented by the synchronous product of the two STSs. $P\|Q = \langle S_1 \times S_2, A_1 \cup A_2, \Delta, (s_1, s_2)\rangle$ where Δ is the smallest relation satisfying the rules:*

$$\frac{P \xrightarrow{[g]a} P'}{P\|Q \xrightarrow{[g]a} P'\|Q}\ a \notin A_2,\quad \frac{Q \xrightarrow{[g]a} Q'}{P\|Q \xrightarrow{[g]a} P\|Q'}\ a \notin A_1,\quad \frac{P \xrightarrow{[g_1]a_1} P',\ Q \xrightarrow{[g_2]a_2} Q'}{P\|Q \xrightarrow{[g_3]\tau} P'\|\ Q'}$$

where g_3 is the logical conjunction between g_1 and g_2 and
the name(a_1) = name(a_2) and signature(a_1) = signature (a_2) and
a_1, a_2 denote complementary actions (when a_1 denotes a message receipt, a_2 denote a message emission and vice-versa).

In the CwSTS model, complementary actions (shared actions to be synchronized) are always in pair: message emission and message receipt. Consequently, the synchronous product denotes that the message emission on one component takes place in the same logical time as the message receipt (synchronous communication semantics). Note that in the actual composite component interface, τ transitions are not presented as they have no relevance for the interaction of the composite component with its environment. Also, as already specified, message receipt on one component does not automatically represent message execution (execution of the corresponding operations on receiver side).

Definition 11 defines the SP of two protocols. However, due to our model specificity (1 to 1 service connection) the generalization to the general case (composition of n protocols) is straightforward. Another consequence of the binary communication specificity is that one action in an STS will be shared with at most one other complementary action in another component at a specific type.

5.3.4 Component Substitutability

In this subsection we consider the conditions under which a component can be successfully replaced by another component. The substitutability usually defines a refinement notion on the types of entities we consider (functions, objects, components).

Substitutability was already addressed in all the area of mathematics and computer science. A function with a parameter of type `T` (defined as `fun f (x : T) : Integer`) can be replaced by a function `g` (defined as `fun g (x : S) : Integer`) if `T` $\leq$ `S`. In other words, if `g` cares less about the type of its parameter, then it can replace `f` anywhere, since both return an `Integer`. We say that the type of `g` parameter is a *generalization* of the type of `f` parameter or that the type of `f` parameter is a *specialization* of the type of `g` parameter. So, in a language accepting function arguments, `g` $\leq$ `f` and the type of the

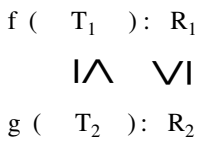


Figure 5.11: Function refinement type relationships.

parameter to `f` is said to be contravariant. While the type of the parameter of `g` is said to be contravariant, the type of the return value is said to be covariant (as the return type of `g` $\geq$ the return type of `f`). Figure 5.11 depicts the case where a function `g` can successfully replace a function `f` because of the covariance and contravariance type relationships.

Objectual programming languages like Java consider the substitution of an object with another one if there is a relation of subtyping (usually by inheritance) from the class of the second one to the class of the first one. Design by Contract principles also play an important role when taken into consideration. According to the aforementioned principles, a subtype can only have weaker pre-conditions and stronger post-conditions than its base class.

Component substitutability is more difficult to define as it must consider containing both the required and provided services and its behavior (i.e. interaction protocol). In the followings, we analyze the structural and behavioral substitutability in CwSTS component model.

5.3.4.1 Structural substitutability

In CwSTS the structural interface is composed of the set of all provided and required services. If we consider the set of signatures of all provided services as the input type of a component and the set of signatures of all required services as the output type we can make an analogy with the function refinement presented above.

The intuition we have in a case of a component refinement indicates that a component can replace another component at the structural interface level if it provides at least what the replaced component provided and requires at most what the replaced component required. By this we assure that a new component will provide what the first one provided or plus and that it requires less or equal as the initial one. We are here in the classical contravariant/covariant relationship between the types of the two components, considering that we call a component input type the set of signatures of all its provided services and the return type the set of signatures of all its required services.

5.3.4.2 Behavioral substitutability

Behavioral refinement implies that a refined component must behave as explained below. If we consider a trace semantics for our STS based protocols, a component protocol must accept at least the same traces (inputs and outputs) as the substituted component protocol. In addition, the component protocol must refuse at most as much requests as the initial component protocol. Nierstratz [Nierstrasz 93] define this kind of substitutability notion

in function of traces and failures. While traces define the set of executed actions, failures define the requests that a component cannot accept after a specific trace was executed.

5.3.4.3 Practical component substitution

In a practical case we can see a component refinement following two different contexts. If we want to substitute a component in isolation, the rules presented above must be applied. If the substituted component is already in an architecture, adaptors must be created in order to adapt the interface and the protocol of the newly created component to the rest of the architecture. Adaptors are essential here as we imposed the strong restriction services can not be interconnected while their name and signature are not the same. This limitation can be relaxed in a future evolution of our model definition and implementation. For the instant, adaptors are most likely to be created when replacing a component already present in an architecture.

Lets reconsider our guideline example presented in Figure 5.10 at page 122. In this case we were obliged to use an *Adaptor* component in order to connect the *Client* and *Server* component as the client protocol was not compatible with the server protocol (see Figure 5.3 at page 116 and Figure 5.7 at page 119 for the *Client* and *Server* protocols).

If we consider substituting the *Client* component with another client component that do not require the login and logout behavior before actually subscribing to a specific topic, the new client component will fit in the architecture without any problem and without the use of an *Adaptor* component.

5.4 CwSTS-Interface Description Language

In this section we present CwSTS-IDL an interface definition language that we propose in order to describe both primitive and composite CwSTS component interface (both structural and behavioral).

In the followings, we present conceptual details of CwSTS-IDL. Starting by an abstract grammar presented in Figure 5.12, we discuss the main characteristics of this language. We exemplify each important section, like component interface definition, composite architecture and protocol specification, by presenting examples driven from our guideline example. These examples are written in a concrete language that we actually use to develop component-based architectures.

5.4.1 Primitive Components

Conforming to the rules specified by the abstract grammar presented in Figure 5.12, a classical component definition contains the name of the component type and in its description two important parts: one defining the structural interface and the other one defining the interaction protocol associated with the component. The structural interface is given in terms of operation names and signatures, each operation represents a provided or a required. The protocol description is given as a SFSP process definition (see Section 5.4.3 at page 131).

```
component_def        ::= primitive_def | composite_def
primitive_def        ::= component_type
                         primitive_strct_def
                         guards_def
                         primitive_protocol
primitive_struct_def ::= service_def+
service_def          ::= service_type op_id formal_parameter_list? | op_id formal_parameter_list?
guards_def           ::= op_id formal_parameter_list?
service_type         ::= 'required' | 'provided'
composite_def        ::= component_type
                         subcomponent_decl+
                         composite_structural_def?
                         connection_def+
subcomponent_decl    ::= component_type subcomponent_id
composite_struct_def ::= service_def+
connection_def       ::= bind_exp | export_exp | import_exp
bind_exp             ::= subcomponent_op_id 'to' subcomponent_op_id
export_exp           ::= subcomponent_op_id 'as' op_id
import_exp           ::= op_id 'to' subcomponent_op_id

subcomponent_op_id   ::= subcomponent_id '.' op_id
```

Figure 5.12: CwSTS Abstract EBNF Grammar.

Following the concrete language that we employ to describe components, the definition of the Client component type in our guideline reservation system is presented in Figure 5.13.

5.4.1.1 Component Interface Definition

Structural interface is specified in the *interface* section of the component definition. Service definitions contain the *provided* or *required* modifier in order to indicate that this service is provided or required by the component. In addition the service name and signature is specified. The signature contains no return type as our model only considers one direction message transmission. The formal parameter list can omit parameter names as the only important think is the parameter type.

5.4.1.2 Guard Definition

Guards, boolean operations guarding component transitions are defined in the *guards* section of the primitive component definition. In order to define a guard only the name and the formal parameter list is required to be specified. Guard definitions are to be used in the protocol description of a component as shown in Figure 5.13.

5.4.2 Composite Components

A composite CwSTS component is given in terms of subcomponent instances having their services either bound among themselves or exported/imported at the composite interface

```
primitive component Client {
    interface
        provided messageEvent(Message msg);
        required login();
        required logout();
        required subscribe(Topic topic);
        required unsubscribe();

    guards
        loggedIn();

    protocol
        P=!login -> P | !logout[!loggedIn()] -> P | !subscribe[!loggedIn()] -> Q,
        Q= ?messageEvent -> Q | !unsubscribe() -> P.
}
```

Figure 5.13: Client Component Definition.

level (see Figure 5.14). When describing a composite component, we start by declaring subcomponent variables of an already defined component type definition. Individual operations of each declared subcomponent are connected in the architecture by following any of the three composition schemas (i.e. bind, export and import) in the CwSTS model. Additionally, the structural interface of the composite must also be defined when operations need to be exported or imported. The interaction protocol of the composite (actually, the synchronous product of all the subcomponent protocol) is not provided in the declaration as it can be automatically computed by a tool in CwSTS-P.

5.4.2.1 Composite Subcomponents

Naturally, we follow an incremental process in the hierarchical component definitions. We start by defining primitive components and then define incremental architectures based on already existing definitions (either of primitive or of composite ones).

Subcomponents are defined inside the *subcomponents* section of a composite component type. They are given under the form of a component type name followed by the subcomponent identifier. Subcomponent identifiers are farther used in the *interface*, *protocol* and *connect* sections of the composite definition.

5.4.2.2 Composite Interface Definition

The composite component interface is constituted of the services exported or imported by its subcomponents. In Figure 5.14, section *interface* of the SystemReservation composite component contains the code lines defining the export and import of specific subcomponent services. Those services are automatically becoming the services required and provided by the composite component. In case when services with different name or signature are required to be specified by the composite component, a regular service definition, as specified in Figure 5.13 (definition of a primitive component) is allowed. Notice that we

```
1   composite component StatisticsEnabledServer {
2       subcomponents
3           Server server;
4           StatisticsReporter reporter;
5
6       interface
7           export server.subscribe() as subscribe();
8           export server.unsubscribe() as unsubscribe();
9           import messageEvent() to server.messageEvent();
10
11      protocol
12          ...
13      connect
14          server.statisticsReq() to reporter.statisticsReq();
15          reporter.statisticsReport() to server.statisticsReport();
16  }
```

Figure 5.14: StatisticsEnabled composite component definition.

can not directly export or import a service to/as another service without the same name and signature. In order to cope with this situation an adaptor component must be used as for the case of connecting incompatible subcomponents.

5.4.2.3 Subcomponents Interconnection

The actual architecture of interconnected subcomponents is specified in the *connect* section of the composite component definition. As shown in Figure 5.14, the syntax specifies the service of a component instance that is to be connected to the service of another component instance.

5.4.3 Symbolic Finite State Processes (SFSP) a process algebra for STSs

In order to describe component protocols we developed a processes definition language we baptize SFSP. Symbolic Finite State Processus is inspired from the FSP (Finite State Processus) [Magee 99] language but keeps only the transition, choice and recursion. In addition, actions are parameterized with types parameters in SFSP. SFSP is a very intuitive language and its semantics is easy to understand. Figure 5.15 depicts the abstract grammar for SFSP. A concrete language is actually used to describe CwSTS protocols (see Annexes for the concrete grammar of SFSP).

5.4.3.1 Primitive Component Protocol Definition

The primitive component protocol is defined in the *protocol* section of the component definition (see Figure 5.13 at page 130). As a concrete example lets consider the protocol definition of the Client component in the case study example. Figure 5.16 depicts the protocol description. This textual description is identical to the graphical description presented in Figure 5.3 at page 116 but allows for tool analysis and checking.

```
specification       ::= process_def+
process_def         ::= process_id formal_param_list process_body

process_body        ::= choice+
choice              ::= action+ process_inst
process_inst        ::= process_id actual_param_list | special_process

special_process     ::= STOP

action              ::= receive_act | send_act | internal_act
receive_act         ::= action_id formal_param_list
send_act            ::= action_id actual_param_list
internal_act        ::= action_id actual_param_list

formal_param_list   ::= param_type param_id | formal_param_list | ε
actual_param_list   ::= param_id | actual_param_list | ε
```

Figure 5.15: SFSP Abstract Grammar.

```
P=!login -> P | !logout[!loggedIn()] -> P | !subscribe[!loggedIn()] -> Q,
Q= ?messageEvent -> Q | !unsubscribe() -> P.
```

Figure 5.16: Client SFSP definition.

A protocol description always starts with the definition of the P process as the entry point in the execution of the component protocol. The P protocol represents the totality of the component protocol. Other processes described at this level (like Q process for example) are viewed as subprotocols of the global protocol defined by P.

A process can contain choice (the | operator) and the process definition can be recursive (as the definition of the P process).

A transition specification contains the identifier of the action to be taken (`subscribe` for example), the provided or the required identifier (`!` or `?`) and the guard identifier (`loggedIn()`).

5.4.3.2 Composite Component Protocol Definition

The composite component protocol description can be automatically generated as it represents the synchronous product of the subcomponent protocols (the parallel execution of the corresponding STS protocols).

5.4.3.3 SFSP To STS Correspondence

In the followings we present some formal definitions showing the relationship between SFSP and STS, the STOP processus, transition, choice, recursion, and finally the semantics of SFSP composition.

Definition 12 (SFSP to STSs correspondence) *The correspondence is defined by the function:*

$$sts:\ Exp \rightarrow \gamma$$

where Exp is the set of SFSP process expressions and γ the set of STSs. The function sts is defined inductively on the structure of the SFSP process expressions.

Definition 13 (STOP) *the special SFSP process :*

$$sts(STOP) = \langle \{s\}, \{\}, \{\}, s \rangle$$

Definition 14 (Transition $\rightarrow$) *If $sts(E) = \langle S, A, \Delta, q \rangle$ then $sts([g]a \rightarrow E) = \langle S \cup \{p\}, A \cup \{(g,a)\}, \Delta \cup \{(p,(g,a),q)\}, p \rangle$ where $p \notin S$,g represents a parameterized guard action and a represents a parameterized action.*

Definition 15 (Choice |) *Let $1 \leq i \leq n$, and $sts(E_i) = \langle S_i, A_i, \Delta_i, q_i \rangle$ then:*

$sts([g_1]a_1 \rightarrow E_1|...|[g_n]a_n \rightarrow E_n) =$

$$\langle S \cup \{p\}, A \cup \{a_1...a_n\}, \Delta \cup \{(p,(g_1,a_1),q_1)...(p,(g_n,a_n),q_n)\}, p \rangle,$$

where $p \notin S_i$, $S = \cup_i S_i$, $A = \cup_i A_i$, $\Delta = \cup_i \Delta_i$.

Definition 16 (Recursion) *We represent the SFSP process defined by the recursive equation $X = E$ as $rec(X = E)$, where X is a variable in E. The process defined by the recursive definition $X = ([g]a \rightarrow X)$ is represented as $rec(X = ([g]a \rightarrow X))$. We use $X[X \leftarrow rec(X = E)]$ to denote the SFSP expression that is obtained by substituting $rec(X = E)$ for X in E. Then, $sts(rec(X = E))$ is the smallest STS that satisfies the following rule:*

$$\frac{sts(E[X \leftarrow rec(X=E)]) \xrightarrow{[g]a} P}{sts(rec(X=E)) \xrightarrow{[g]a} P}$$

Intuitively, any action inferred by the expression E unwound once can also be inferred by the process represented by the recursive definition. Mutually recursive equations can be reduced to the simple form described above.

