Département de formation doctorale en informatique **École doctorale SPI Lille**
UFR IEEA

Tinap : Modèle et infrastructure d'exécution orienté composant pour applications multi-tâches à contraintes temps réel souples et embarquées

THÈSE

présentée et soutenue publiquement le 26 mai 2008 (numéro d'ordre 4186)

pour l'obtention du

Doctorat de l'Université des Sciences et Technologies de Lille

(spécialité informatique)

par

Frédéric Loiret

Composition du jury

Président :	M. Jean-Luc Dekeyser, *Université de Lille I*
Rapporteurs :	M. Yvon Trinquet, *IRCCyN - Université de Nantes* M. Jean-Bernard Stefani, *INRIA Rhône-Alpes*
Examinateurs :	M. Jacques Pulou, *France Telecom R&D* M. David Servat, *CEA-LIST* Mme Laurence Duchien, *Université de Lille I* M. Lionel Seinturier, *Université de Lille I*

CEA – Laboratoire d'Intégration des Systèmes et des Technologies
Laboratoire d'Informatique Fondamentale de Lille — UMR USTL/CNRS 8022
INRIA Lille - Nord Europe

Mis en page avec la classe thloria.

Résumé

Cette thèse a été réalisée en co-encadrement entre les équipes CEA-LIST/LISE et LIFL-INRIA/ADAM. Notre proposition consiste à présenter un modèle et une infrastructure d'exécution orienté composant pour le domaine des applications multi-tâches à contraintes temps réel souples et embarquées (nommé TINAP).

Notre approche définit un modèle de composant reposant sur plusieurs vues : une « vue structurelle », placée au centre du cycle de conception, reposant initialement sur le modèle FRACTAL (assemblage de composants fonctionnels relativement à leurs interfaces, composition hiérarchique), une « vue dynamique » permettant au concepteur, dans une démarche descriptive, de personnaliser l'architecture métier pour définir les aspects d'ordre non-fonctionnels liés à la concurrence, et enfin une « vue implantation » et une « vue comportement » fournissant respectivement une abstraction de l'implantation interne des composants et de leur comportement à l'égard de leur environnement.

Nous avons également expérimenté le paradigme composant à différents niveaux d'abstraction. D'abord au niveau applicatif pour lequel notre modèle propose des concepts de haut niveau. Ensuite, pour celui de l'infrastructure d'exécution qui les implante. Enfin, dans une moindre mesure, au niveau du système d'exploitation qui doit fournir les services élémentaires nécessaires. Cette démarche est motivée par la volonté d'exploiter notre modèle « multi-vues » initialement défini en tant que modèle canonique et de l'adapter en fonction des besoins de chaque niveau d'abstraction. Cette expérimentation est menée avec THINK, une implantation en C des spécifications FRACTAL, initialement développée pour la construction de systèmes d'exploitation.

Enfin, nous avons cherché à ancrer notre proposition au sein d'un cadre méthodologique « dirigé par les modèles », nous permettant notamment d'implanter un prototype d'éditeur de haut-niveau facilitant significativement la conception d'applications TINAP via la plate-forme ECLIPSE.

Nous expérimentons TINAP par l'intermédiaire de deux cas d'étude que nous avons prototypés. En premier lieu, pour concevoir une application d'analyse et de contrôle de flux multimédias par l'intermédiaire de disques vinyles (il s'agit d'une application destinée aux « *DJ's* »). En second lieu pour expérimenter la mise en œuvre du modèle d'exécution ACCORD (une méthodologie de conception pour applications temps-réel développée au sein de l'équipe LISE).

Table des matières

Liste des tableaux **ix**

Chapitre 1 Introduction **1**

Chapitre 2 État de l'art **5**

2.1 Architectures, principes fondamentaux 5

2.2 Approches orientées architecture pour le domaine de l'embarqué 9

2.3 Synthèse et positionnement 24

Chapitre 3 Présentation générale **29**

3.1 Proposition 29

3.2 Organisation de la contribution 34

Chapitre 4 Espace de conception TINAP **35**

4.1 Introduction 36

4.2 Vue structurelle : le socle de l'approche 39

4.3 Vue dynamique et de contrôle 52

4.4 Vues implantations 65

4.5 Vues comportementales 72

4.6 TINAP à différents niveaux d'abstraction 85

4.7 Spécification de contraintes d'ordonnancement 86

4.8 Conclusion du chapitre 87

Chapitre 5 Infrastructure d'exécution TINAP **89**

5.1 Introduction 89
5.2 Concepts de niveau infrastructure 90
5.3 Implantation des membranes TINAP 93
5.4 Méta-modélisation d'une « vue infrastructure » 103
5.5 Bilan sur le niveau infrastructure 103
5.6 Niveau système d'exploitation 104
5.7 Conclusion du chapitre 105

Chapitre 6 Expérimentations et évaluations **107**

6.1 Introduction 107
6.2 DECKX : un « contrôleur » de flux multimédias 108
6.3 Implantation du modèle d'exécution ACCORD 117
6.4 Conclusion sur les expérimentations 127

Chapitre 7 Conclusion générale **129**

Bibliographie **137**

Annexe A Autres méta-modèles TINAP **143**

A.1 Comportement TINAP de « niveau architecture » 143
A.2 Descripteur de contrôle ACCORD 145

Annexe B Intégration ECLIPSE **147**

Glossaire **152**

Index **153**

Table des figures

2.1 Concepts structuraux et terminologie du modèle de composants FRACTAL. 10
2.2 Un connecteur COMPARE. 13
2.3 Assemblage de composants PECOS et espaces de données associés. 16
2.4 Exécution d'un assemblage de composants PECOS 16
2.5 Exemple de configuration CLARA (syntaxe graphique). 18
2.6 Exemple de configuration SAVECCM (syntaxe graphique). 19
2.7 Exemple de spécification d'une configuration AADL (notations graphiques). 21
2.8 Abstractions AUTOSAR , du matériel à l'applicatif. 22

3.1 Domaine d'application. 30
3.2 Paradigme composant à différents niveaux d'abstraction d'un système. 33

4.1 Liens entre vues TINAP. 36
4.2 Principaux paquetages des concepts TINAP. 37
4.3 Extrait du méta-modèle relatif aux définitions de composants. 39
4.4 Structure interne d'un composite (paquetage `core`). 40
4.5 Définition des interfaces. 42
4.6 Caractérisation des « interfaces d'entrée » et « de sortie ». 43
4.7 Interface pour définir des attributs. 43
4.8 Extrait de méta-modèle relatif aux liaisons de service. 44
4.9 Liaison de délégation pour les interfaces d'attributs. 44
4.10 Caractérisation des liaisons en fonction de la nature des interfaces liées. 45
4.11 Langage de définition d'interface et types de données manipulables (extrait). 45
4.12 Points d'interactions atomiques. 46
4.13 Paramétrages initiaux pour les interfaces d'attributs (extrait). 47