Definition 17 (SFSP Composition) $sts(Q_1||Q_2) = sts(Q_1) \parallel sts(Q_2)$.

5.5 Conclusion

Component models and languages presented in the state-of-the-art (see Chapter 3 at page 57) integrate too many concepts or are too specific. In this context it was very difficult to analyze the consequences of adding new concepts like the interaction protocols and code generation in a fully fledged component model.

CwSTS (acronym for *Component with Symbolic Transition System*) is a simple, yet general component model integrating only the minimal features required in order to analyze

the integration of interaction protocols at component interface level. CwSTS features black-box components, communicating exclusively through one declared interface. The interface consists of two parts: a *structural interface* and a *behavioral interface*. The structural interface describes the required and provided component service signatures and corresponds to the first level of contracts in the taxonomy of Beugnard and al [Beugnard 99] (see Section 3.2.1.2 at page 61). The behavioral interface describes the component interaction protocol (the rules governing the component in term of message emission and receipt). It corresponds to the third level of contracts in the aforementioned taxonomy.

Interaction protocols are expressed as Symbolic Transition Systems (STSs) where transitions are composed of actions to be executed at message receipt or actions resulting in message emission and guards (boolean operations with parameters) that condition the transition from the initial state to its final state. STS based formalisms offer advantages over the classical LTSs (Labelled Transition Systems). Namely, STSs cope better with state explosion problem, are more expressive and are more likely to be adopted by ordinary component developers.

Composition is hierarchical in the sense of GoF Design Pattern (a composite can be further on composed with other components into a larger architecture) but in addition to the classical structural composition our model also exhibits behavioral composition.

In order to describe components and their protocols we propose both a graphical and a textual language. CwSTS-IDL is a concrete textual language employed to describe component interfaces, architecture (component composition) and protocol. As part of CwSTS-IDL, SFSP (Symbolic Finite State Processus) is a process description language inspired from FSP (Finite State Processus) where we keep only the transition, choice and recursion. SFSP is a very intuitive language and its semantics is easy to understand. In addition, previous experience with FSP (a suit of tools available at http://www.doc.ic.ac.uk/ltsa/) could be extended in order to allow the visualization and verification of behavioral STS architectures.

CwSTS characteristics are cornered out to allow a generative approach when constructing components. We target two utilization scenarios (see Section 5.1.1 at page 111) each of them requiring automatic code generation starting from component type definitions.

We voluntarily restricted the set of features of the CwSTS model (only one interface, communication is 1 to 1, only static architectures) in order to simplify things and easily analyse the integration of interaction protocols at the level of component interfaces. However, the model can be extended in order to provide communication multiplicity for example. Future work related to our component model is suggested in Section 7.1 at page 163 and take into consideration the extension of CwSTS in order to include 1 to many communication, multiple interfaces, dynamic configurations, etc.

The next chapter presents the prototype implementation of our model we baptize CwSTS-P.

Chapter 6

CwSTS Implementation

In this chapter we present CwSTS-P, the prototype implementation of our component model in the Java programming language. We give details on how we achieve the implementation of primitive and composite components that conform to the model specification.

Contents

6.1 Component Implementation

6.1.1 Introduction

In this chapter we present CwSTS-P the prototype implementation of our component model. We use the Java programming language to implement the components with all

their structural entities and the synchronization mechanism involved in the component composition. We also provide tools to implement the component compatibility and substitution and architecture completeness algorithms. Finally, component generator tools are also proposed in order to automatically generate component code. Our generators address only the code related to the component structure and interaction with its environment. The component business code is not generated in our approach and remains up to the programmer to provide it by either of the two approaches indicated in Section 5.1.1 at page 111. Adaptors, a special case of components, can be automatically and fully generated as their business behavior is completely determined by the interaction context.

As specified in Chapter 5 at page 110, the CwSTS model defines encapsulated black-box components communicating exclusively through declared interfaces. A component has only one business interface (defining both provided and required service) and behaves as defined by its declared interaction protocol. The interaction protocol defines the valid sequences of messages exchange between the component and its environment.

Each primitive component also comes equipped with a mechanism for composition into larger structures called architectures or composite components. A composite component does not provide additional business logic code other than that contained in its subcomponents. Composition is also hierarchical, in the sense that a composite components can be further on composed (integrated) into larger architecture.

In the following subsections we present the details on the implementation of primitive and composite CwSTS components in the Java programming language. Important details include how the packaging of a CwSTS component is realized in Java and how the structural and behavioral composition is achieved.

6.1.2 Java Packages and Component Entities

In CwSTS-P, each component entity is packaged as a classical Java package containing the component implementation classes. Figure 6.1 presents the structure of this package where constituent classes and interfaces are structured in sub-packages.

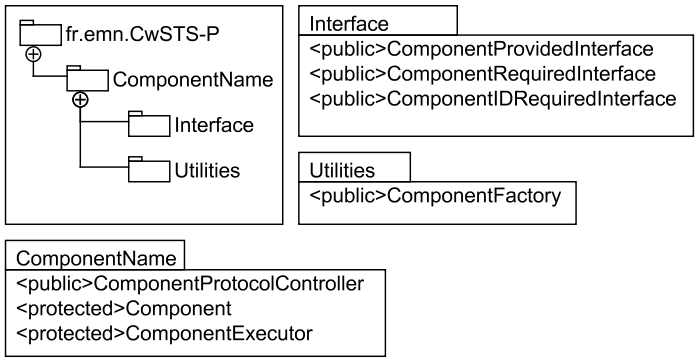


Figure 6.1: Component Package in Java.

The name of the package is the name of the component during its development and packaging phases of its lifetime. The name conforms to a defined standard which is:

fr.emn.CwSTS-P.ComponentName. The root of this package contains the classes (see Section 6.1.3) that will be actually instantiated to obtain a runnable component entity. The two subpackages *Interface* and *Utilities* contain the component declared interfaces and a component factory class, respectively. While interfaces define the component provided and required services, the factory class is used at instantiation time to obtain a new instance of this component type.

Our component model defines encapsulated black-box components. This means that the component internals are not visible and cannot be interacted with both before and after the instantiation time. The component implementation code is not visible to parties that are not given explicitly access and at execution time, the only possible interaction with the component instance is through the explicitly defined interfaces. In order to cope with these requirements, once created and tested, a component package is compressed as a jar [1] file. Further on, the only *public* entities in the component package are the component interfaces, the protocol controller (which represents the component front end) and the component specific factory class. The other classes inside the component are declared as *protected* ensuring that, once the development phase is over, the component could be accessed exclusively through the controller object (actually implementing all the component declared interfaces). Details on component structure are given in the next subsections.

6.1.3 Primitive Component Implementation

In this section we present the standard structure of a component, the details of each entity composing a component and the relationships (both structural and behavioral) between them.

6.1.3.1 Component Structure

A primitive CwSTS-P component is constituted of three entity types: one or more business logic implementation classes, a component executor class and a protocol controller class. Figure 6.2 presents a partial view of the different entities interaction, that is, the functional interaction view, where entities cooperate in order to provide the components implemented functionality.

Business Logic Entity

The business logic entity represents the code actually realizing the component functionality. As a consequence, it implements the declared component provided interface. Figure 6.3 depicts a use case class diagram where a component *CType* entities constituents implement different interfaces needed in order to form a valid CwSTS component. The class *fr.emn.CWSTS-P.CType.C* represents the components business logic entity implementing the components provided interface (*fr.emn.CWSTS-P.CType.CProvidedInterface*). Besides the functional interface, this entity also implements a configuration interface (allowing to configure it before execution) and a lifecycle interface (allowing to control the start and stop events).

1. Java Archive

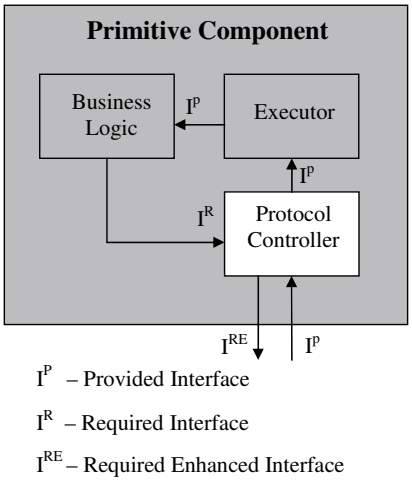


Figure 6.2: Primitive Component Structure with a Partial View Over the Interactions.

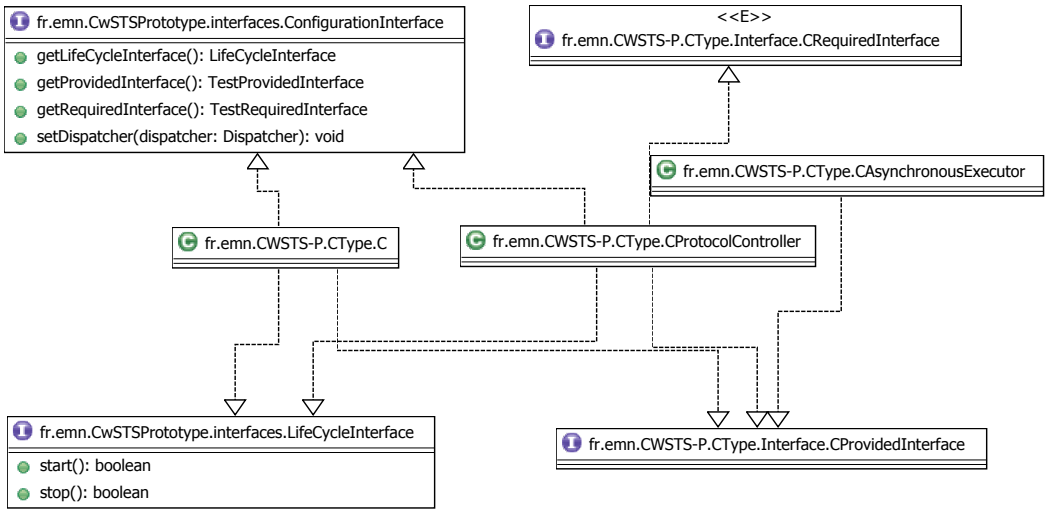


Figure 6.3: Primitive Component Class Diagram.

Component Executor Entity

The component executor entity acts as a proxy in behalf of the business logic entity. Its main purpose is to intercept the threads calling the methods of the business logic entity and to proceed with the call but in a different thread (see Figure 6.4). This is useful to implement our model requirement that we assure messages receipt in a synchronous manner and that we do no assumption on when/how these messages will be consumed (corresponding methods on business logic entity actually executed).

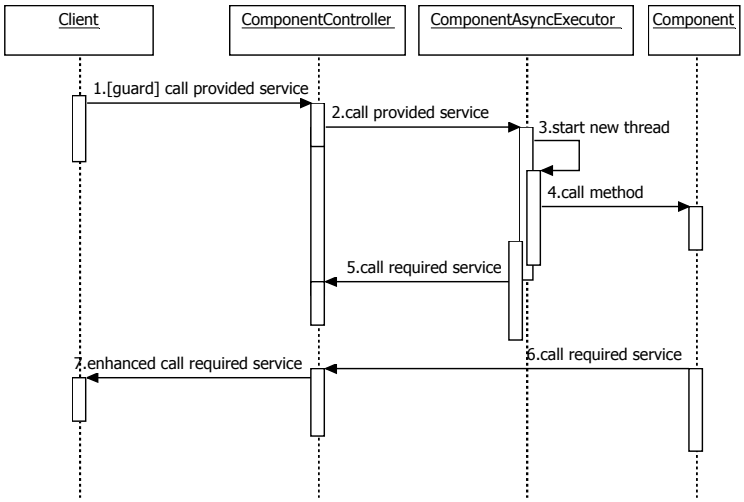


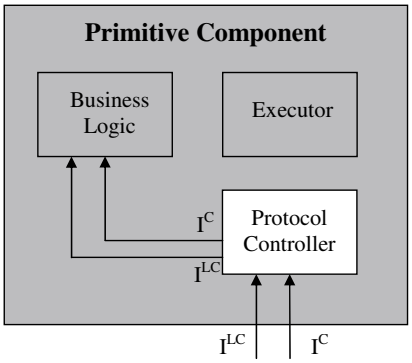
Figure 6.4: Primitive Component Internals.

Component Protocol Controller

The component protocol controller has as its main purpose the control of component inbound and outbound calls and this according to the component declared protocol. The protocol specifies the allowed sequence of service calling (either targeting or leaving the component). The controller assures that call order will occur exactly as specified by the protocol. Details on how the protocol is "executed" and "controlled" are given in Section 6.2 at page 146 where we also explain how the behavioral synchronization (message exchange synchronization) is achieved in our prototype implementation. In order to comply with the composition mechanism, the controller implements the components provided interface but also a slightly enhanced version of the required interface (see Section 6.1.4.1 at page 141, the dispatcher explanation).

6.1.3.2 Component Configuration, Instantiation and Execution

Figure 6.5 depicts the configuration view over the relationships between the different entities of a primitive component. Different phases are taken before the component is actually starting execution (see sequence diagram in Figure 6.6). A new instance of the business logic entity (*Component*), of the component executor (*ComponentAsyncExec*) and of the controller entity (*ComponentController*) are created. Different settings are done at the



I^{LC} - Life Cycle Interface
I^{C} - Configuration Interface

Figure 6.5: Component Structure.

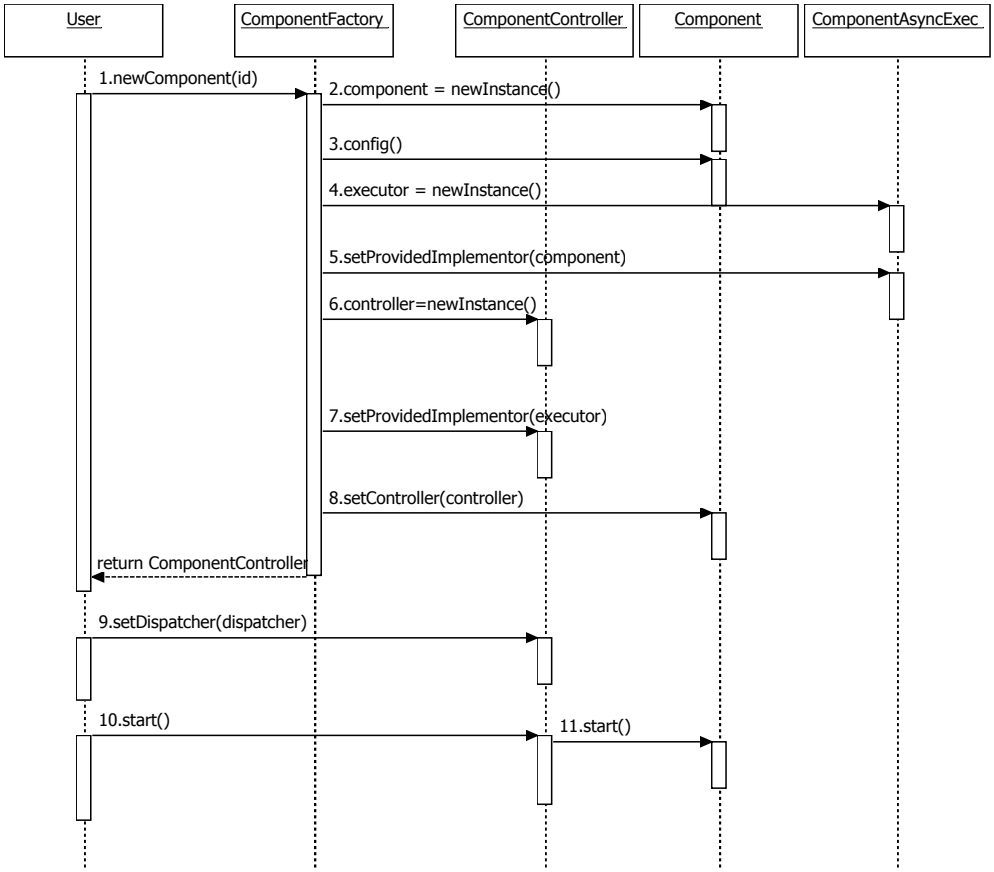


Figure 6.6: Primitive Component Instance Creation.

level of the three entities instances. Finally, the reference to the component controller is returned to the client that can further on set the components environmental information (the composite dispatcher as explained later) and then start the component execution (by calling the `start()` method which is propagated to the business logic entity).

Once started, for each inbound service call, the component behaves as depicted in Figure 6.4. The guard in line 1 of the sequence diagram states that the call is received under some conditions (see explanations on implementing the distributed synchronization in Section at page) at the level of the *ComponentController* entity. In case of a required service call (issued by the component itself), the business logic entity's call is just forwarded by the component controller after enhancing it with some identification information in order for the composite to reroute correctly this call towards the right subcomponent in the architecture.

6.1.4 Architectures Implementation

6.1.4.1 Composite Component Structure

The structure of a composite component is depicted in Figure 6.7 at page 142. Unlike a primitive component, a composite one does not contain a business logic entity. Instead, the functionality of a composite component is given at execution time by the interaction of the containing primitive components. One composite can contain one or more components (either primitives, in the sense described in Section 6.1.3, or composite components, as the composition is hierarchical) also designed as subcomponents. Besides the set of subcomponents, a composite also contains a controller entity (similar to the one in the primitive component structure) and a dispatcher entity. The dispatcher is implementing a call forward strategy that realizes at run time the structural relationships between components. In the following, we present the composite component entities in more details and how they differ from the entities of a row primitive component.

Subcomponent

A composite component is composed of two or many subcomponents. Originally subcomponents can be row primitive components or other composite components (as composition is hierarchical). As shown in Figure 6.7 at page 142, subcomponents are not directly functionally interconnected. Instead, they are all connected to the centralized dispatcher entity. As already showed in Section 6.1.3 each row primitive component implements the specific provided interface and a slightly enhanced required interface in order to correctly connect to the composite dispatcher.

Dispatcher

The dispatcher is responsible for routing calls coming from subcomponents to their exact destination subcomponents. In this purpose, the dispatcher integrates a routing table implementing the structure of the composite component as defined in its specification. In order to implement its purpose (that of routing calls between subcomponents) the dispatcher needs to know the source of a service call. As depicted in Figure 6.7, the

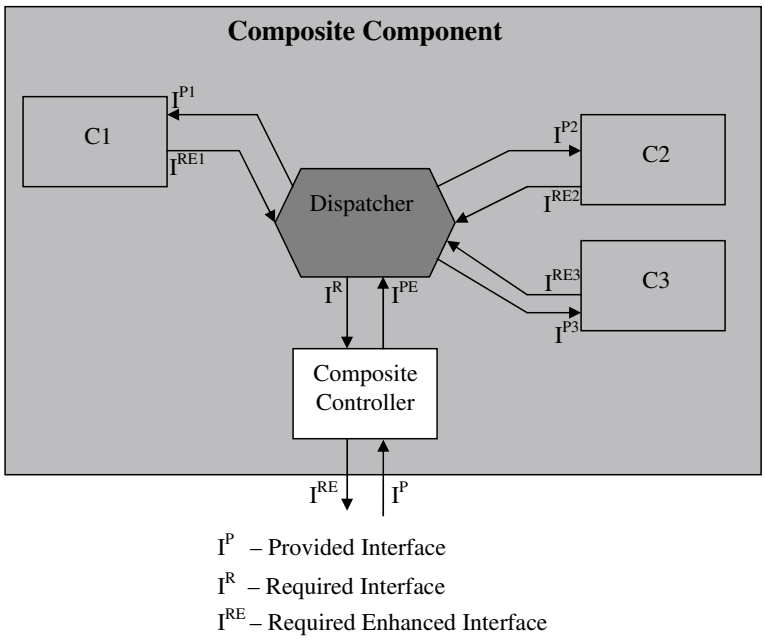


Figure 6.7: Composite Functional Structure.

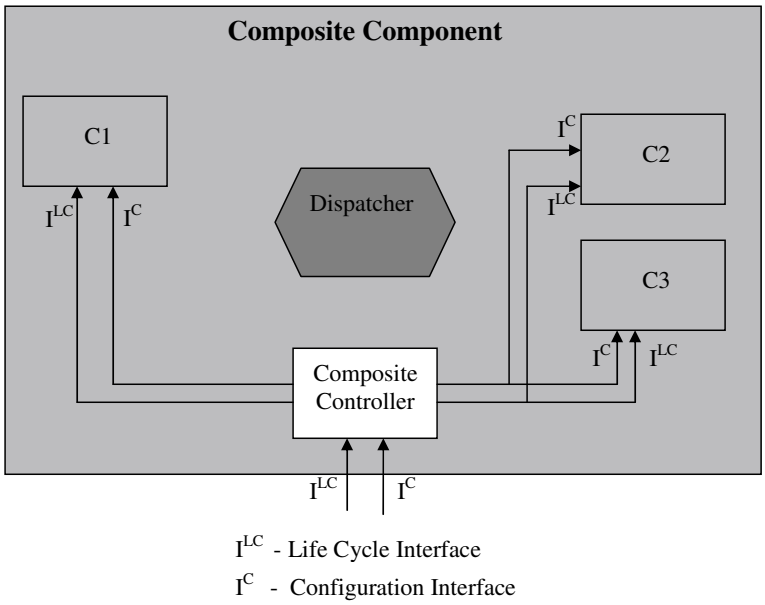


Figure 6.8: Composite Control Structure.

dispatcher implements all the required interfaces of the composite subcomponents. As the dispatcher needs to know the source of a call, these interfaces are all enhanced with an single parameter which is the *id* of the subcomponent that issued the call.

Composite Controller

The composite controller represents, as in the case of a primitive component, the composites front end. All calls issued or targeting the composite component pass through it. In this purpose, the controller implements the composite provided and required interfaces. But, unlike the primitive controller, no execution protocol checking is realized. In fact, when composing compatible components in an architecture, the execution of all the components realize the synchronous product of the set of all protocols in the architecture. The control is needed only at row primitive component levels to ensure the synchronization is correctly realized.

The package of the composite has the same structure as the primitive one. One additional requirement is that the required packages containing the subcomponent classes are in the classpath at runtime. If the application is to be deployed as a single unit, an additional */lib* directory entry can be specified in the composite package. This directory contains the classes of all of the subcomponents of the architecture application.

6.1.4.2 Architecture Configuration, Instantiation and Execution

Figure 6.8 depicts the configuration view over the interaction of all the composite entities and subcomponents. The phase of configuration is interlaced between the two actions of component creation and component execution start showed in Figure 6.9. The user of a composite component sets the data needed by the composite for execution. The composite controller charges itself to propagate required data to subcomponents. Usually, the kind of data to be set is the value of initialization parameters. Other data, like for example the dispatcher reference is set into the subcomponents transparently for the user of the composite component. Other data that do not depend on composite external information is the structural links between subcomponents. Figure 6.9 shows the composite instantiation sub-phases, where subcomponents are created (by the bias of specific component factories), the dispatcher re-routing table is configured and set into subcomponents and where the components are started.

Once configured and started, inbound and outbound composite calls are treated as in the case study presented in Figure 6.10. Calls to the composite cut down to the composite controller entity which forwards them to the dispatcher entity. Based on the identity of the caller (either the subcomponents or the controller itself) and the re-routing table, the dispatcher decides to which entity the call will be forwarded. Calls issued by the composite component follow a similar path.

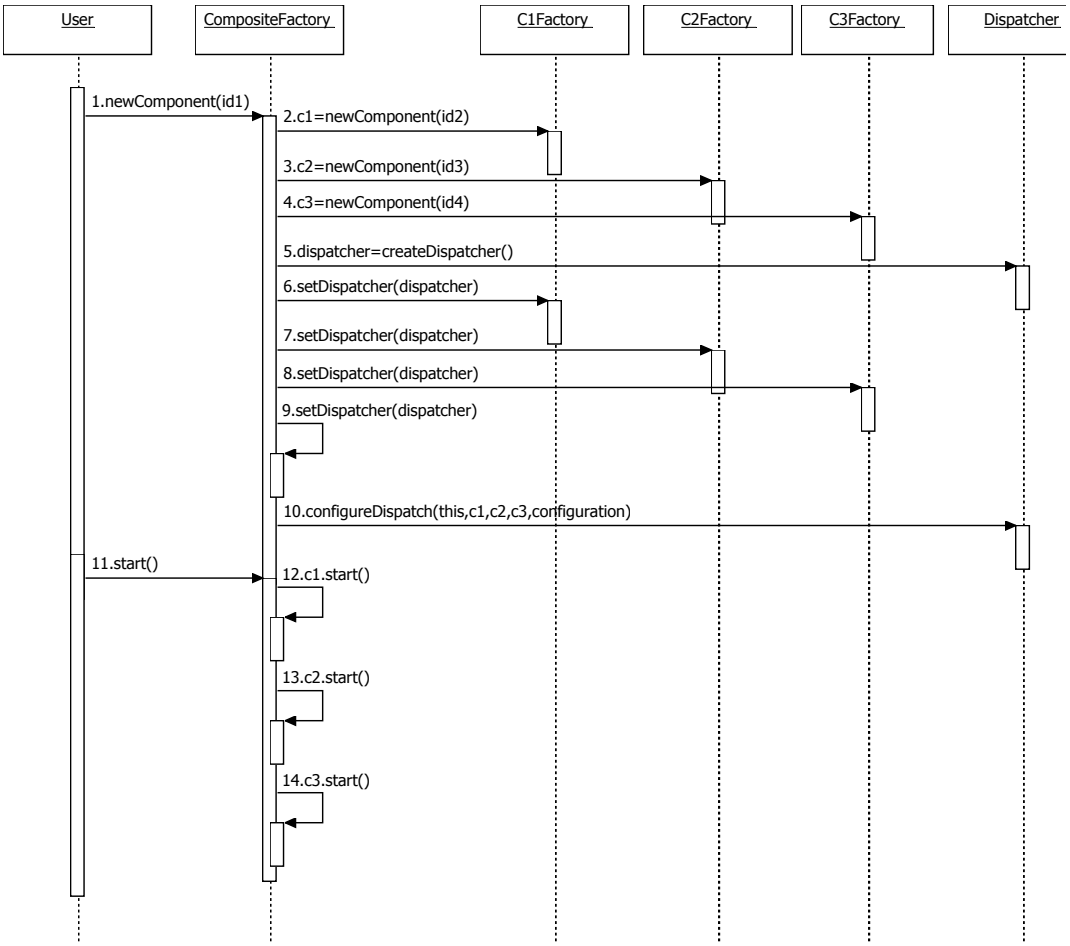


Figure 6.9: Composite Instance Creation.

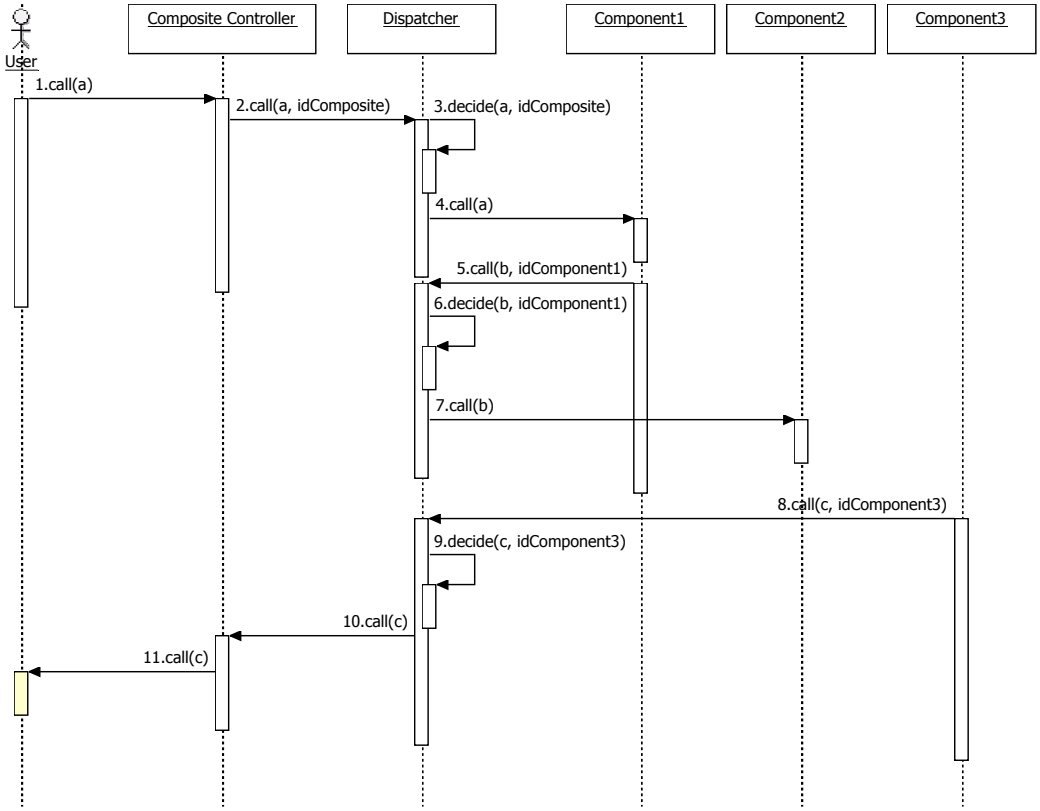


Figure 6.10: Composite Execution.

6.2 Behavioral Composition Implementation

The synchronous product of the set of interaction protocols associated to components describe the global behavior of the system obtained after the composition. The semantics of the synchronous product stipulate that shared actions (i.e. one message emission and one message receipt) are seen as executing in the same time. That means that the corresponding transitions (in individual component protocols) are considered as a unitary transition in the composition synchronous product protocol.

Conversely, in an implementation, the action of sending a message and that of receiving the corresponding message must be unitary executed as if they were in a transaction. More precisely, in order to obtain the synchronous product behavior at execution time, each emission/receipt pair actions must begin and end in the same (absolute) time. But this is hard to realize as components are individual entities executing in parallel (possibly, on different machines) and without any global clock in order to synchronize actions.