4.14 Conventions graphiques utilisées pour représenter la vue structurelle de niveau applicatif. . . . 48
4.15 Annotations. . . . 50
4.16 Annotations supplémentaires sur les interfaces et le contenu. . . . 50
4.17 Descripteurs de contrôle. . . . 53
4.18 Contrôleurs FRACTAL. . . . 54
4.19 Graphe d'états (a) d'une activité, (b) d'un composant mono-actif et (c) d'un composant multi-actif. . . . 55
4.20 Descripteurs de contrôle élémentaires pour composants mono-actif et multi-actif. . . . 56
4.21 Traces d'exécutions associées à un composant **(1)** lorsqu'il est attaché à un descripteur mono-actif et **(2)** à un descripteur multi-actif. . . . 57
4.22 Descripteur simple pour activation périodique. . . . 57
4.23 Chronogrammes pour les protocoles incrémentés (a) non borné et (b) borné (à 2) pour une liaison d'événement. . . . 58
4.24 Chronogrammes pour les protocoles (a) fugace et (b) rendez-vous. . . . 59
4.25 Chronogrammes pour les protocoles rafraîchis (a) avec consommation et (b) sans consommation (la notation ⊘ représente le non-rafraîchissement de la variable). . . . 59
4.26 Extrait du méta-modèle permettant d'annoter un comportement spécifique aux liaisons. . . . 61
4.27 Descripteur pour protéger les accès à un composant passif. . . . 61
4.28 Conventions graphiques utilisées pour représenter la vue dynamique de l'architecture. . . . 62
4.29 Composant passif et interactions avec son environnement. . . . 63
4.30 Exemple de composant primitif encapsulant une implantation multi-séquence : projection des points d'interactions atomiques en points d'entrée et d'interaction interne de séquence. . . . 66
4.31 Extrait du méta-modèle pour abstraire la structure interne d'une implantation. . . . 67
4.32 Vue structurelle du primitif `CompA` (partie supérieure) et vue implantation interne de son contenu (partie inférieure). . . . 70
4.33 Étapes d'extraction du comportement des descriptions TINAP. . . . 72
4.34 Exemple d'architecture TINAP (vue dynamique présentant les noms des interfaces). 73
4.35 Extrait de l'implantation de `CompA` et comportement de niveau composant primitif associé. . . . 75
4.36 Comportements des primitifs de l'application présentée figure 4.34. . . . 76
4.37 Extrait du méta-modèle relatif à la vue comportement de niveau composant. . . . 77
4.38 Automate de comportement de l'unique séquence d'activité globale du composant actif `PlayerManager`. . . . 79
4.39 Séquences d'activité globales de l'architecture. . . . 80
4.40 Exemples de composants passifs activement partagés. . . . 82

4.41 Une représentation graphique possible de la vue comportement de niveau architecture (à partir des entités définies au sein du méta-modèle). 83
4.42 Interactions entre les différents niveaux d'abstraction. 85
4.43 Annotations des signatures des interfaces TINAP. 85

5.1 Interposition. 91
5.2 Projection des interfaces (et signatures) de niveau applicatif (a) de `CompA` vers les interfaces de niveau infrastructure (b). 94
5.3 Intercepteurs et notion de « liaisons composites » au niveau infrastructure. 95
5.4 Vue dynamique de `CompA` de niveau applicatif (a) et intercepteurs qui lui sont associés au niveau infrastructure (b). 96
5.5 Contrôleurs de `CompA`. 97
5.6 Liaisons de la membrane. 98
5.7 Extrait de la membrane de `CompA`. 99
5.8 Comportements d'un intercepteur et d'un contrôleur de la membrane de `CompA`. . 100
5.9 Un composant protégé spécifié au niveau applicatif (a) et implantation de sa membrane au niveau infrastructure (b). 102
5.10 Laison composite non préemtable. 102
5.11 Méta-modèle pour caractériser une architecture TINAP de niveau infrastructure. . 103

6.1 Extrait du signal MSPINKY en stéréo. 110
6.2 Spectre fréquentiel du signal. 110
6.3 Architecture logicielle « gros grain » et connectique externe. 111
6.4 Définition des interfaces de flux audio. 112
6.5 Exemple d'un tableau d'échantillons (a) non-entrelacés et (b) entrelacés pour un flux audio en stéréo. 112
6.6 Vue dynamique d'un extrait de l'architecture de l'application DECKX. 113
6.7 Exemple d'architecture globale d'une application DECKX (avec deux entrées/sortie stéréo). 114
6.8 Structure informelle d'un RTO. 117
6.9 Ensemble des traitements associés à un message, `O1` envoit `m()` à `O2`. 120
6.10 Exemple d'architecture TINAP de niveau applicatif dont un composant est attaché à un « descripteur ACCORD ». 122
6.11 Intercepteur côté client. 123
6.12 Structure de la membrane de contrôle associée au composant métier `serveurAccord`. 124
6.13 Extrait de membrane de `RtcDispatcher`. 125

A.1 Méta-modèle pour sérialiser les informations relatives à la « vue comportement de niveau architecture » TINAP. 144

A.2 Descripteur de contrôle TINAP de niveau applicatif pour la spécification d'un « composant ACCORD ». 145

B.1 Superposition de captures d'écran de l'éditeur arborescent TINAP de niveau applicatif (vue structurelle). 148

B.2 Représentation sous forme de méta-modèle des principales balises définies dans la DTD standard de FRACTAL-ADL 148

Liste des tableaux

3.1 Concepts TINAP canoniques et extensions nécessaires à différents niveaux d'abstraction. 34

4.1 Préfixes des « mots clefs » utilisés pour l'implantation des primitifs TINAP. 69

Chapitre 1

Introduction

De nos jours, l'informatique prend une place prépondérante dans notre quotidien. Les systèmes de traitements de l'information couvrent inexorablement de nouveaux domaines. Les objets physiques qui nous entourent sont maintenant dotés de processeurs et de capacités de communication. Depuis ses débuts, l'ingénierie logicielle ne cesse donc de faire face à la complexité de ces systèmes d'information, certainement due à un nombre sans cesse grandissant de nouvelles fonctionnalités à intégrer pour parvenir à un produit fini. En outre, elle se doit de répondre aux besoins croissants de flexibilité car notre environnement engendre désormais une grande diversité, et ce, tant dans les supports matériels que pour les applicatifs qui s'y embarquent. De nombreux outils, formalismes, méthodes, infrastructures ont été proposés pour maîtriser les différentes étapes du cycle de conception du logiciel. L'innovation dans ce domaine est souvent caractérisée par un gain en pouvoir d'abstraction : il s'agit d'éliminer les aspects les moins pertinents de la réalité pour n'en considérer que les plus importants. C'est ainsi que de nouveaux paradigmes de conception de haut niveau ont été proposés.