Regarding the implementation of the synchronous product realization, we identified two different approaches: the distributed and the centralized synchronization mechanisms. The distributed synchronization mechanism is our first experimentation and consists in encapsulating and deploying the code for action synchronization at the component level. This seems to be the ideal solution when in a distributed environment as it maximizes the parallelism in the execution of the global system. The current prototype proposal is based on this approach and the component structure presented in the previous sections is targeted to integrate in the distributed synchronization mechanism. However, this solution also suffers from an important limitation: the *mixed states* problem as identified in [Yellin 97] is not effectively addressed. Mixes states are explained in detail in Section 6.2.3 at page 151.

In order to cope with this important situation, we explored a second solution where there is a unique centralized *arbiter* entity that guides components in the shared actions synchronization. Using such an entity avoids the problem of mixed states but also represents a non efficient solution especially in a distributed context with a large number of interacting components.

It is obvious that a pure distributed solution can not successfully consider the case of mixed states and a form of centralized synchronization must intervene in order to resolve this special situation. Future work must consider the combination of the two approaches with the objective of dealing with the mixed states situation in a distributed approach. In the following, we will start by presenting the general mechanism associated with each solution and then show how we can integrate each solution into our implementation of CwSTS.

Lets remember that we consider only 1 to 1 one way communication and that we place our components in both a structural and behavioral hierarchical composition. Components are seen as autonomous active entities executing their own protocol. This means that the externally visible actions of a component are decided in function of its protocol. A specific message can be sent only when the execution of the component has arrived in a state where the specific message can be emitted. This is also true for message receptions. At execution time, a component independently decides on one particular send action to perform from the set of actions that are specified by its protocol in its current execution state. The two

following approaches decline these requirements.

6.2.1 Distributed Synchronization Mechanism

The purpose of this approach is to propose a completely decentralized solution to the problem of synchronizing executing components. This mechanism we propose allows a maximum of parallelism in the execution of the components especially when components are deployed in a distributed environment. In addition, the extra communication needed for coordination (synchronization) purposes is reduced to zero as it is completely encapsulated in the regular communication between components.

This mechanism relies on the following principles:

- each component is executing its protocol (in terms of emissions, receipts, internal actions and guard evaluation) and no unspecified behavior is issued,
- each component plays the role of a client (for the component sending a message) and the role of a server (for the component receiving a message),
- while playing the role of a server, each component manages a queue of incoming messages that are to be treated by the component,
- each component is, at any specific time, executing any of the following operations:
  1. execute an internal action (as specified by the protocol)
  2. involve in an emission phase (as specified by the protocol)
  3. treat a waiting message (to be received) from the queue
- when in an emission phase, a component can not execute any other operation while the message is not acknowledged by the other party, in other words it is blocked while waiting for the message receipt,
- when a change occurs in its logical state, each component is verifying if any message is waiting in the queue and if so proceed as follows:
  1. take one waiting message (a non deterministic choice) and test if that specific message can be receipt in the current logical state of the component,
  2. if the message can be receipt in the current state, test if the guard associated with the transition evaluates to true,
  3. if the guard evaluation returns true, acknowledge the receipt and proceed with another operation,
  4. if the guard or the state tests fail, leave the message in the queue and proceed with another operation (often, another queue message treatment).
- when receiving an acknowledgment for the message is sending, the component involved in an emission phase unblocks and proceed with its execution.

This mechanism is close to the approach of data-driven coordination (see Section 3.3.7.1 at page 90) where it is the data (the message in our case) that determines the coordination (of message emission and its receipt).

It is a simple mechanism assuring that each component will finally behave as described by the protocol and that the global observed behavior of the system is identical to that of the synchronous product. This is realized due to the fact that each sending component is blocked while the other party is not acknowledging the receipt. The queue of waiting messages is a structure that allows the component to execute normally (in concordance with its protocol) and to pool for waiting messages only when it is ready to treat them.

6.2.2 Distributed Synchronization Mechanism Integration in CwSTS-P

The integration of the distributed synchronization mechanism in CwSTS-P was realized by using the support for multithreading in the Java programming language. Much of this support centers on *thread synchronization* which is coordinating activities and data access among multiple threads. The mechanism that Java uses to support synchronization is the *monitor*.

A monitor is a special entity associated with any java object that can be occupied by only one thread at a time. The entity usually contains some data. From the time a thread enters this entity to the time it leaves, it has exclusive access to any data associated with the entity and also to the object methods the monitor is attached to.

Related to our component synchronization mechanism, the monitor assures that only one thread will execute some code at the component level at any given moment in time. In order to achieve the correct communication synchronization, we need to impose the right conditions on the thread synchronization.

The actual implementation details assuring the component synchronization are given bellow:

- incoming and outgoing calls are encapsulating message receiving and sending, respectively. Incoming calls are carried out by threads generated by the other parties. As the component is an active entity (one or multiple threads are executing the component business code), outgoing calls are carried out by the thread (or threads) generated inside the component itself.

- when inside the monitor, each incoming call thread execute a specific method implementing the message receipt. This method is actually implementing the message receive operation related to a transition specified in the component protocol. The method integrates the control related to the logic state of the component and the test of the guard (restricting the transition) if the component is in the right state to receive the corresponding message. The method also implement an indirection towards the actual implementation of the business logic code related to the specific message.

- when exiting the monitor, the active thread signals the JVM[2] to awake a waiting

2. Java Virtual Machine

thread to enter the monitor.

- the JVM awakes a waiting thread based on a non deterministic algorithm and as the monitor is no longer occupied, the newly activated thread is free to execute the method actually implementing the message receipt.
- the thread carrying out an outgoing call is constraint in the same way as the incoming call threads. The outgoing call can be issued only if the thread owns the monitor and if the logical state of the components allows that specific message.

```
public class ComponentController {
  //Protocol field definition.
   ...
  //Protocol current state execution field definition .
   ...
  //Private boolean guard implementing methods definition.
  ...

  public synchronized void aMessage(Object...parameters) {
      /* Do not allow the thread to continue until the condition is OK.
         The wait() method call allows for another thread to enter the
          monitor.
       */
      while (!condition) wait();

     /* Proceed with a call to the component Executor entity.
        The message was RECEIVED!
      */
       ...

    /* The current thread finished its work and as the message was
       received, the method return in order to unblock the caller.
       But before ... wake up other  threads waiting to enter the
       monitor after the current thread quits.
     */
       notifyAll();

  }
}
```

Figure 6.11: ComponentController Java Class Excerpt.

The success of this implementation resides in the fact that the monitor is allowing only one waiting thread to access to the component methods actually implementing the message receipt. Also, the fact that the methods implementing receipts integrate the tests for the logical state and the guard evaluation assure that the thread will actually succeed in executing the receiving method code only if the conditions required to do so are accomplished. If not, the thread is put in a waiting state (that of waiting to enter the monitor) expecting that the required conditions will be met. Figure 6.11 presents an

excerpt of the implementation of the `ComponentController` class (the class implementing the component controller entity as described in Section 6.1.3 at page 137).

The class integrates a protocol structure to represent the component communication protocol and an internal indicator to keep the current state in the execution of the protocol. Boolean operations representing the protocol transition guards are also defined inside this class. The method `aMessage(Object...parameters)` represent a message receipt method treatment. This method is declared as `synchronized` meaning that only one thread can access it at any given moment in time. The method is also `public` and returns `void` thus conforming to the specification of our model where the clients access the component by well defined access points (the public methods of the `ComponentController` entity) and communication is only one way (no return on method calls).

Once inside the method, a thread will test for conditions. These conditions are represented by the logical state in which the component is present and that must allow for the specific message receipt and the guard evaluation, if any. While these conditions are not satisfied, the thread is put in waiting state (the `wait()` method call). Next time the thread is awaken, and if the conditions are satisfied, the message receipt method can be successfully executed. This execution carry out a call to the component `executor` entity which decouples the receipt from the actual execution of the message by the component business code.

Before exiting the method execution, the thread will update the current protocol logical state to the state specified to be reached after the receipt of this specific message. The last line in the method definition is `notifyAll()` which will awake all the threads waiting to enter the component monitor allowing thus to proceed with execution once the current thread finished. When the thread exits the method, a reply is issued to the caller (the component client) thus unblocking it from waiting that the message is received at the callee side.

The current implementation specifies that each `ComponentController` class implements as many public synchronized methods as there are provided services specified by the component protocol. The fact that they are all declared as `synchronized` assure that only one thread will be able to execute any of these methods at any moment in time.

Required services are implemented at the `ComponentController` class level as `protected`, `synchronized` methods. They are called by the component business logic code implementation and are only implementing the same controls as the methods treating message receipts.

Thus, incoming and outgoing threads are synchronized in their execution by the monitor attached to the `ComponentController` entity. The monitor assure that only one thread is executing the corresponding message receipt or emission method and control inside the methods assure that the execution will proceed only when the conditions required to do so are satisfied (the component execution is in the right state to execute that action and the associated guard, if any, evaluates to true) .

The use of `wait`, `notify` and `notifyAll` represent a rather basic approach to implementing thread synchronization in the Java programming language. Better synchronization structures are available starting from the release of Java 5. Thus, `Condition` factors out the monitor methods (`wait`, `notify` and `notifyAll`) into distinct objects to give the effect

of having multiple wait-sets per object, by combining them with the use of arbitrary `Lock` implementations. Where a `Lock` replaces the use of synchronized methods and statements, a `Condition` replaces the use of the monitor methods.

The use of `Condition` and `Lock` objects allow for a better implementation of our mechanism as, for example, we are capable of awakening threads that are waiting for a specific condition rather than awake all waiting threads of a monitor even if they do not have the chance to enter it as their conditions are not satisfied.

6.2.3 Distributed Synchronization Mechanism Evaluation

In the context of a large number of distributed interacting components the distributed synchronization mechanism represents the real solution. There is no extra overhead related to the synchronization as it is the communication coordination realized at the receivers side that determines the component interaction synchronization.

This mechanism is close to the approach of data-driven coordination (see Section 3.3.7.1 at page 90) where it is the data (the message in our case) that determines the coordination (of message emission and its receipt).

As each interaction is synchronized at the receiver level, the distributed mechanism allows for a better parallelism in the execution of the global system as independent actions can take place without the need for a global entity to decide.

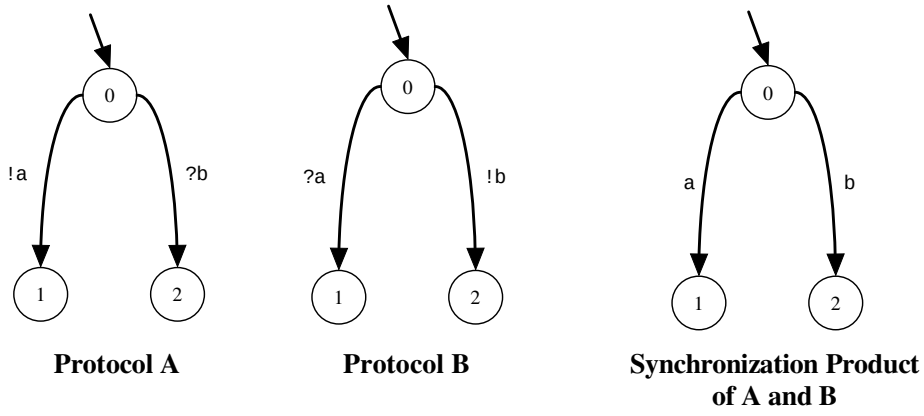


Figure 6.12: Mixed State Situation.

One drawback of our current implementation is that the current implementation does not explicitly consider a special case known under the name of *mixed states* [Yellin 97]. Figure 6.12 depicts such a case where two components (A and B) each of them executing its own protocol (Protocol A and Protocol B respectively) are in a state where a receipt or an emission is possible. For this special case, our current implementation does not guarantee that the interaction of the two protocol is deadlock free. For a better understanding, lets assume that the two components initiate a sending in *the same time*. According to our implementation, the sender is blocked until the receiver acknowledges the receipt but in this case the two components will be blocked as each of them is in a sending phase that

do not allow for a receipt. This behavior is completely non deterministic and depends uniquely on the thread management at the JVM level.

In order to cope with this limitation we explored a second solution: the centralized synchronization mechanism. This mechanism, even if not actually implemented in our CwSTS prototype, is presented in the next subsections as it represents the base for a solution to the mixed states situation.

6.2.4 Centralized Synchronization Mechanism

This solution implies the existence of an *Arbiter* entity that helps components synchronize on executing actions. The role of the Arbiter is that of a monitor actually observing the messages that are going to be exchanged among the components and that based on its local information decides the beginning of specific pair actions (message emission/receipt) on different components.

Each component is an active entity (i.e. running thread) executing its own protocol and that before actually executing a specific action (either a send or receive) in its protocol, must ask the Arbiter to allow the action execution. Figure 6.13 depicts this situation where all components in the system are linked to a unique centralized entity (the arbiter).

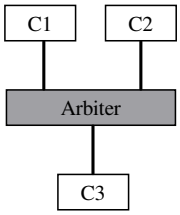


Figure 6.13: Centralized Arbiter Entity

This situation implies that each component in the system has knowledge of the arbiter entity. Additionally, the arbiter entity knows all the components and also their associated actions (that they need to synchronize on).

The actual mechanism that allows the Arbiter to synchronize actions on components is based on a two synchronization barrier mechanism. Before each action (a send or a receive) to be performed, the concerned component sends a *syncOn* message to the Arbiter. The Arbiter *blocks* those calls while the conditions required to proceed are not fulfilled. These conditions typically concern the presence of calls from both a sender and a receiver for the same message. In other words, the Arbiter allows components to proceed only when one sender notified the Arbiter (by a blocking syncOn call) that it is ready for the specified action and that a receiver (or the specified number of receivers in case of group communication) has done the same thing for the complementary action. At this moment, the Arbiter unblocks the syncOn calls of all concerned components. Once released, components proceed with their action (the sender will actually send the message to the receiver(s)) and the receiver will wait to be called. Once the message was sent and received, each component will sent a *syncOff* message to the Arbiter. This is identical with the syncOn message, but while syncOn helps components synchronize on the beginning of the message transfer, syncOff helps them synchronize on the end of message transfer. The second synchronization barrier is of major importance because it assures that components finish their message transfer phase in the same time and before proceeding with another different action.

Figure 6.15 depicts the sequence diagram of a possible execution of an architecture based on three components C1, C2 and C3 and an Arbiter entity. Each component executes

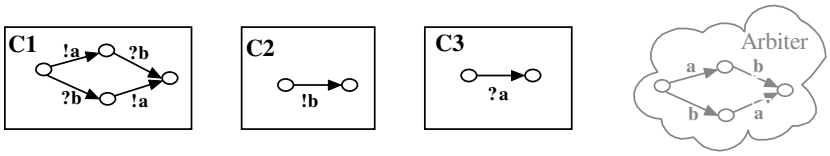


Figure 6.14: Component Protocols.

its own protocol and the Arbiter must assure the behavior described by the synchronous product (see Figure 6.14).

One important remark is that the Arbiter is not knowing (nor executing) the synchronous product protocol of the component system. However, at execution, with the information related to components, the resulting behavior is conforming to that specified by the synchronous product. This is also possible because each component follows the two synchronization barrier mechanism either for send or for receive actions.