Le « paradigme de conception composant » en est l'une des manifestations les plus récentes. Sa caractérisation marque un saut significatif dans l'histoire de l'ingénierie logicielle au même titre que la transition qui s'est opérée entre les langages procéduraux et la programmation orientée objet dans les années 90.
Il prend son essence dans la complémentarité de trois disciplines. D'abord dans l'énonciation du principe de **séparation des préoccupations** [Parnas, 1972], dont l'objectif est de modulariser au maximum un logiciel en fonctionnalités fortement couplées, plus petites et canoniques. En second lieu, des **langages de description d'architecture** [Medvidovic and Taylor, 2000], qui offrent les abstractions pour découpler les traitements fonctionnels du logiciel de leurs communications. Il s'agit alors de raisonner sur une structure de haut niveau, définie sous forme d'assemblages de composants interconnectés. Enfin, du processus de **développement par composants** [Szypersky *et al.*, 2002], initié par l'industrie qui offre la perception d'entité logicielle dotée d'interfaces contractuelles, conditionnée de telle sorte à être déployée de manière indépendante et donc pour laquelle les dépendances fonctionnelles (l'expression du requis) sont clairement identifiées. C'est aussi dans ce contexte que la notion de *conteneur* a été développée. Il s'agit de séparer les services techniques – implantés au sein d'une structure d'accueil – des services fonctionnels – les « composants sur étagères » – favorisant ainsi la réutilisation et l'intégration de l'existant.

L'utilisation du paradigme de conception composant est largement répandue au sein de l'industrie dans le domaine du traitement de l'information – les systèmes informatiques que l'on peut qualifier de « standards » – et est sujette à un développement continu. La complexité est alors

appréhendée par un gain significatif dans la compréhension de la structure du système en développement. Cette structuration du logiciel facilite sensiblement son analyse, sa maintenabilité, son évolutivité et elle identifie clairement les opportunités de réutilisation.

Cette thèse s'inscrit dans l'émergence de nouveaux besoins applicatifs dans le contexte de systèmes embarqués. Le logiciel mis en œuvre sur de tels systèmes est soumis à des contraintes de différentes natures, qui dépendent des domaines d'application. Il peut notamment s'agir de contraintes en terme d'espace mémoire et de ressources de calcul (qui sont limitées), de mobilité (consommation énergétique à partir d'une source autonome), d'ordre temporel (temps d'exécution déterminés, délais connus ou bornés à priori). Néanmoins, les récents progrés effectués dans le domaine de l'électronique embarquée et des systèmes de communications permettent de plus en plus d'envisager le support de nouvelles fonctionnalités. Ces changements engendrent inévitablement une complexité dans le processus de développement de tels applicatifs, il faut être alors en mesure de les maîtriser. Une approche prometteuse consiste à promouvoir le paradigme de conception composant, ayant fait ses preuves dans le domaine de l'ingénierie logicielle, pour le domaine de l'embarqué, et d'espérer en attendre les mêmes retombées.

Toutefois, cette adaptation n'est pas spontanée dès lors que le modèle de conception se doit de prendre en considération les bonnes abstractions inhérentes à ce domaine. Cela se manifeste par la nécessité de capturer dans l'espace conceptuel de haut niveau proposé les modèles de programmation ainsi que les heuristiques et patrons de conception du domaine.
En outre, l'applicatif embarqué dépend intrinsèquement de propriétés extra-fonctionnelles relatives aux contraintes sus-citées. Il est alors du ressort du processus de développement d'assurer leurs prédictabilités en amont, la fiabilité du logiciel, au même titre que les exigences fonctionnelles doit être validée. Pour cela, les informations contextuelles couramment externalisées sur les interfaces des composants issus de l'ingénierie logicielle doivent être sensiblement enrichies pour réifier les caractéristiques nécessaires.
Un élément capital provient également du fait que la fiabilité du logiciel dépend de caractéristiques propres à l'infrastructure d'exécution et à la plate-forme matérielle sous-jacente. De plus, le lien entre le langage de haut niveau proposé et les abstractions de bases manipulées par les outils d'analyse existant doit être établi.
Enfin, un frein à l'adoption de ce paradigme dans l'embarqué est lié au besoin important de gestion des ressources du système. La configurabilité et le contrôle de l'infrastructure d'exécution à fine granularité sont essentiels.

L'objectif de la thèse consiste à proposer un modèle de composant, une infrastructure d'exécution et un cadre méthodologique outillé de développement pour logiciels embarqués, considérant les points énoncés. Nous nous sommes notamment intéressés à expérimenter le paradigme composants à différents niveaux d'abstraction. D'abord au niveau applicatif pour lequel notre modèle propose des concepts de haut niveau. Ensuite, pour celui de l'infrastructure d'exécution qui les implante. Enfin, dans une moindre mesure, au niveau du système d'exploitation qui doit fournir les services élémentaires nécessaires. Cette démarche est motivée par la volonté d'exploiter un modèle canonique présentant de bonnes propriétés (en terme d'abstractions manipulées, de possibilités d'analyses, de contrôle sur les ressources) et de l'adapter en fonction des besoins de chaque niveau d'abstraction. En outre, il s'agit de bénéficier d'une « approche constructiviste » à tous ces niveaux, permettant de contrôler finement l'ensemble de l'infrastructure par l'intermédiaire de concepts communs.

Cette thèse s'articule autour de six chapitres. Le **chapitre 2** commence par présenter les principes fondamentaux des langages de description d'architecture et des modèles de composant qui font consensus d'un point de vue de l'ingénierie logicielle. Il est alors intéressant de considérer comment les besoins et contraintes de conception de systèmes embarqués sont adressés pour

promouvoir une démarche basée sur l'assemblage de composant et sur l'architecture. Nous présentons différentes approches pour tenter d'en donner un aperçu global : elles sont influencées par le domaine de l'ingénierie logicielle ou définissent un langage spécifique pour le domaine de l'automobile ou des applications temps réel de contrôle de processus, ou encore sont définies dans le cadre de standards industriels. Une synthèse de ce chapitre nous permet de mettre en lumière les éléments qui nous semblent pertinents et qui ont notamment influencé notre proposition.

Le **chapitre 3** présente de manière générale et concise les éléments essentiels de TINAP, et les facteurs qui ont influencé notre proposition de modèle de composant et d'infrastructure d'exécution pour le domaine de l'embarqué. Ils sont inspirés de démarches et mécanismes issus de l'ingénierie logicielle, comme par exemple le patron « de conception et de programmation orientée membrane » qui est introduit dans ce chapitre. Celui-ci prend son essence dans les spécifications FRACTAL et mis en œuvre dans ses différentes implantations. Il s'agit d'une approche générative intervenant en amont de la compilation (ou à l'exécution). À partir d'informations contextuelles, elle consiste à déployer des fonctionnalités de contrôle dédiées autour d'instances de composants déterminées. Enfin, ce chapitre présente notre vision de l'exploitation du paradigme composant à différents niveaux d'abstraction du système comme nous l'avons énoncé. Cela correspond au cadre général dans lequel s'inscrit notre démarche : un ensemble de concepts canonique pour une conception orientée composant, alors utilisée à tous ces niveaux, ces derniers étant intercomposés également par assemblages.