6.2.5 Centralized Synchronization Mechanism Integration in CwSTS-P

The centralized synchronization can be realized by seeing the *synchronization* as a subsidiary service (as for example security, transactions and persistence in commercial component models). Components, firstly, register themselves to the Arbiter entity by sending it the information needed in order to provide the synchronization service. The Arbiter is itself a server playing the role of a system monitor (see Figure 6.16). It monitors the actions that are going to be executed by the connected components and, based on its local information, at any moment in time, decides what components can proceed with their actions such that the global behavior of the system is followed by communicating components. After the register phase, components start executing by following a two synchronization barrier protocol identical to that explained in the preceding subsection. However, the actual communication is realized directly among components (see Figure 6.16). The Arbiter is only deciding which action (and so, which components must execute) is to be realized at any specific moment in time.

As the synchronization service is transparent if seen from an architectural point of view, the hierarchical composition property of the CwSTS model is straightforward in the provided implementation. Composites themselves, at the configuration phase, will recursively instruct subcomponents to register on the Arbiter. Note, that only primitive components (those that are really providing business code) will actually register on the Arbiter as communication is generated and finally targeted to them.

In CwSTS, a transition can be realized only if the associated guard (if any) evaluates to true. Due to the fact that guards are implemented as private operations (only internal access) and can take parameters (that will be known only when a message arrive), the Arbiter can not decide on what transition to execute based on guard evaluation. This must be done once the Arbiter decided on the transition to execute and when a component actually receives a message. At this time, however, the guard evaluation can return false thus resulting in a thread deadlock, as the message can not be receipt while the guard is

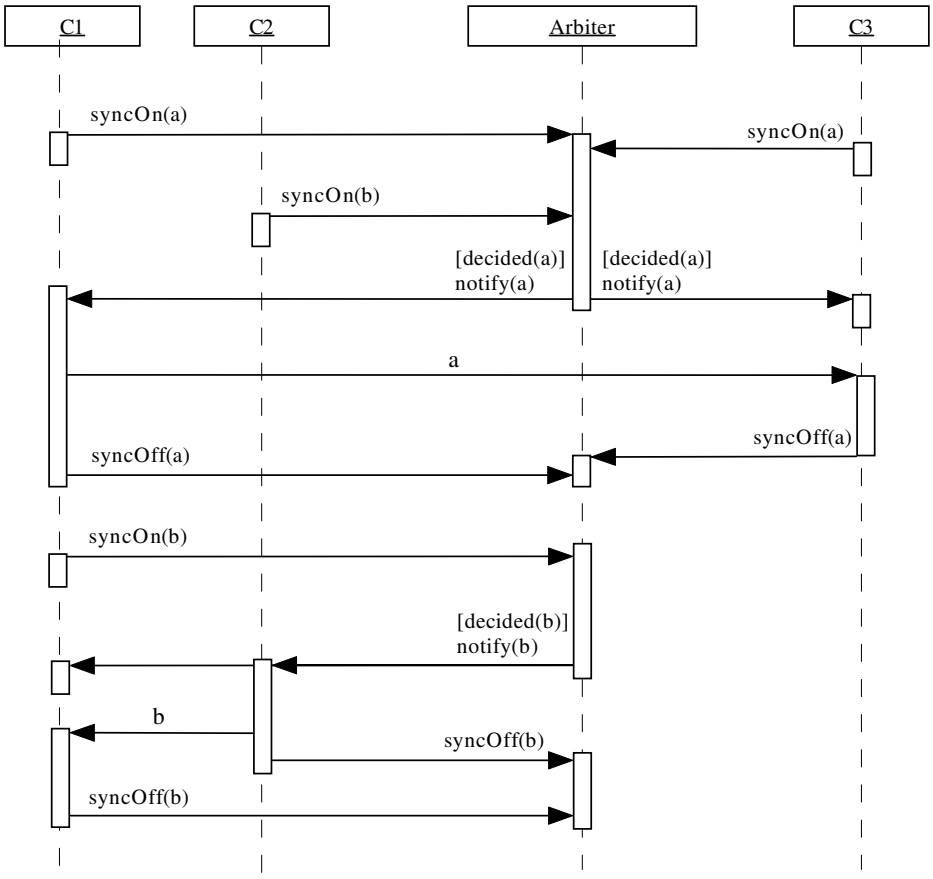


Figure 6.15: Synchronization Mechanism in Action.

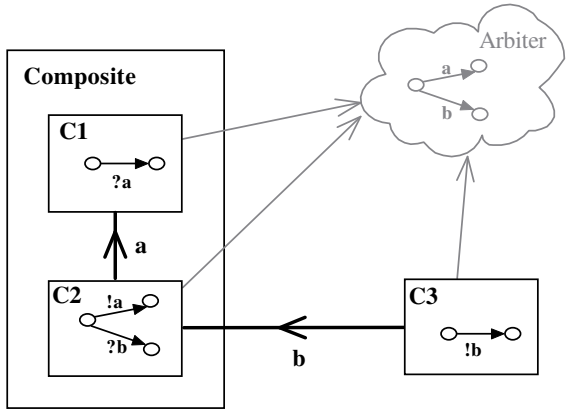


Figure 6.16: Arbiter Implementation.

not true.

6.2.6 Centralized Synchronization Mechanism Evaluation

Applying a two synchronization barrier mechanism in order to synchronize complementary actions on different components is a very natural solution. The centralized monitor entity (the arbiter) is the masterpiece of such a solution as it actually coordinates communicating components in order to achieve synchronization. The arbiter is not executing any protocol (i.e. the synchronous product), but based on the architecture topology (i.e. the number of components and the actions to be executed by each component) it restricts the global behavior of the system in order to exhibit the behavior specified by the synchronous product of the components interaction protocols. This solution also relies on the idea that each component, prior to actually executing a specific action (an emission or a reception, according to its protocol), asks the Arbiter for permission. Only after the Arbiter allows this action the component can proceed with its execution.

The idea of having the components ask permission to "receive" a specific message might not seam natural, especially at the implementation level. But if we remove this constraint from the synchronization mechanism, the Arbiter will have to know the synchronous product protocol of the component architecture. Otherwise, it cannot be capable of deciding the right moment when a specific message can be safely sent and received. Moreover, in such a case we would have to find a solution to remove the second synchronization barrier (the one synchronizing on the end of the message transfer) and guarantee the realization of the synchronous product by using only one synchronization barrier.

This mechanism is close to the ones used in control-driven coordination (see Section 3.3.7.2 at page 91) where the actual coordination of entities is realized by employing coordination signals (syncOn and syncOff messages on Arbiter in our case).

What makes the force of this mechanism (the centralized arbiter) also represents its disadvantage. In a distributed environment it is not always possible nor convenient to insert a centralized monitor. The coordination of an increasing number of components can cause loss of information and bottleneck, with corresponding increase of the response time of the centralized arbiter. Furthermore, as each component is an autonomous entity in communication with different partners, we would like to maximize parallelism when possible. Applying the two synchronization barrier mechanism with a centralized arbiter forces the global execution of the components to be totally sequenced.

6.3 Code Generation

We follow a generative approach when constructing components. We actually take into consideration two scenarios (see Section 5.1.1 at page 111) each of them requiring the generation of java code in order to obtain valid CwSTS components. The main objectives we pursuit when employing code generation are the conformance between the component descriptions and the implementation code and the automation of the creation of non business and adaptation code. Conformance between the component description and its implementation code is obtained by construction. It is the generation process that assures that final

code will behave as specified.

From an informal point of view, the generator engine uses previously developed templates that are going to be customized in order to obtain final code. As already specified, we do address the creation of business code. Business code must fit into the set of the entities that represent the component implementation code.

In the first scenario, the business code is created after the generation of the component outer layer (the set of interfaces and the entities presented in Sections 6.1.3 and 6.1.4). It is the developer responsibility to create the business logic class (or classes) that exhibit the right behavior in term of emissions and receipts and to implement the communication type (synchronous or asynchronous) regarding the actual execution of the received messages.

In the second scenario, the business code is already present before the generation process. Thus, the generator wraps the existing code in order to interface it with the generated outer layer. In order to realize that, the generator is provided with information regarding the existing business code that must be wrapped.

The component generation process comprises four stages. Starting from each component description and some additional information, the generation engine:

- generates the Java interfaces and packages (as presented in the beginning of this chapter) that represent a CwSTS component in Java,
- generates the `Protocol Controller` entity (the one responsible to assure component synchronization with its environment),
- generates the `Executor` entity (synchronous or asynchronous),
- plugs in the business logic implementation.

Adaptors are seen as first class components in our model. This means that they have the same internal structure as any other component in the architecture. The difference comes from the fact that their interface and behavior is employed to connect two or more components that can not directly connect due to their incompatibility. To generate adaptors rather then business components, the generation engine is given the description of the adaptor component and the information needed in order to adapt provided services to required ones (the signature transformation for example). Based on this information, the generation engine will automatically generate an executor (synchronous by default) and a by default business logic implementation. The business logic is only forwarding messages received to a provided service to the corresponding required service after processing some transformations as instructed.

The current state in the development of the prototype consider only a limited set of featured related to the generator capabilities. Thus, we do not offer tools that analyze the existing business code (in the case of the second scenario) or tools to check that the business code created in the first scenario approach can safely connected to the generated code. However, this can be addressed in a future evolution of the prototype where some other important improvements (see the Perspectives section in the next chapter) can be realized.

6.4 Conclusion

In this chapter we presented details on CwSTS-P, the prototype implementation of our component model. We use the Java programming language to implement the components with all their structural entities and the synchronization mechanism involved in the component composition.

In CwSTS-P, each component entity is packaged as a classical Java package containing the component implementation classes. Our component model defines encapsulated black-box components. This means that the component internals are not visible and cannot be interacted with both before and after the instantiation time. The component implementation code is not visible to parties that are not given explicitly access and at execution time, the only possible interaction with the component instance is through the explicitly defined interfaces. In order to cope with these requirements, once created and tested, a component package is compressed as a jar[3] file. Further on, the only *public* entities in the component package are the component interfaces, the protocol controller (which represents the component front end) and the component specific factory class. The other classes inside the component are declared as *protected* ensuring that, once the development phase is over, the component could be accessed exclusively through the controller object (actually implementing all the component declared interfaces).

A primitive CwSTS-P component is constituted of three entity types: one or more business logic implementation classes, a component executor class and a protocol controller class. The business logic entity represents the code actually realizing the component functionality. The component executor entity acts as a proxy in behalf of the business logic entity. Its main purpose is to intercept the threads calling the methods of the business logic entity and to proceed with the call but in a different thread. The component protocol controller has as its main purpose the control of component inbound and outbound calls and this according to the component declared protocol. The protocol specifies the allowed sequence of service calling (either targeting or leaving the component). The controller assures that call order will occur exactly as specified by the protocol.

Unlike a primitive component, a composite one does not contain a business logic entity. Instead, the functionality of a composite component is given at execution time by the interaction of the containing primitive components. One composite can contain one or more components (either primitives, in the sense described in Section 6.1.3, or composite components, as the composition is hierarchical) also designed as subcomponents. Besides the set of subcomponents, a composite also contains a controller entity (similar to the one in the primitive component structure) and a dispatcher entity. The dispatcher is implementing a call forward strategy that realizes at run time the structural relationships between components. In the following, we present the composite component entities in more details and how they differ from the entities of a row primitive component.

Different configuration phases are taken before the components are actually starting execution. At a primitive component level, new instances of the business logic entity, of the component executor and of the controller entity are created. Different settings are

3. Java Archive

done at the level of the three entities instances. Finally, the reference to the component controller is returned to the client that can further on set the components environmental information and then start the component execution (by calling the start() method which is propagated to the business logic entity). At a composite component level, the user sets the data needed by the composite for execution. The composite controller charges itself to propagate required data to subcomponents. Usually, the kind of data to be set is the value of initialisation parameters. Other data, like for example the dispatcher reference is set into the subcomponents transparently for the user of the composite component. Other data that do not depend on composite external information is the structural links between subcomponents.

The synchronous product of the set of interaction protocols associated to components describe the global behaviour of the system obtained after the composition. The semantics of the synchronous product stipulates that shared actions (i.e. one message emission and one message receipt) are seen as executing in the same time. That means that the corresponding transitions (in individual component protocols) are considered as a unitary transition in the composition synchronous product protocol.

Conversely, in an implementation, the action of sending a message and that of receiving the corresponding message must be unitary executed as if they were in a transaction. More precisely, in order to obtain the synchronous product behaviour at execution time, each emission/receipt pair actions must begin and end in the same (absolute) time. But this is hard to realise as components are individual entities executing in parallel (possibly, on different machines) and without any global clock in order to synchronise actions.

We presented two implementation solutions to the synchronous product realization problem. The distributed synchronization mechanism is our first experimentation and consists in encapsulating and deploying the code for action synchronization at the component level. This seems to be the ideal solution when in a distributed environment as it maximizes the parallelism in the execution of the global system. The current prototype proposal is based on this approach and the component structure presented in this chapter is targeted to integrate in the distributed synchronization mechanism. However, this solution also suffers from an important limitation: the *mixed states* problem as identified in [Yellin 97] is not effectively addressed.

In order to cope with this important situation, we explored a second solution where there is a unique centralized *arbiter* entity that guides components in the shared actions synchronization. Using such an entity avoids the problem of mixed states but also represents a non efficient solution especially in a distributed context with a large number of interacting components.

It is obvious that a pure distributed solution can not successfully consider the case of mixed states and a form of centralized synchronization must intervene in order to resolve this special situation. Future work must consider the combination or the extension of the two approaches with the objective of dealing with the mixed states situation in a distributed approach.

Chapter 7

Conclusions and Perspectives

In this thesis we considered the design and implementation of a model defining black-box software components that integrate interaction protocols at the interface level. The interaction protocol represents an interaction contract between a component and its environment and a special formalism based on Symbolic Transition Systems is employed in order to facilitate the description of complex interactions. The implementation follows a generative approach where components execution code is generated from high level descriptions of their structural and behavioral interfaces.

The notion of software component was firstly mentioned by McIlroy [McIlroy 68] in his speech to the OTAN conference in 1968 and the idea of software reutilization emerged as a solution to the problem encountered in the development of increasingly larger and complex software applications. The CBSE paradigm considers the software component as the hearth of any software application that need to meet high quality standards in its design, implementation, deployment and execution. The CBSE approach encompasses developments in component models definition, Software Architectures, Architecture Description Languages (ADLs) [Medvidovic 00a], Component-Oriented Programming, etc.

Particular component characteristics are described by models (component models). A component model describes what a component is (by defining its constituent parts) and says how components can be eventually composed (by following some composition rules). It also describes the component life cycle and the roles associated with different actors in the development and exploitation of applications. There are two categories of component models: academic and industrial.

The Software Architecture [Garlan 94, Bass 98, Garlan 94] is the organization of a software system as a collection of components, connections between the components, and constraints on how the components interact. When using Software Architectures, one systematically targets to: reduce the development cost, improve the quality in terms of reliability, maintainability and resource efficiency, reduce time-to-market and reduce maintenance costs.