Le **chapitre 4** constitue le chapitre central de la thèse en présentant cet espace de conception canonique TINAP. Il repose sur plusieurs vues : une « vue structurelle », placée au centre du cycle de conception, reposant initialement sur le modèle Fractal (assemblage de composants fonctionnels relativement à leurs interfaces, composition hiérarchique), une « vue dynamique » permettant au concepteur, dans une démarche descriptive, de personnaliser l'architecture métier pour définir des aspects d'ordre non fonctionnels par exemple liés aux aspects multi-tâches, concurrents, aux modèles d'interaction et de synchronisation entre composants. Enfin une « vue comportement » fournissant une abstraction de l'implantation interne des composants. À partir de ces spécifications, il est possible d'automatiser certains traitements dont l'objectif est d'aboutir à des modèles analysables relativement aux assemblages des composants.

Le **chapitre 5** présente les concepts de l'infrastructure d'exécution TINAP et son implantation. Comme nous l'avons précisé, elle se conçoit également par assemblages de composants, dits « composants de contrôle » dont la sémantique est spécialisée pour prendre en compte les besoins nécessaires de cette couche d'abstraction. Notamment, elle met en œuvre au sein des membranes les concepts de haut niveau proposés par TINAP. Ce chapitre donne également un aperçu des fonctionnalités élémentaires fournies par le système d'exploitation (OS) nécessaire à l'exécution de l'infrastructure TINAP. Comme nous l'avons précisé, l'OS est aussi caractérisé par des assemblages de composants et implanté en THINK, une implantation en C de FRACTAL.

Le **chapitre 6** présente deux études de cas dont l'objectif est d'évaluer notre approche. En premier lieu, pour concevoir une application d'analyse et de contrôle de flux multimédias par l'intermédiaire de disques vinyles – DECKX– il s'agit d'une application destinée aux « disc-jockeys ». Elle permet d'évaluer certains aspects de TINAP au niveau applicatif. En second lieu pour expérimenter la mise en œuvre du modèle d'exécution ACCORD, une méthodologie de conception d'application temps réel, développée au sein du LLSP. L'objectif est d'évaluer TINAP pour la conception de membranes dédiées au sein de son infrastructure d'exécution.

Enfin, le **chapitre 7** conclut les travaux menés durant cette thèse et en donne les perspectives.

Chapitre 2

État de l'art

Sommaire

2.1 Architectures, principes fondamentaux 5
2.1.1 Le paradigme composant 5
2.1.2 Langages de description d'architecture 6
2.1.3 Une pléthore de propositions 8
2.2 Approches orientées architecture pour le domaine de l'embarqué 9
2.2.1 FRACTAL/THINK 9
2.2.2 COMPARE 12
2.2.3 PECOS 14
2.2.4 CLARA 17
2.2.5 SAVECCM 18
2.2.6 AADL 19
2.2.7 AUTOSAR 22
2.3 Synthèse et positionnement 24
2.3.1 Espace de conception 24
2.3.2 Support à l'exécution 26
2.3.3 Importance d'un support outillé 27

2.1 Architectures, principes fondamentaux

L'objectif de cette section est de présenter les points fondamentaux qui caractérisent les modèles de composants et les langages de descriptions d'architectures. Il s'agit de caractériser les éléments qui font consensus d'un point de vue de l'ingénierie logicielle avant de se focaliser sur différentes approches exploitant de manière prépondérante ces notions.

2.1.1 Le paradigme composant

La réutilisabilité et la composabilité ont toujours été des objectifs importants de l'ingénierie du logiciel. En leur temps, les approches procédurales et orientées objet ont chacune apporté leurs solutions. Actuellement, la tendance pour les applications complexes, et plus récemment dans le domaine de l'embarqué, promeut un développement à base de composants.

Aux prémices de ce paradigme, on retrouve le principe de la séparation des préoccupations déjà énoncé dans les années soixante dix [Parnas, 1972, Dijkstra, 1976], mais ce n'est que bien plus tard qu'un début de consensus semble se dégager [Szypersky *et al.*, 2002], pour caractériser un composant logiciel en tant qu'unité de composition et de déploiement, doté d'interfaces contractuelles. Au delà des différences de définitions entre les modèles, conçus pour les « applications génériques » ou pour celles plus spécifiques de l'embarqué, l'objectif reste le même : augmenter la réutilisabilité et permettre la création de nouvelles entités par assemblage d'entités existantes.

D'une manière générale, un composant est perçu comme une boîte qui abstrait un comportement et est doté de points d'entrée et de sortie explicites (ses interfaces) qui lui permettent de communiquer avec d'autres composants de son environnement. L'idée est alors de n'exprimer – ou de n'abstraire – que le strict nécessaire sur ces interfaces pour caractériser précisément le composant ; il s'agit alors d'être en mesure de le déployer de manière indépendante au sein de contextes applicatifs différents.

Considérant les approches orientées composants, nous faisons la distinction entre les deux points suivants [Bouyssounouse and Sifakis, 2005] :

- Le **modèle de composant** qui spécifie les concepts du langage et les conventions à adopter par le concepteur. En d'autres termes, un tel modèle détermine « l'**espace de conception** » proposé au concepteur pour construire son système.
- Le **canevas à composant** qui désigne l'infrastructure nécessaire à l'étape de conception et d'exécution pour mettre en œuvre le modèle, comme par exemple la gestion des ressources nécessaires aux composants, la prise en charge de leurs assemblages et interactions, etc.

2.1.2 Langages de description d'architecture

Les langages de description d'architectures (ADL, pour « Architecture Description Language ») constituent l'un des aboutissements des travaux sur la notion d'architecture logicielle, présentée comme une discipline à part entière à partir des années 90 [Garlan and Shaw, 1993] en tant que « perspective haut niveau d'un système logiciel ». Cependant, de nombreuses tentatives de définitions du concept d'architecture logicielle ont été proposées[1], sans réel consensus, essentiellement en raison de l'hétérogénéité des domaines d'applications auquel il est appliqué. Medvidovic et Taylor ont alors proposé un cadre de classification et de comparaison des ADL couramment utilisé comme référence dans le domaine [Medvidovic and Taylor, 2000]. Un langage est alors caractérisé pour être un ADL s'il présente les quatre concepts de **composant**, **interface**, **connecteur** et **configuration**.

2.1.2.1 Composant

Le composant est la brique de base d'une architecture logicielle, et correspond à une unité de traitement et/ou de stockage de données. Lorsque l'on parle d'instance, le composant est une entité dynamique dont l'état interne évolue au cours du temps en fonction des stimuli de l'environnement. Il définit des points d'interactions avec son environnement.

Un composant est une unité d'encapsulation. On parle de composant primitif pour caractériser une entité attachée à un comportement décrit dans un langage spécifique (compilé en langage machine ou byte-code, sous forme de code source, de langage formel, ou de langage dédié). Dans certains cas, le comportement se caractérise par une abstraction d'un code source, généralement utilisée en entrée d'un processus d'analyse. Le degré d'encapsulation du comportement d'un primitif (ou granularité des fonctionnalités) varie selon les approches. Il peut alors s'agir d'encapsuler une application complète ou une micro-fonctionnalité élémentaire.

[1] `http ://www.sei.cmu.edu/architecture/definitions.html`

Il n'y a aucun consensus possible quant à considérer la sémantique donnée aux composants, elle dépend fortement du domaine d'application visé par le langage. Par exemple, les « ADL génériques » promeuvent un développement « orienté fonctionnalité », il s'agit alors de manipuler des modules encapsulant du code fonctionnel, alors que les approches proposées dans le domaine du temps réel s'attachent couramment à manipuler des activités concurrentes. Cette distinction est importante car fortement intrusive dans l'esprit du concepteur et relativement à ses choix de conception de l'application.

2.1.2.2 Interface

Cette notion regroupe l'ensemble des points d'interaction (ou ports), au sens large du terme, du composant avec son environnement. Il s'agit généralement de points orientés, par exemple pour qualifier un service offert ou requis, un flot (de contrôle, de données) de sortie ou d'entrée. Une interface détermine généralement un point ouvert à la composition par l'intermédiaire des connecteurs.

Comme nous l'avons déjà mentionné, une motivation importante du paradigme composant est d'être en mesure de réutiliser l'existant, spécifié sous forme de « boîte noire ». Les informations capturées par les interfaces doivent permettre d'atteindre cet objectif. On parle communément de contrats, dont [Beugnard *et al.*, 1999] propose une taxonomie qui identifie quatre niveaux pour les approches composants :

- **basique** : les propriétés syntaxiques, comme par exemple les noms de méthodes ou de paramètres pour qualifier un service, par exemple un langage de définition d'interfaces.
- **comportemental** : permet de spécifier des contraintes sur l'utilisation des points d'interactions spécifiés par les interfaces en tant qu'unités atomiques, comme une méthode par exemple. Elles prennent généralement la forme de pré/post-conditions et d'invariants (assertions booléennes).
- **synchronisation** : il s'agit de propriétés concernant les interactions entre les composants. L'objectif est alors d'abstraire leurs comportements internes en prenant en compte leurs interactions avec l'environnement. On peut décrire leur exécution sous forme de suite d'étapes. Les algèbres de processus [Garavel, 2003] sont parfois utilisés à un tel niveau.
- **quantitatif** : ce niveau est spécifié pour qualifier toutes les propriétés non fonctionnelles, comme par exemple la qualité de service, la gestion des ressources ou encore les échéances temporelles. Bien entendu, de telles propriétés dépendent fortement du domaine d'application ciblé par le langage.

2.1.2.3 Connecteur

Il s'agit d'une entité spécialisée dans les interactions entre composants, par l'intermédiaire de leurs interfaces et qui généralement définit les règles qui permettent la gestion du comportement de ces interactions.
Un connecteur peut prendre des formes très variées selon les approches, comme par exemple une entité de première classe, elle-même réifiée sous forme de composant, et dotée d'interfaces assurant le liant entre les interfaces métier qui participent à l'interaction. À l'inverse, il peut également s'agir d'une simple **liaison**, spécifiant une dépendance entre composants par exemple, mais ne superposant aucun comportement supplémentaire.

2.1.2.4 Configuration

Elle se caractérise par un graphe bipartite composant - connecteur, spécifiant l'architecture du système selon un certain point de vue. En premier lieu, elle exprime un assemblage horizontal de composants de telle sorte à obtenir une sémantique globale par construction. En second lieu, elle peut permettre un assemblage vertical, permettant l'encapsulation, éventuellement à un degré arbitraire. Dans ce cas, on parle communément de **composite** présentant des interfaces à l'environnement en faisant abstraction de son contenu interne.
La sémantique donnée à une configuration peut s'avérer également très différente d'une approche à l'autre et est notamment fonction du style architectural sur lequel le langage repose. Nous pouvons par exemple faire une distinction capitale entre un style de type « pipes and filters » acyclique, généralement employé pour caractériser des flots de données, et un modèle tolérant les cycles et dont les interactions du composant avec son environnement peuvent être « encadrées » par des structures de contrôle arbitraires (boucles, tests).

2.1.2.5 Apports des approches composant et architecture

Les principaux bénéfices attendus de l'utilisation de l'approche architecturale peuvent être résumés selon les points suivants [Déplanche and Faucou, 2005] :

- Tout d'abord, l'architecture constitue un cadre pour maîtriser la complexité d'un système. En effet, il s'agit de raisonner sur ses abstractions sans se soucier des détails de son implantation.
- Elle offre un support pour le prise en considération de l'évolutivité et identifie clairement les opportunités de réutilisation du logiciel.
- Elle constitue une base de raisonnement à des fins d'analyse et de validation (éventuellement à différents niveaux de raffinement et d'abstraction).
- Elle constitue un socle pour la conception et offre une perspective centrale au sein du cycle de développement du logiciel, mais aussi pour l'estimation des coûts et la gestion de projet, la documentation et sa communication.

2.1.3 Une pléthore de propositions

De nombreux modèles de composants et de langages de description d'architectures logicielles ont vu le jour au cours de ces dernières années dans le domaine de l'ingénierie logicielle. On peut citer par exemple : EJB, JAVA BEANS, CCM, COM+, JMX, UML2.0, OSGI, SCA, CCA, SF, FRACTAL, K-COMPONENT, COMET, KILIM, OPENCOM, FUSEJ, JIAZZI, ARTICBEANS, DRACO, WCOMP, CARBON, PLEXUS, SPRING, PACC-PIN, BONOBO, ou encore RAPIDE, WRIGHT, DARWIN, XADL, ARCHJAVA, SOFA.

Conçernant les approches orientées composants pour le domaine de l'embarqué et du temps réel, on peut notamment citer : KOALA [VAN OMMERING *et al.*, 2000], RUBUS [D. ISOVIC, 2002], AUTOCOMP [KRISTIAN SANDSTROM, 2004], PECOS [MÜLLER *et al.*, 2001B], CLARA [DURAND, 1998] METAH [HON, 1998], RTCOM [TEŠANOVIĆ *et al.*, 2004], AADL [SAE, 2004], BIP [BASU *et al.*, 2006], VEST [STANKOVIC, 2001], PECT [WALLNAU, 2003], ROBOCOP [ROB, 2003], SAVECCM [HANSSON *et al.*, 2004], COCONES [BERBERS *et al.*, 2005], EAST-ADL [PROJECT, 2004], ou encore le rapport de synthèse de l'AS CNRS « Composants et Architectures Temps Réel » [Déplanche, 2005].

Bien entendu, l'objectif de l'état de l'art n'est en aucun cas de présenter ces approches de manière exhaustive. Dans la prochaine section, nous nous limitons à n'en présenter qu'un sous-ensemble, mais qui selon nous, permet de donner un aperçu global : il s'agit essentiellement de présenter à quel titre le domaine de l'embarqué est appréhendé dans les approches qui proposent de raisonner en terme d'architecture de composants.