The ADLs represent one of the results of the research done in the field of Software Architectures. Their objectives is to aid application architects structure and compose their software parts in order to obtain valid applications. An ADL is defined as a formal

or informal notation, either textual or graphical, allowing to specify software architectures and that are accompanied with specific tools [Medvidovic 00a]. The result of using an ADL is mostly an abstract view over an application in terms of *components*, *connectors* and *configurations*, rather than a concrete view over the implementation details.

While architectural analysis in existing ADLs may reveal important architectural properties, these properties are not guaranteed to hold in the implementation. In order to enable architectural reasoning about an implementation, the implementation must conform to its architecture. Luckham and Vera [Luckham 95a] identify three criteria for architectural conformance: decomposition, interface conformance and communication integrity. In order to deal with these three requirements, a new approach was born, that is programming with components. A new class of programming languages integrating architectural abstractions into a general-purposed language like Java. Examples include ArchJava [Aldrich 02a, Aldrich 02b], Java/A [Baumeister 06] and Jiazzi [McDirmid 01].

A new class of models, formalisms and mechanisms has evolved for describing concurrent and distributed computations based on the concept of "coordination" (a concept by no means limited to Computer Science). The purpose of a coordination model and associated language is to provide a means of integrating a number of possibly heterogeneous components together, by interfacing with each component in such a way that the collective set forms a single application that can execute on and take advantage of parallel and distributed systems. Configuration and architectural description are closely related to coordination. Configuration and architecture description languages share the same principles with coordination languages. From a slightly liberal point of view, one can include configuration and ADLs in the category of coordination languages.

In order to successfully interact, components need to conform to a certain form of interaction contract (interaction protocol based on the notion of behavior protocol). Interaction protocols describe the entity behavior in terms of sequences of messages exchanged between a component and its environment.

The term *process algebra* refers to a family of specification techniques particularly well suited to describing systems of concurrent communicating components. Process algebras describe the process interactions in terms of calculus (an ensemble of rules defined on a set of process expression construction operators). There are many process algebra formalisms issued from the research field. The most important ones are CCS [Milner 89] and CSP [Hoare 85] as they stand at the base of other algebras like the PI-calculus [Milner 92, Milner 99] and the mobile agents [Cardelli 98], integrating primitives for the expression of distribution and mobility.

Behavior models are precise, abstract descriptions of the intended behavior of a system. Behavior models have solid mathematical foundations that can be used to support rigorous analysis and mathematical verification of properties. Effective techniques and tools have been developed for this purpose and have shown that behavior modeling and analysis are successful in uncovering the subtle errors that can appear when designing concurrent and distributed systems.

Much of the work on developing type-theoretic foundations for programming languages has its roots in typed lambda calculus. In such approaches, an instance of a type is viewed as a record of functions together with a hidden representation type [Cardelli 85]. We talk

about *service types* when referring to such an instance as it only describes the signatures of the services an entity (either object or component) provides to or requires from its communication partners.

More recent work ([Nierstrasz 93, Puntigam 96]) advocate for the extension of classical service types in order to allow the description of dynamic properties of objects (and also components) and their composition. This kind of types are called *behavioral types* as they specify not only a set of messages to be exchanged between entities in order to communicate but also constraints on acceptable sequences of these messages.

State machine based formalisms are generally assumed to be complete descriptions of system behavior at some level of abstraction. From a component modeling perspective, the system behavior is what an external entity can observe about the system's interaction with its environment. This is usually the messages the system (black-box component) exchanges with its environment in terms of emissions and receipts. A finite state machine is describing the set of all possible traces a component can produce when interacting with its partners. Labelled Transition Systems (LTS) [Keller 76] represent one of the FSM based formalisms employed in order to describe process behavior. An LTS consists of a finite set of states and a corresponding set of transitions between states. LTS suffer from a very important drawback when it comes to model checking: the well known state explosion problem. When considering the synchronized product of two or many LTSs, the potential number of states (an transitions) can be enormous. Model checking tools used to analyze the configuration can be hardly used or fail in this context In order to deal with these issue, STSs extend LTSs by using symbolic transitions. Unlike LTSs, in STSs transitions describe *classes* of possible operations to be effectively executed. One transition defines an operation that can be parameterized with input and out parameters. In addition a transition can be guarded allowing the execution of the operation only if the condition of the guard is true. Where an LTS explicitly describes all the traces that can be realized at execution, an STS abstracts on the possible traces. Thus, an STS description is much more readable, compact and expressive than classical LTS.

I/O Automata provide an appropriate model for discrete event systems consisting of concurrently-operating components. I/O automata may be nondeterministic and this is an important characteristic of the model's descriptive power. Describing algorithms as nondeterministically as possible tends to make results about the algorithms quite general. The I/O automata model is especially helpful when describing the interfaces between system components and it provides a clean composition model for fair composition. Although I/O automata can be used to model synchronous systems, they are best suited for modeling systems in which the components operate asynchronously.

The main critics that can be done regarding some of these formalisms is the abstraction level, much too weaker to describe higher level entities like components. Some other drawback is the fact that some important properties like deadlock are not decidable. Some other formalisms sacrifice expressiveness in order to increase decidability. Developing object models, important notions like the substitutability and refinement made their appearance, easing the way to developing satisfactory component models integrating protocols.

In component models, interaction protocols usually specify component externally visible behavior. The specification is done at interface level and enhances purely structural

interfaces with behavioral descriptions. Practically, non of the presented component models consider the coherence between the specification and implementation. Excepting for CwEP (see Section 4.3.2 at page 102), the other component models have a rather evasive approach when considering implementation issues (see Section 4.3 at page 101). However, CwEP is only flat hierarchical component model and structural hierarchical composition is not considered at all.

It is obvious from the state-of-the-art that it misses a component model integrating a form of interaction protocol that is readable, expressive and easy-to-use by an average component designer. The implementation has to be part of the main concerns when developing the component model as the coherence between specification and implementation is essential when applying the model in a real software development. Generative techniques, either by using an MDA [1] approach or a rather empirical one, represent an appealing research field when considering the component implementation.

The result of our research work is CwSTS. CwSTS (acronym for *Component with Symbolic Transition System*) is a simple, yet general component model integrating only the minimal features required in order to analyze the integration of interaction protocols at component interface level. CwSTS features black-box components, communicating exclusively through one declared interface. The interface consists of two parts: a *structural interface* and a *behavioral interface*. The structural interface describes the required and provided component service signatures and corresponds to the first level of contracts in the taxonomy of Beugnard and al [Beugnard 99] (see Section 3.2.1.2 at page 61). The behavioral interface describes the component interaction protocol (the rules governing the component in term of message emission and receipt). It corresponds to the third level of contracts in the aforementioned taxonomy.

Interaction protocols are expressed as Symbolic Transition Systems (STSs) where transitions are composed of actions to be executed at message receipt or actions resulting in message emission and guards (boolean operations with parameters) that condition the transition from the initial state to its final state. STS based formalisms offer advantages over the classical LTSs (Labelled Transition Systems). Namely, STSs cope better with state explosion problem and are more expressive.

Composition is hierarchical in the sense of GoF Design Pattern (a composite can be further on composed with other components into a larger architecture) but in addition to the classical structural composition our model also exhibits behavioral composition.

In order to describe components and their protocols we propose both a graphical and a textual language. CwSTS-IDL is a concrete textual language employed to describe component interfaces, architecture (component composition) and protocol. As part of CwSTS-IDL, SFSP (Symbolic Finite State Processus) is a process description language inspired from FSP (Finite State Processus) where we keep only the transition, choice and recursion. SFSP is a very intuitive language and its semantics is easy to understand. In addition, previous experience with FSP (a suit of tools available at http://www.doc.ic.ac.uk/ltsa/) could be extended in order to allow the visualization and verification of behavioral STS architectures.

1. Model Driven Architecture

CwSTS characteristics are cornered out to allow a generative approach when constructing components. We presented two implementation solutions to the synchronous product realization problem. The distributed synchronization mechanism is our first experimentation and consists in encapsulating and deploying the code for action synchronization at the component level. This seems to be the ideal solution when in a distributed environment as it maximizes the parallelism in the execution of the global system. The current prototype proposal is based on this approach and the component structure presented in this chapter is targeted to integrate in the distributed synchronization mechanism. However, this solution also suffers from an important limitation: the *mixed states* problem as identified in [Yellin 97] is not effectively addressed.

In order to cope with this important situation, we explored a second solution where there is a unique centralized *arbiter* entity that guides components in the shared actions synchronization. Using such an entity avoids the problem of mixed states but also represents a non efficient solution especially in a distributed context with a large number of interacting components.

It is obvious that a pure distributed solution can not successfully consider the case of mixed states and a form of centralized synchronization must intervene in order to resolve this special situation. Future work must consider the combination or the extension of the two approaches with the objective of dealing with the mixed states situation in a distributed approach.

7.1 Perspectives

We have chosen a generative approach when constructing CwSTS components. This is in line with the actual trend imposed by the MDA approach to constructing software applications. Code generation allows for a faster and safer software development then a classical approach and can be integrated with sophisticated verification and analysis techniques. The tools we propose in our CwSTS prototype represent the base modules required in order to construct powerful generators dealing with interaction protocols described under the Symbolic Transition System formalism.

The actual CwSTS model considers solely static configurations. That is, once the component architecture is configured, deployed and executed, no structural and behavioral modification is possible. The actual tendency in the software development is to allow applications to reconfigure *on the fly*, while executing. One of the possible CwSTS evolutions is to allow the components to reconfigure themselves at execution time, especially when considering intra-component connections, without having to recompile code. Another complementary evolution could allow components to substitute their interaction protocol with a compatible one (protocol substitution was studied in Section 5.3.4 at page 126) at execution time. This would be possible by conceiving components capable of executing by following many phases like, for example, *started*, *in configuration*, *paused*, *stopped*. For simplicity reasons we studied only the case where components communicate by following a point to point (1 to 1) communication schema. This means that a component provided service can be called by only one unique other component requiring that service. In reality,

the point to point communication is very restrictive especially if we consider client-server applications where multiple clients access the services of one server component. Other communication schemas could be considered by an extension of the CwSTS model. These schemas could include 1 to n or n to 1 communications allowing for the design of complex architectures. The synchronization mechanism as already implemented is partially equipped in order to deal with these kind of communication schemas, but complementary studies have to be realized for an effective implementation.

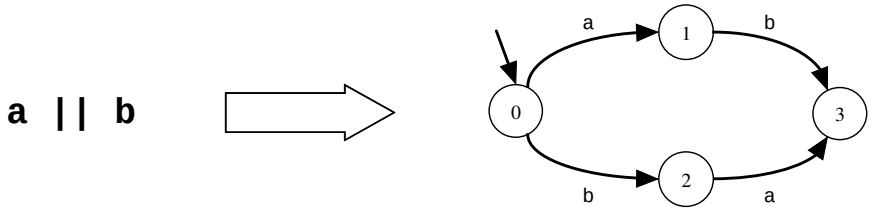


Figure 7.1: Parallel Execution of Two Actions.

SFSP is a process-based description language. This language is very precise in describing interaction protocols, but for a special case when we want to describe two transitions that can be executed in parallel, the only possible way is to represent it as presented in Figure 7.1. The SFSP and FSP languages describe parallel transitions as executing in a sequential manner. However, in a distributed environment, the execution of individual actions is rarely sequential. What we require from a protocol description at this level is to abstract from execution details which, for this case, seems not to be the case. One possible evolution of the SFSP language will be to abstract from the parallel execution implementation in order to allow for a more general description of interaction protocols.

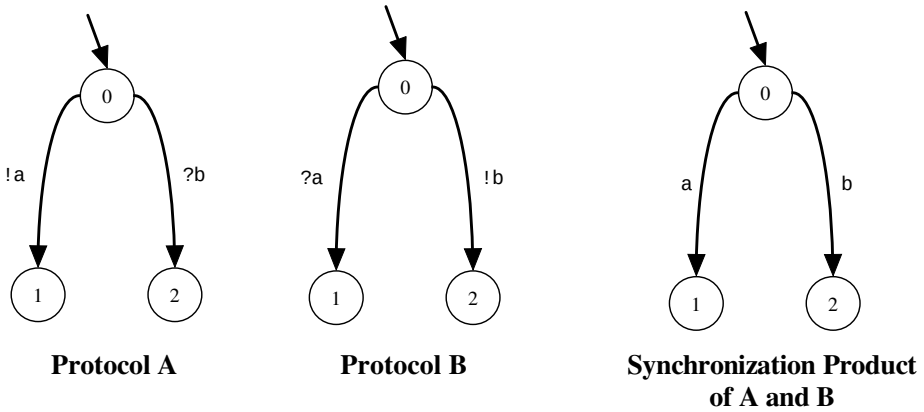


Figure 7.2: Mixed State Situation.

Our CwSTS-P implementation deals with the realization of the synchronous product in a very simple way. However, the current (distributed synchronisation) implementation does not explicitly consider a special case known under the name of *mixed states* [Yellin 97]. Figure 7.2 depicts such a case where two components (A and B) each of them executing

its own protocol (Protocol A and Protocol B respectively) are in a state where a receipt or an emission is possible. For this special case, the distributed synchronisation does not guarantee that the interaction of the two protocol is deadlock free. For a better understanding, lets assume that the two components initiate a sending in *the same time*. According to our implementation, the sender is blocked until the receiver acknowledges the receipt but in this case the two components will be blocked as each of them is in a sending phase that do not allow for a receipt.

For the centralised synchronisation mechanism mixed states does not represent a problem. However, the main disadvantage of this mechanism is the overhead communication necessary in order to synchronise components and the fact that guards can not be evaluated at the arbiter level thus representing a problem in the realisation of the synchronous product. Future work can consider the combination of the two mechanisms that will solve the problems related to mixed states or the extension of the two mechanisms to allow multiplicity, or another guard evaluation mechanism in the case of the centralised solution.

Some other future evolutions can include the development of visualization and verification tools associated with our SFSP language as this is the case for the FSP language (see at http://www.doc.ic.ac.uk/ltsa/).

Bibliography

[Abadi 93] M. Abadi & L. Lamport. *Composing specifications.* ACM Trans. Program. Lang. Syst., vol. 15, no. 1, pages 73–132, 1993.

[Abadi 95] M. Abadi & L. Lamport. *Conjoining Specifications.* ACM Transactions on Programming Languages and Systems, vol. 17, no. 3, pages 507–535, May 1995.

[Ahuja 86] S. Ahuja, N. Carriero & D. Gelernter. *Linda and Friends.* IEEE Computer, pages 26–34, August 1986.

[Aldrich 02a] J. Aldrich, C. Chambers & D. Notkin. *Architectural Reasoning in ArchJava*, 2002.

[Aldrich 02b] J. Aldrich, C. Chambers & D. Notkin. *ArchJava: Connecting Software Architecture to Implementation*, May 2002.

[Aldrich 02c] J. Aldrich, C. Chambers & D. Notkin. *ArchJava: Connecting software architecture to implementation.* In Proceedings of the 24th International Conference on Software Engineering (ICSE-02), pages 187–197. ACM Press, 19–25 2002.

[Allen 97] R. J. Allen. *A formal approach to software architecture.* PhD thesis, Carnegie Mellon University, 1997.

[André 96] C. André. *Representation and Analysis of Reactive Behaviors: A Synchronous Approach.* In Computational Engineering in Systems Applications (CESA), pages 19–29, Lille (F), July 1996. IEEE-SMC.

[Arbab 93] F. Arbab, I. Herman & P. Spilling. *An Overview of Manifold and its Implementation*, September 23 1993.

[Arbab 01] F. Arbab. *Coordination of Mobile Components.* Electr. Notes Theor. Comput. Sci., vol. 54, 2001.

[Arbab 02] F. Arbab, J. V. Guillen Scholten, F.S. de Boer & M. M. Bonsangue. *A Channel-Based Coordination Model for Components.* Rapport technique, Centruum voor Wiskunde en Informatica, 2002.

[Arnold 94] A. Arnold. Finite Transition Systems. International Series in Computer Science. Prentice-Hall, 1994.

[Arnold 98] K. Arnold & J. Gosling. The Java programming language. The Java Series. Addison-Wesley, 2nd edition, 1998.

[Attiogbé 03] C. Attiogbé, P. Poizat & G. Salaün. *Integration of Formal Datatypes within State Diagrams.* In FASE'2003, volume 2621 of *Lecture Notes in Computer Science*, pages 344–355. Springer-Verlag, 2003.

[Barbacci 90] M. R. Barbacci & J. M. Wing. *A language for distributed applications.* In Proceedings: 1990 International Conference on Computer Languages, pages 59–68. IEEE Computer Society Press, 1990.

[Barros 05] T. Barros, L. Henrio & E. Madelaine. *Behavioral Models for Hierarchical Components.* In Proceedings of SPIN'05. Springer-Verlag, 2005. To Appear.

[Bass 98] L. Bass, P. Clements & R. Kazman. Software Architecture in Practice. Series in Software Engineering. Addison Wesley, Reading, MA, USA, 1998.

[Baumeister 06] H. Baumeister, F. Hacklinger, R. Hennicker, A. Knapp & M. Wirsing. *A Component Model for Architectural Programming.* Electr. Notes Theor. Comput. Sci., vol. 160, pages 75–96, 2006.

[Bellissard 95] L. Bellissard, S.B. Atallah, A. Kerbrat & M. Riveill. *Component-based programming and Application Management with Olan*, June 1995.

[Bergstra 84] J. A. Bergstra & J. W. Klop. *The Algebra of Recursively Defined Processes and the Algebra of Regular Processes.* In J. Paredaens, editeur, Proceedings ICALP '84, volume 172 of *LNCS*, pages 82–95, Antwerp, 1984. Springer-Verlag.

[Bergstra 01] J. A. Bergstra, A. Ponse & S. A. Smolka, editeurs. Handbook of Process Algebra. Elsevier, 2001.

[Berry 92] G. Berry & G. Gonthier. *The Esterel Synchronous Programming Language: Design, Semantics, Implementation.* Sci. Comput. Program., vol. 19, no. 2, pages 87–152, 1992.

[Berthomieu 89] B. Berthomieu. *Implementing CCS, the LCS experiment.* Rapport technique 89425, 1989.

[Beugnard 99] A. Beugnard, J.-M. Jézéquel, N. Plouzeau & D. Watkins. *Making Components Contract Aware*, July 1999.

[Bosch 00] J. Bosch. Design and Use of Software Architectures - Adopting and evolving a Product Line Approach. Addison-Wesley, 2000.

[Bowman 97] H. Bowman, C. Briscoe-Smith, J. Derrick & B. Strulo. *On behavioural subtyping in LOTOS*, 1997.

[Brinksma 87] E. Brinksma, G. Scollo & C. Steenbergen. *LOTOS Specifications, their Implementations and their Tests*, 1987.

[Brockschmidt 95] K. Brockschmidt. Inside OLE. Microsoft Press, 2nd edition, 1995.

[Brooks 75] F. Brooks. *The Mythical Man-Month*, 1975.

[Brown 96] A. W. Brown & K. C. Wallnau. *Engineering of Component-Based Systems.* In Alan W. Brown, editeur, Component-Based Software Engineering: Selected Papers from the Software Engineering Institute, pages 7–13. IEEE Computer Society Press, Los Alamitos, CA, 1996.

[Brown 98] A. W. Brown & K. C. Wallnau. *The Current State of CBSE*, September/ October 1998.

[Bruneton 04] E. Bruneton, T. Coupaye, M. Leclercq, V. Quéma & J.-B. Stefani. *An Open Component Model and Its Support in Java.* In Ivica Crnkovic, Heinz W. Schmidt Judit A. Stafford & Kurt C.Wallnau, editeurs, CBSE, volume 3054, pages 7–22. Lecture Notes in Computer Science, 2004.

[Büchi 97] M. Büchi & W. Weck. *A Plea for Grey-Box Components.* In Gary T. Leavens & Murali Sitaraman, editeurs, Proceedings of the First Workshop on the Foundations of Component-Based Systems, Zurich, Switzerland, September 26 1997, pages 39–49, September 1997.

[Calder 02] M. Calder, S. Maharaj & C. Shankland. *A Modal Logic for Full LOTOS Based on Symbolic Transition Systems.* The Computer Journal, vol. 45, no. 1, pages 55–61, 2002.

[Cardelli 85] L. Cardelli & P. Wegner. *On Understanding Types, Data Abstraction, and Polymorphism.* ACM Computing Surveys, vol. 17, no. 4, pages 471–522, December 1985.

[Cardelli 98] L. Cardelli & A.D. Gordon. *Mobile Ambients*, 1998. Held as Part of the Joint European Conferences on Theory and Practice of Software (ETAPS'98), (Lisbon, Portugal, March/April 1998).

[Cardone 00] R. Cardone, D. Batory & C. Lin. *Java Layers: Extending Java to Support Component-Based Programming*, June 2000.

[Carreiro 89] N. Carreiro & D. Gelernter. *Linda in Context.* Communications of the ACM, vol. 32, no. 4, 1989.

[Carrez 03] C. Carrez. *Contrats Comportementaux pour Composants.* Thèse, December 01 2003.

[Cherinka 98] R. Cherinka, C. Michael Overstreet, J. Ricci & M. Schrank. *Maintaining a COTS Component-Based Solution Using Traditional Static Analysis Techniques.* In LNCS, volume 1543, pages 165–166, 1998.

[Choppy 00] C. Choppy, P. Poizat & J.-C. Royer. *A Global Semantics for Views*. In T. Rus, editeur, International Conference, AMAST'2000, volume 1816 of *Lecture Notes in Computer Science*, pages 165–180. Springer-Verlag, 2000.

[Choppy 01] C. Choppy, P. Pascal & J.-C. Royer. *The Korrigan Environment*, January 2001.

[Clarke 81] E.M. Clarke & E.A. Emerson. *Design and synthesis of synchronization skeletons using branching-time temporal logic*, May 1981.

[Clarke 00] E. M. Clarke, S. M. German, Y. Lu, H. Veith & D. Wang. *Executable Protocol Specification in ESL*, 2000.

[Collet 05] P. Collet, R. Rousseau, T. Coupaye & N. Rivierre. *A Contracting System for Hierarchical Components*. In George T. Heineman, Ivica Crnkovic, Heinz W. Schmidt, Judith A. Stafford, Clements Szyperski & Kurt C. Wallnau, editeurs, CBSE, volume 3489, pages 187–202. Lecture Notes in Computer Science, 2005.

[Coupaye 02] T. Coupaye, E. Bruneton & J.-B. Stéfani. *The Fractal Composition Framework*, 2002.

[de Alfaro 01] L. de Alfaro & T. Henzinger. *Interface automata*. In Proceedings of the. ACM Press, January 2001.

[DePaoli 93] F. DePaoli & F. Tisato. *Development of a Collaborative Application in CSDL*. In Robert Werner, editeur, Proceedings of the 13th International Conference on Distributed Computing Systems, pages 210–218, Pittsburgh, PA, May 1993. IEEE Computer Society Press.

[DePaoli 94] F. DePaoli & F. Tisato. *Cooperative Systems Configuration in CSDL*. In Proceedings of the 14th International Conference on Distributed Computing Systems, pages 304–311, Los Alamitos, CA, USA, June 1994. IEEE Computer Society Press.

[Eddon 98] G. Eddon & H. Eddon. Microsoft Programming Series. Microsoft Press, Redmond, WA, 1998.

[E.M. Clarke 83] E.M. Clarke, E.A. Emerson & A.P. Sistla. *Automatic Verification of Finite-State Concurrent Systems Using Temporal Logic*. In Proceedings of the tenth Annual ACM Symposium on Principles of Programming Languages, 1983.

[Escoffier 05] C. Escoffier & D. Donsez. *FractNet - Une implémentation du modèle á composant Fractal pour .NET*. In Lionel Seinturier, editeur, 2nd French Workshop on Aspect-Oriented Software Development (JFDLPA 2005), September 2005.

[Eugster 01] P. Eugster, R. Guerraoui & C. Damm. *On objects and events*, October 2001.

[Farias 03] A. Farias. *Un modèle de composants avec des protocoles explicites*. PhD thesis, Ecole Doctorale Sciences et Technologies de l4information et des Materiaux, December 2003.

[Fassino 02] J.-P. Fassino, J.-B. Stefani, J. Lawall & G. Muller. *THINK: a Software Framework for Component-based Operating System Kernels*. In 2002 USENIX Annual Technical Conference, pages 73–86, Monterey, CA, June 2002. USENIX.

[Fernandez 92] J.-C. Fernandez, H. Garavel, L. Mounier, A. Rasse, C. Rodriguez & J. Sifakis. *A Toolbox for the Verification of LOTOS Programs*, May 1992.

[Fielding 00] R. T. Fielding. *Architectural Styles and the Design of Network-based Software Architectures*. PhD thesis, University of California, IRVINE, 2000.

[Findler 01] R. B. Findler, M. Latendresse & M. Felleisen. *Behavioral Contracts and Behavioral Subtyping*. In Volker Gruhn, editeur, Proceedings of the Joint 8th European Software Engeneering Conference and 9th ACM SIGSOFT Symposium on the Foundation of Software Engeneering (ESEC/FSE-01), volume 26, 5 of *SOFTWARE ENGINEERING NOTES*, pages 229–236, New York, September 10–14 2001. ACM Press.

[Flatt 98] M. Flatt & M. Felleisen. *Units: Cool modules for HOT languages*, May 1998.

[Frølund 98] S. Frølund & J. Koistinen. *Quality of Service Specification in Distributed Object Systems Design*. In Proceedings of The Fourth USENIX Conference on Object-Oriented Technologies and Systems, pages 1–18. The USENIX Association, 1998.

[Gamma 95] E. Gamma, R. Helm, R. Johnson & J. Vlissides. Design patterns. Addison Wesley Professional Computing Series. Addison Wesley, 1995. http://www.aw.com.

[Garavel 03] H. Garavel. *Défense et illustration des algèbres de processus*. In Zoubir Mammer, editeur, Actes de l'Ecole dété Temps Réel ETR, Toulouse, France, septembre 2003.

[Garlan 94] D. Garlan & M. Shaw. *An Introduction to Software Architecture*. Technical Report CS-94-166, Carnegie Mellon University, School of Computer Science, Software architecture, software design, software engineering 1994.

[Gelernter 92] D. Gelernter & N. Carriero. *Coordination languages and their significance*. Communications of the ACM, vol. 35, no. 2, pages 97–107, 1992.

[Giannakopoulou 99] D. Giannakopoulou. *Model Checking for Concurrent Software Architectures*, 1999.

[Godefroid 91] P. Godefroid & P. Wolper. *Using partial orders for the efficient verification of deadlock freedom and safety properties*. In Proc. 1991 Computer-Aided Verification Workshop, 1991.

[Graf 86] S. Graf & J. Sifakis. *A Logic for the Specification and Proof of Regular Controllable Processes of CCS*. ACTAINF: Acta Informatica, vol. 23, 1986.

[Haase 02] K. Haase. *Java^{TM} Message Service API Tutorial*, November 2002. Version 1.3.

[Hamilton 97] G. Hamilton. *JavaBeans^{TM}*, July 1997. Version 1.01.

[Harel 87] D. Harel. *Statecharts: A Visual Formalism for Complex System*, March 1987.

[Heineman 01a] G. T. Heineman & W. T. Councill, editeurs. Component-based software engineering. Addison Wesley, 2001.

[Heineman 01b] G. T. Heineman & W. T. Councill. *Component-Based Software Engineering: Putting the Pieces Together*, 2001.

[Heisel 97] M. Heisel & N. Lévy. *Using LOTOS Patterns to Characterize Architectural Styles*, 1997.

[Hoare 85] C. A. R. Hoare. *Communicating Sequential Processes*, 1985.

[Holzbacher 96] A. A. Holzbacher. *A Software Environment for Concurrent Coordinated Programming*. In P. Ciancarini & C. Hankin, editeurs, Coordination Languages and Models, volume 1061 of *LNCS*, pages 249–266. Springer-Verlag, Berlin, Germany, 1996.

[Hopcroft 01] J. E. Hopcroft, R. Motwani & J. D. Ullman. *Introduction to Automata Theory, Languages, and Computation*, 2001.

[Hoschka 98] P. Hoschka. *An Introduction to the Synchronized Multimedia Integration Language; World Wide Web Consortium;*. IEEE-MULTIMEDIA, vol. 5, no. 4, pages 84–88, October–December 1998.

[Ingolfsdottir 01] A. Ingolfsdottir & H. Lin. A Symbolic Approach to Value-passing Processes, chapitre 7 in [Bergstra 01], pages 427–478. Elsevier, 2001.

[Keller 76] R. M. Keller. *Formal verification of parallel programs*. Commun. ACM, vol. 19, no. 7, pages 371–384, 1976.

[Kenney 95] J. J. Kenney. *Executable Formal Models of Distributed Transactions Systems based on Event Processing*, December 1995.

[Kramer 90] J. Kramer, J. Magee & A. Finkelstein. *A Constructive Approach to the Design of Distributed Systems*. In Proceedings of the 10th International Conference on Distributed Computing Systems (ICDCS), pages 580–587, Washington, DC, 1990. IEEE Computer Society.

[Krüger 99] I. Krüger, R. Grosu, P. Scholz & M. Broy. *From MSCs to Statecharts*, 1999.

[Larsson 00] M. Larsson. *Applying Configuration Management Techniques to Component-Based Systems*, 2000.

[Layaïda 05] O. Layaïda & D. Hagimont. *Plasma: A component-based framework for building self-adaptive applications.* In SPIE/IS&T Symposium On Electronic Imaging, Conference of Embedded Multimedia Processing and Communications, San Jose, CA, USA, January 2005.

[Leavens 00] G. T. Leavens, K. Rustan, M. Leino, E. Poll, C. Ruby & B. Jacobs. *JML: notations and tools supporting detailed design in Java.* In OOPSLA 2000 Companion, Minneapolis, Minnesota, pages 105–106, 2000.

[Legond-Aubry 03] F. Legond-Aubry, G. Florin & D. Enselme. *Modèle abstrait d'assemblage de composants par contrats.* Technical report livrable 1.4, Projet RNTL Accord, 2003.

[Lin 96] H. Lin. *Symbolic Transition Graph with Assignment.* In International Conference on Concurrency Theory, pages 50–65, 1996.