2.2 Approches orientées architecture pour le domaine de l'embarqué

Dans un premier temps, nous présentons les spécifications du modèle de composant FRACTAL et de son implantation THINK pour l'embarqué. Le modèle de composant proposé dans le contexte du projet COMPARE illustre ensuite une volonté d'adapter une infrastructure d'exécution issue de l'ingénierie logicielle pour le domaine du temps réel. À eux trois, PECOS , CLARA et SAVECCM illustrent de manière complémentaire comment les démarches orientées architectures ont été adaptées pour le domaine qui nous intéresse. Enfin, nous présentons AADL et AUTOSAR , deux standards issus de consortiums fédérant de nombreux industriels.

2.2.1 FRACTAL/THINK

Le modèle de composant FRACTAL [Bruneton *et al.*, 2006] réalisé dans le cadre du consortium OBJECTWEB[2] par FRANCE TELECOM R&D et l'INRIA vise à autoriser une définition, une configuration et une reconfiguration dynamique d'une architecture à base de composants. Les spécifications mettent l'accent sur la séparation des préoccupations entre le fonctionnel (définies par le développeur en terme de définition d'interface et d'assemblage de composants) et le contrôle associé à ces composants métier.

Modèle de composant. Les spécifications FRACTAL sont basées sur deux niveaux d'abstraction : Un modèle général pour une description de haut niveau des concepts de composants et un modèle concret, plus restrictif, mais dont les abstractions sont plus proches des langages de programmation, donc plus facilement implantable.
Le modèle général définit la notion de composant en analogie avec une cellule biologique : une cellule est composée d'un plasme (son contenu) et entourée d'une membrane. Le plasme contient d'autres cellules qui sont sous le contrôle de le membrane qui encapsule ces cellules ; cette relation de contenance est récursive. La membrane (appelée également contrôleur dans le modèle concret) d'une cellule incarne le contrôle du comportement propre à cette cellule. Les interactions entre les cellules et leur environnement se font par échanges de signaux sur des points d'accès bien identifiés (les interfaces).
Le modèle concret peut être considéré comme une spécialisation du modèle général sans toutefois définir des caractéristiques propres à un langage de programmation donné. La terminologie utilisée par les spécifications FRACTAL est donnée sur la figure 2.1.

À ce niveau d'abstraction du modèle, la notion de composant et d'interface est défini concrètement. Un composant est composé de deux parties : le contrôleur (correspondant à la notion de membrane) et le contenu (en analogie à la notion de plasme). Le contenu d'un composant est composé d'un ensemble fini d'autres composants (appelés sous-composants) qui sont sous le contrôle du contrôleur de composant qui les encapsule (ou composant composite). Ce modèle de composant est complètement récursif, cette récursivité se terminant par des composant dits primitifs qui ne disposent pas de contenu mais qui encapsulent des entités du langage de programmation sous-jacent (par exemple un objet Java ou un ensemble de fonctions en C). Une originalité des spécifications FRACTAL propose la notion de composant partagé : un tel composant appartient à différents contenus de composites.

Un composant externalise concrètement ses points d'accès avec l'environnement sous forme d'interface[3] spécifiant où des invocations d'opérations peuvent être reçues (interfaces fournies/ser-

[2] `http ://www.objectweb.org`

[3] Une interface doit être différenciée de son type. En effet, le type d'une interface (par exemple au sens Java du terme, c'est-à-dire un ensemble de signatures d'opérations) peut être défini globalement pour différents composants, alors qu'une interface proprement dite (ou une instance d'interface) est associée à un et un seul composant.

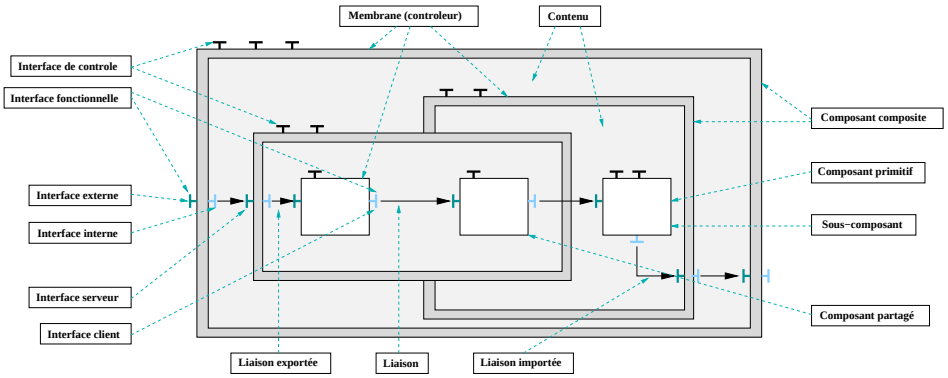


Figure 2.1 – Concepts structuraux et terminologie du modèle de composants FRACTAL.

veurs qui spécifient les services offerts par le composant) ou émises (interfaces requises/clientes qui spécifient les fonctionnalités qu'un composant requiert à son environnement).
Deux catégories d'interfaces sont définies dans les spécifications, (1) les interfaces dites fonctionnelles qui permettent de définir la logique métier attachée à un composant et (2) les interfaces de contrôle (voir figure 2.1) qui externalisent la logique non fonctionnelle propre au composant (comme nous le verrons plus loin, les spécifications proposent un ensemble prédéfini de contrôleurs avec leurs interfaces associées). On distingue aussi les interfaces par leur visibilité : une interface externe est accessible de l'extérieur du composant alors qu'une interface interne n'est accessible que par son contenu.

FRACTAL définit la notion de liaison (orientée) entre deux interfaces : il s'agit d'un lien entre une interface cliente et une interface serveur permettant alors aux composants d'interagir (par l'intermédiaire d'invocations d'opérations spécifiées par les interfaces). Le modèle étant fortement typé, une liaison est considérée comme correcte si le type de l'interface serveur associée à cette liaison est du type ou du sous-type de l'interface cliente.
FRACTAL spécifie deux types de liaisons : les liaisons primitives et les liaisons composites. Une liaison primitive s'établit entre deux interfaces appartenant au même espace d'adressage (typiquement implantée par une liaison de niveau langage de programmation, par exemple une référence vers un objet ou un pointeur en C). Une liaison composite est un chemin de communication entre un nombre arbitraire de composants de liaison. Ces derniers sont des composants spécialisés dédiés à la communication entre les composants qui prennent en charge, par exemple la répartition (*stubs*, *skeletons*, adaptateurs).Des liaisons dites de délégation (liaisons exportées et importées figure 2.1) se définissent également entre les interfaces d'un composite et les composants de son contenu.

Spécification du contrôle. Comme nous l'avons déjà exprimé, le modèle fait une distinction claire entre les aspects fonctionnels et non fonctionnels des composants. La prise en charge du non fonctionnel est assurée par les membranes composées d'un ensemble de contrôleurs qui offrent (par l'intermédiaire des interfaces de contrôle) différents niveaux de contrôle (interception) et d'introspection sur le contenu des composants. La membrane FRACTAL peut-être considérée comme un concept proche de celui de conteneur mais les spécifications ne restreignent pas la prise en charge du contrôle à un ensemble de services techniques donnés, il est donc possible d'exercer un contrôle adapté sur les composants. Ainsi, une membrane peut par exemple :

- fournir une représentation explicite de son contenu (sous-composants).
- intercepter les invocations de méthodes entrantes et sortantes à destination ou originaires de son contenu et superposer différents comportements non fonctionnel (suspendre ou re-

lancer une activité, réifier ou changer les paramètres des opérations invoquées, gérer de manière transparente des services techniques comme la persistance ou la sécurité, la QoS...).

– ou simplement n'avoir aucune capacité de contrôle (dans ce cas la membrane permet uniquement d'encapsuler un ensemble de sous-composant et ainsi de préserver la structure de la composition hiérarchique).

FRACTAL spécifie un ensemble prédéfini d'interfaces de contrôle (leurs API) pour naviguer, introspecter et gérer dynamiquement la configuration et la structure des composants (pour les aspects de reconfiguration dynamique). La découverte des interfaces d'un composant est définie dans l'interface `Component`. L'API de configuration spécifie cinq contrôleurs : le `BindingController` pour naviguer le long des liaisons entre interfaces et créer/détruire dynamiquement ces liaisons, le `ContentController` et `SuperController` pour la gestion du contenu des composites et la structure hiérarchique des composants. L'`AttributeController` pour modifier les attributs d'un composant, le `LifeCycleController` permettant de contrôler le cycle de vie associé au composant qui possède le contrôleur.

Implantations. Il existe de nombreuses implantations du modèle de composants FRACTAL pour différents langages de programmation. La première implantation proposée est celle de référence nommé JULIA [Bruneton *et al.*, 2004]. Nous pouvons également citer FRACTNET [Escoffier and Donsez, 2005], une implantation pour le framework .NET, ou encore AOKELL [Seinturier *et al.*, 2006], PLASMA [Layaïda and Hagimont, 2005], FRACTALK, des mises en œuvre du modèle à l'aide respectivement des langages Java, C++ et SMALLTALK. THINK est une implantation du modèle FRACTAL pour le domaine de l'embarqué.

THINK. THINK [Fassino, 2001, Fassino *et al.*, 2002] est un framework dédié au développement de systèmes d'exploitation pour le domaine de l'embarqué débuté en 2002 par l'INRIA et FRANCE TELECOM R&D. Il est composé d'un ensemble d'outils permettant de gérer la composition des composants et d'une bibliothèque de composants (nommée KORTEX) qui fournit les implantations des principaux services pour construire des systèmes d'exploitation[4]. L'un des objectifs principaux de THINK est d'être le plus flexible possible [Tournier, 2005], le but étant de pouvoir construire des systèmes d'exploitation dédiés et totalement configurables. D'un point de vue structurel, le modèle de composants adopté est celui de FRACTAL, et la bibliothèque KORTEX adopte cette approche de composants le plus « profondément » possible dans les implantations des briques de base des services de l'OS. Ajouté au fait que les définitions des composants sont d'une granularité quelconque (le paradigme composant est omniprésent dans THINK, d'une granularité très fine pour un mutex par exemple, de granularité moyenne pour un pilote de périphérique), cela permet de construire un système d'exploitation sans être contraint par une structure de noyau prédéfini. À ce jour, différentes chaînes d'outils basées sur le canevas FRACTAL-ADL peuvent être utilisées pour compiler une spécification THINK, nous pouvons citer CECILIA [Leclercq *et al.*, 2007, Özcan, 2007] ou NUPTSE (dédiée aux aspects d'optimisation) [Lobry and Polakovic, 2008].

La bibliothèque KORTEX. Elle est définie par un ensemble de composants primitifs implantés en C et en assembleur et de configurations spécifiées avec le langage de description d'architecture (ADL) de FRACTAL. Concrètement, elle offre les implantations pour les composants d'abstraction matérielle (HAL) pour différentes cibles (exceptions, MMU, cache, pilotes de périphériques, etc.) et les implantations des principaux services d'un système d'exploitation (gestion mémoire, thread et ordonnanceur, pile de protocole réseau, systèmes de fichiers, etc.).

[4] `http ://think.objectweb.org`

Résumé - points marquants. D'un point de vue structurel, les implantations de FRACTAL se basent sur des notions communément employées au sein des « modèles de composant génériques » : notions d'interfaces fournies et requises qui matérialisent les dépendances fonctionnelles d'un composant, approche constructiviste pour spécifier la sémantique globale d'un système par assemblage horizontal et vertical (composition hiérarchique) de fonctionnalités, modèle de communication synchrone entre composants, et enfin une approche « orientée conteneur » pour séparer les aspects fonctionnels des aspects de contrôle.
Notons que le composant est une entité déployable et réutilisable aussi bien pour la phase de développement qu'à l'exécution.
THINK présente notamment la particularité de promouvoir une approche où « tout est composant » (y compris les ressources matérielles), permettant un contrôle fin des ressources. De plus, l'exécution d'un système basé sur THINK est autonome (ne nécessitant pas de noyau de système d'exploitation sous-jacent).

Cependant, une architecture FRACTAL se cantonne à décrire des assemblages fonctionnels spécifiés par l'intermédiaire des interfaces des composants, elles-mêmes exclusivement définies sous forme de signatures de méthodes et le modèle de composant se focalise sur les aspects d'introspection et de reconfiguration dynamique. **L'approche ne propose pas de mécanisme pour caratériser les aspects multi-tâches d'une architecture ou pour spécialiser différents modèles de communication et de synchronisation** et ne permet donc pas de raisonner sur ces aspects.

2.2.2 COMPARE

Le modèle de composants COMPARE [ISTCompare, 2005] a été défini dans le cadre du projet IST FP6 (priorité *Embedded Systems*) du même nom. L'objectif est de définir un modèle de composants standard prenant en compte les propriétés temps-réel pour le domaine de l'embarqué.

COMPARE définit un framework appelée ECF (*Embedded Component Framework*). Il se présente comme une extension de *lightweight* CCM (LW-CCM). Sous l'initiative de Thales et Mercury, LW-CCM est un sous-ensemble des spécifications CCM [OMG, 1999] ayant pour objectif de réduire l'empreinte mémoire des applications CCM en y retirant les fonctionnalités qui ne sont pas nécessaires aux applications embarquées. LW-CCM est compatible avec les spécifications Minimum CORBA [OMG, 2002a]. ECF étend LW-CCM selon trois directions :

- la possibilité de définir des schémas d'interaction complexe entre conteneurs,
- la définition d'un modèle de conteneur extensible pour les propriétés non fonctionnelles,
- la spécialisation de ce modèle pour le temps-réel.