[Liskov 94] B. H. Liskov & J. M. Wing. *A Behavioral Notion of Subtyping.* ACM Transactions on Programming Languages and Systems, vol. 16, no. 6, pages 1811–1841, November 1994.

[Logrippo 92] L. Logrippo, M. Faci & M. Haj-Hussein. *An introduction to LOTOS: Learning by examples.* Computer Networks and ISDN Systems, vol. 23, pages 325–342, 1992.

[Lomuscio 04] A. Lomuscio & D. Nute, editeurs. Deontic logic in computer science, 7th international workshop on deontic logic in computer science, deon 2004, madeira, portugal, may 26-28, 2004. proceedings, volume 3065 of *Lecture Notes in Computer Science.* Springer, 2004.

[Loureiro 91] A. A. F. Loureiro, Samuel T. Chanson & Song T. Vuong. *FDT Tools For Protocol Development.* Rapport technique TR-91-05, Department of Computer Science, University of British Columbia, May 1991. Tue, 22 Jul 1997 22:21:25 GMT.

[Luckham 95a] D. C. Luckham, J. L. Kenney, L. M. Augustin, J. Vera, D. Bryan & W. Mann. *Specification and Analysis of System Architecture Using Rapide.* IEEE Transactions on Software Engineering, vol. 21, no. 4, pages 336–355, 1995.

[Luckham 95b] D.C Luckham & J. Vera. *An Event-Based Architecture Definition Language*, 1995.

[Löwy 01] J. Löwy. COM and .NET component services. O'Reilly, September 2001.

[Lynch 87] N. Lynch & M. Tuttle. *Hierarchical Correctness Proofs for Distributed Algorithms*, 1987.

[Magee 92] J. Magee, N. Dulay & J. Kramer. *Structuring Parallel and Distributed Programs.* In Proceedings of the International Workshop on Configurable Distributed Systems, London, 1992.

[Magee 95] J. Magee, N. Dulay, S. Eisenbach & J. Kramer. *Specifying Distributed Software Architectures*, 25–28 September 1995.

[Magee 99] J. Magee & J. Kramer. Concurrency: State models & Java programs. Wiley, 1999.

[Malone 94] T. W. Malone & K. Crowston. *The Interdisciplinary Study of Coordination.* ACM Computing Surveys, vol. 26, no. 1, pages 87–119, March 1994.

[Marvie 02] R. Marvie & M.-C. Pellegrini. *Modèles de composants, un état de l'art*, 2002.

[Matena 06] V. Matena & M. Hapner. *Enterprise JavaBeans^{TM} specification v3.0*, March 2006. Final Release.

[McDirmid] S. McDirmid & W.C. Hsieh. *Aspect-Oriented Programming with Jiazzi.*

[McDirmid 01] S. McDirmid, M. Flatt & W. C. Hsieh. *Jiazzi: New-Age Components for Old-Fashioned Java*, 2001.

[McIlroy 68] M.D. McIlroy. *Mass produced software components.* In P. Naur & B. Randell, editeurs, Proceedings of the NATO Conference on Software Engineering, pages 138–155, Garmish, Germany, October 1968. NATO Science Committee.

[Medvidovic 96] N. Medvidovic, P. Oreizy, J. E. Robbins & R. N. Taylor. *Using Object-Oriented Typing to Support Architectural Design in the C2 Style*, October 16–18 1996.

[Medvidovic 00a] N. Medvidovic & R. N. Taylor. *A Classification and Comparison Framework for Software Architecture Description Languages.* IEEE - Transactions on Software Engineering, vol. 26, no. 1, pages 70–93, 2000.

[Medvidovic 00b] N. Medvidovic & R. N. Taylor. *A Classification and Comparison Framework for Software Architecture Description Languages*, 2000.

[Mehta 00] N. R. Mehta, N. Medvidovic & S. Phadke. *Towards a taxonomy of software connectors.* In ICSE '00: Proceedings of the 22nd international conference on Software engineering, pages 178–187, New York, NY, USA, 2000. ACM.

[Mercouroff 97] N. Mercouroff & A. Parhar. *TINA Computational Modelling Concepts and Object Definition Language.* In IS&N, pages 15–24, 1997.

[Merritt 91] M. Merritt, F. Modugno & M.R. Tuttle. *Time-Constrained Automata.* In J. C. M. Baeten & J. F. Groote, editeurs, CONCUR: 2nd International Conference on Concurrency Theory, volume 527 of *Lecture Notes in Computer Science*, pages 408–423, Berlin, August 1991. Springer-Verlag.

[Meyer 91] B. Meyer. *Eiffel: The Language*, 1991.

[Meyer 92] B. Meyer. *Applying "Design by Contract"*, October 1992.

[Mikhajlov 98] L. Mikhajlov & E. Sekerinski. *A Study of the Fragile Base Class Problem.* In Eric Jul, editeur, ECOOP '98 — Object-Oriented Programming, 12th European Conference , Brussels, Proceedings, volume 1445 of *Lecture Notes in Computer Science*, pages 355–382. Springer-Verlag, July 1998.

[Milner 84] R. Milner. *A Complete Inference System for a Class of Regular Behaviors.* Journal of Computer and System Sciences, vol. 28, no. 3, pages 439–466, June 1984.

[Milner 89] R. Milner. Communication and concurrency. Prentice-Hall, Inc., 1989.

[Milner 92] R. Milner, J. Parrow & D. Walker. *A Calculus of Mobile Processes, I*, September 1992.

[Milner 99] R. Milner. *Communicating and Mobile Systems: the π-Calculus*, May 1999.

[Mitchell 79] J. G. Mitchell, W. Maybury & R. Sweet. Xerox Research Center, Palo Alto, CA, 1979.

[Morisio 00] M. Morisio, C. B. Seaman, A. T. Parra, V. R. Basili, S. E. Kraft & S. E. Condon. *Investigating and improving a COTS-based software development.* In Proceedings of the 22nd International Conference on Software Engineering, pages 32–41. ACM Press, June 2000.

[Najm 99] E. Najm & A. Nimour. *Explicit Behavioral Typing for Object Interfaces.* In Ana M. D. Moreira & Serge Demeyer, editeurs, ECOOP Workshops, volume 1743 of *Lecture Notes in Computer Science*, page 321. Springer, 1999.

[Nielson 93] F. Nielson & H. R. Nielson. *From CML to Process Algebras (Extended Abstract).* In Eike Best, editeur, CONCUR, volume 715 of *Lecture Notes in Computer Science*, pages 493–508. Springer, 1993.

[Nierstrasz 93] O. Nierstrasz. *Regular types for active objects.* In Proceedings of the eighth annual conference on Object-oriented programming systems, languages, and applications, pages 1–15. ACM Press, 1993.

[Noyé 05] J. Noyé, S. Pavel, P. Poizat & J.-C. Royer. *A Formal Component Model with Explicit Symbolic Protocols and its Java Implementation.* Rapport technique, Ecole des Mines de Nantes, 2005.

[OCL 03] Object Management Group. *Object Constraint Language Specification,*, March 2003. statut : « Version 1.5 ».

[Ozanne 07] A. Ozanne. *Interact : un modèle général de contrat pour la garantie des assemblage de composants et services.* PhD thesis, Université Pierre et Marie Curie (Paris VI), November 2007.

[Papadopoulos 98] G. A. Papadopoulos & F. Arbab. *Coordination Models and Languages.* In The Engineering of Large Systems, volume 46 of *Advances in Computers*, pages 329–400. Academic Press, August 1998.

[Pavel 04] S. Pavel, J. Noyé & J.-C. Royer. *Dynamic Configuration of Software Product Lines in ArchJava.* In Robert L. Nord, editeur, Software Product Lines: Third International Conference, LNCS, pages 90–109, Boston, MA, USA, September 2004. Springer-Verlag Heidelberg.

[Pavel 05a] S. Pavel, J. Noyé, P. Poizat & J.-C. Royer. *A Java Implementation of a Component Model with Explicit Symbolic Protocols.* In Proceedings of the 4th International Workshop on Software Composition (SC'05), volume 3628 of *LNCS*, pages 115–125. Springer-Verlag, April 2005.

[Pavel 05b] S. Pavel, J. Noyé & J.-C. Royer. *Un modéle de composant avec protocole symbolique.* In Journée du groupe Objets, Composants et Modèles, Bern, Suisse, March 2005.

[Plašil 98] F. Plašil, D. Bálek & R. Janečźek. *SOFA/DCUP: Architecture for Component Trading and Dynamic Updating*, January 1998.

[Pnueli 77] A. Pnueli. *The temporal logic of programs*, October 1977.

[Pnueli 81] A. Pnueli. *The temporal semantics of concurrent programs*, November 1981.

[Pryce 98] N. Pryce & S. Crane. *A Model of Interaction in Concurrent and Distributed Systems.* Lecture Notes in Computer Science, vol. 1429, pages 57–??, 1998.

[Puntigam 96] F. Puntigam. *Types for Active Objects based on Trace Semantics.* In Elie Najm et al., editeur, Proceedings of the Workshop on Formal Methods for Open Object-based Distributed Systems (FMOODS'96), Paris, France, 1996. Chapman & Hall.

[Puntigam 99] F. Puntigam. *Non-Regular Process Types.* In P. Amestoy*et al.*, editeurs, Proceedings of the 5th European Conference on Parallel Processing (Euro-Par'99), numéro 1685, Toulouse, France, 1999. Springer-Verlag.

[Purtilo 94] J. M. Purtilo. *The POLYLITH Software Bus.* ACM Transactions on Programming Languages and Systems, vol. 16, no. 1, pages 151–174, January 1994.

[Roman 90] G.-C. Roman & H. C. Cunningham. *Mixed Programming Metaphors in a Shared Dataspace Model of Concurrency.* IEEE Transactions on Software Engineering, vol. 16, no. 12, pages 1361–1373, December 1990.

[Rosenblum 97] D. S. Rosenblum & A. L. Wolf. *A Design Framework for Internet-Scale Event Observation and Notification.* Lecture Notes in Computer Science, vol. 1301, pages 344–360, 1997.

[Royer 03a] J.-C. Royer. *The GAT Approach to Specify Mixed Systems.* Informatica, vol. 27, no. 1, pages 89–103, 2003.

[Royer 03b] J.-C. Royer & M. Xu. *Analysing Mailboxes of Asynchronous Communicating Components.* In D. C. Schmidt R. Meersman Z. Tari & al., editeurs, CoopIS, DOA, and ODBASE, volume 2888 of *Lecture Notes in Computer Science*, pages 1421–1438. Springer-Verlag, 2003.

[Ruane 84] L. M. Ruane. *Abstract data types in assembly language programming.* j-SIGPLAN, vol. 19, no. 1, pages 63–67, January 1984.

[Rumbaugh 90] J. Rumbaugh, W. Lorenson & M. Blaha. *Object-Oriented Modeling and Design*, 1990.

[Seco 00] J. Costa Seco & L. Caires. *A Basic Model of Typed Components.* In Elisa Bertino, editeur, ECOOP, volume 1850 of *Lecture Notes in Computer Science*, pages 108–128. Springer, 2000.

[Seco 02] C. J. Seco & L. Caires. *ComponentJ in a nutshell*, 2002.

[Seinturier 05] L. Seinturier. *Réflexivité, aspects et composants pour l'ingénierie des intergiciels et des applications réparties.* Habilitation à diriger des recherches, Université de Pariv VI, France, December 2005.

[Shaw 95] M. Shaw, R. DeLine, D. V. Klein, T. L. Ross, D. M. Young & G. Zelesnik. *Abstractions for Software Architecture and Tools to Support Them.* IEEE Transactions on Software Engineering, vol. 21, no. 4, pages 314–335, April 1995.

[Shaw 96a] M. Shaw. *Procedure Calls Are the Assembly Language of Software Interconnection: Connectors Deserve First-Class Status.* Lecture Notes in Computer Science, vol. 1078, pages 17–32, 1996.

[Shaw 96b] M. Shaw & D. Garlan. Software architecture. perspectives on an emerging discipline. Prentice-Hall, 1996.

[Siegel 00] J. Siegel. *Corba 3 Fundamentals and Programming*, January 2000.

[Sommerville 94] I. Sommerville & G. Dean. *PCL: A configuration language for modelling evolving system architectures.* Rapport technique, January 05 1994.

[Sreedhar 02] V. C. Sreedhar. *Mixin'Up components*, May 19–25 2002.

[Staehli 03] R. Staehli, F. Eliassen, J. Ø. Aagedal & G. S. Blair. *Quality of Service Semantics for Component-Based Systems.* In Middleware Workshops, pages 153–157, 2003.

[Szyperski 96] C. Szyperski. *Independently Extensible Systems – Software Engineering Potential and Challenges.* In Proceedings of the 19th Australian Computer Science Conference, Melbourne, Australia, February 1996.

[Szyperski 02] C. Szyperski. *Component Software: Beyond Object-Oriented Programming*, 2002. 2nd edition.

[Turner 93] K. J. Turner, editeur. Using formal description techniques, an introduction to estelle, lotos and sdl. Wiley, 1993. ISBN 0-471-93455-0.

[Uchitel 03] S. Uchitel, J. Kramer & J. Magee. *Behaviour model elaboration using partial labelled transition systems.* In ESEC/FSE-11: Proceedings of the 9th European software engineering conference held jointly with 11th ACM SIGSOFT international symposium on Foundations of software engineering, pages 19–27, New York, NY, USA, 2003. ACM Press.

[UML 03] Object Management Group. *Unified Modeling Language*, March 2003. statut : « Version 1.5 ».

[van den Bos 89] J. van den Bos & C. Laffra. *PROCOL: A Parallel Object Language with Protocols.* In Norman Meyrowitz, editeur, OOPSLA'89 Conference Proceedings, pages 95–102. ACM Press, 1989.

[van Ommering 00] R. van Ommering, F. van der Linden, J. Kramer & J. Magee. *The Koala Component Model for Consumer Electronics Software*, March 2000.

[Vanderperren 03] W. Vanderperren, D. Suvée, B. Wydaeghe & V. Jonckers. *PacoSuite and JAsCo: A Visual Component Composition Environment with Advanced Aspect Separation Features.* In Mauro Pezzè, editeur, FASE, volume 2621 of *Lecture Notes in Computer Science*, pages 166–169. Springer, 2003.

[Wang 02] C. Wang, A. Carzaniga, D. Evans & A.L Wolf. *Security Issues and Requirements for Internet-Scale Publish-Subscribe Systems*, January 2002.

[Wegner 88] P. Wegner & S. B. Zdonik. *Inheritance as an Incremental Modification Mechanism or What Like Is and Isn't Like.* Lecture Notes in CS, vol. 322, page 55, 1988.

[Wegner 97] P. Wegner. *Why Interaction is more Powerful than Algorithms.* Communications of the ACM, vol. 40, no. 5, pages 80–91, May 1997.

[Wirth 77] N. Wirth. *MODULA : A Language for Modular Multiprogramming.* Software Practice and Experience, vol. 7, pages 3–35, 1977.

[Yellin 97] D. M. Yellin & R. E. Strom. *Protocol Specifications and Component Adaptors.* ACM Transactions on Programming Languages and Systems, vol. 19, no. 2, pages 292–333, 1997.