Schéma d'interaction. Par défaut, LW-CCM, tout comme CCM, offre deux modes d'interaction entre composants : requête-réponse et envoi d'événements asynchrone avec un mécanisme de publication-abonnement (*publish/subscribe*). Ces mécanismes de base offrent aux développeurs une grande liberté pour implémenter les interactions dont ils ont besoin. Néanmoins, ils ne permettent pas d'isoler clairement le code lié aux interactions du reste de l'implémentation des composants. Pour palier à ce manque, ECF introduit la notion de schéma d'interaction.
Un schéma d'interaction permet de capitaliser une certaine forme de communication entre composants. Ces schémas peuvent être réutilisés indépendamment des composants qui interagissent. Des communications de type envoi de messages asynchrones (par ex. avec des MOM *Message Oriented Middleware*) ou par invocation asynchrone avec récupération de résultat par *polling* ou *callback*) sont des exemples types que les concepteurs de ECF souhaitent pouvoir implémenter sous forme de schémas d'interaction.

Notion de connecteur. Le connecteur est l'élément de ECF qui capitalise la notion de schéma d'interaction. Un connecteur est une entité virtuelle partagée par les différents composants prenant part à l'interaction. Dès lors que les composants sont localisés sur des nœuds physiques distincts, le connecteur acquiert un statut réparti. La figure 2.2 illustre la notion de connecteur.

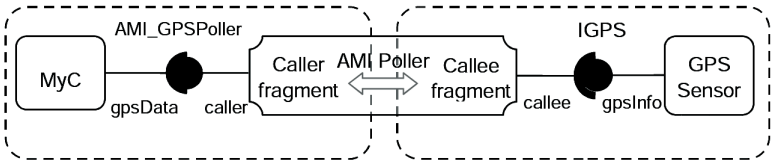


Figure 2.2 – Un connecteur COMPARE.

Un connecteur est mis en œuvre par un ensemble de ports dit ports étendus.

Notion de port étendu. Un port étendu est l'élément qui permet de relier le connecteur à un composant. Il est qualifié d'étendu pour le différencier de la notion de port de réception d'événement qui est déjà présente dans LW-CCM (et CCM). Un port étendu est proche de la notion de port de UML2.
Un port étendu comprend de 0 à n interfaces requises et de 0 à n interfaces fournies. Il s'agit donc de structurer logiquement l'ensemble des interfaces d'un composant en fonction de leur rôle de communication. Un port étendu est connecté localement à un composant.
COMPARE propose une extension du langage IDL3 pour la définition des connecteurs et des ports étendus.

Notion de fragment. Un connecteur est implémenté à l'aide d'un ensemble de fragment. Chaque fragment implémente localement la logique de communication correspondant au comportement global du connecteur. Le fragment constitue ainsi une souche à laquelle le composant est connecté localement via ses ports étendus et qui est le représentant local pour le connecteur.

Modèle de conteneur extensible pour les propriétés non fonctionnelles. Les serveurs d'applications à base de composants comme CCM , EJB ou .NET offrent aux développeurs une structure d'accueil pour l'hébergement de leurs composants. Ceux-ci sont pris en charge par des conteneurs qui sont associés à des services techniques dit non fonctionnels (tels que sécurité, persistance, transaction, etc.). Ces services sont en général figés et ne peuvent ni être changés, ni être remplacés.

Le but du modèle de conteneur extensible de COMPARE est d'offrir la possibilité aux développeurs applicatifs d'ajouter leurs propres services techniques. Pour cela, un ensemble de points d'accroche dans les interactions entre un composant et son conteneur sont définis. Des comportements fournis par le développeur applicatif peuvent être ajoutés autour de ces points.
Ces points d'accroche, au nombre de sept, sont accessibles au travers d'une API. Chaque fois que l'exécution d'un programme LW-CCM atteint l'un de ces points, l'intercepteur enregistré pour ce point est invoqué. Les onze points d'interception sont :

- `On_POA_creation` : correspond à la création d'un POA[5]. Le développeur a ainsi la possibilité de personnaliser les paramètres du POA en cours de création.
- `On_activation` : correspond à la création d'un composant.
- `On_connection` : correspond à la création d'une connexion sur un composant.
- quatre autres événements correspondant à l'interception d'une invocation de méthode : envoi et réception d'une requête, envoi et réception d'une réponse.

[5] *Portable Object Adaptor*.

Spécialisation pour le temps-réel. Le modèle de conteneur extensible précédent est général et peut être utilisé pour n'importe quel type de service non fonctionnel. Dans le cadre de COMPARE , il a été utilisé plus particulièrement pour le temps réel.
Le modèle de temps réel de COMPARE s'appuie sur la notion d'activité. De façon naturelle, une activité COMPARE matérialise la concurrence qui a cours au sein d'une application ou plus généralement d'un système. Elle est issue de l'analyse du problème à modéliser et résulte de choix fait par les concepteurs. Elle constitue également le support sur lequel les paramètres de qualité de service liés au temps-réel sont attachés.
Une telle activité est orthogonale aux composants et peut donc s'étendre sur plusieurs composants. Des activités peuvent être créées et détruites au cours de l'exécution d'une application.
L'intégration de propriétés temps-réel se fait par annotation d'un diagramme d'instances de composant à l'aide d'activités. Chaque interaction entre deux ou plusieurs composants peut être annotée par une ou plusieurs activités. Ces activités seront créées lorsque le flot d'exécution du programme atteint l'interaction concernée.
De manière classique, deux paramètres par activités sont considérés :

- *ready_time* : la date absolue à laquelle l'activité est prête.
- *deadline* : la date relative (à *ready_time*) qui constitue l'échéance de l'activité.

Une projection de ce modèle vers les plates-formes conformes aux spécifications RT-CORBA [OMG, 2003] est définie par COMPARE .
L'analyse d'ordonnançabilité est au cœur de cette projection. Elle vise à garantir qu'en fonction du choix d'un algorithme d'ordonnançabilité et de propriétés temporelles définies pour une application, le système obtenu va pouvoir être exécuté. Néanmoins, cette analyse ne fait pas partie des objectifs du projet. Le but de COMPARE est plutôt de fournir un support permettant de déduire :

- une liste d'interactions à intercepter pour mettre en œuvre les propriétés temporelles,
- une correspondance entre les propriétés temporelles et les priorités RT-CORBA à appliquer à ces points d'interception.

Exécutif. Le modèle ECF (*Embedded Component Framework*) du projet COMPARE fournit une infrastructure système (conteneur, API) et applicative (notion de connecteur, modèle de propriétés temporelles) permettant aux développeurs de définir leurs applications temps-réel à base de composants LW-CCM .

Résumé - points marquants. Le projet COMPARE tient son originalité dans la volonté d'exploiter un modèle de composant issu du génie logiciel, et donc orienté plate-forme de développement, pour la conception d'applications temps réel. Dans ce contexte, la notion de composant est avant tout considérée au niveau de l'infrastructure d'exécution, en tant qu'instance d'un module fonctionnel, et qui présente notamment des capacités d'introspection et de reconfiguration. Les activités ne sont pas considérées comme entités de première classe mais sont transverses à l'architecture. La proposition de réifier la notion de connecteur à l'exécution, et de séparer au sein de sa mise en œuvre les préoccupations relatives au temps réel illustre également la volonté d'exploiter les principes formulés par la « communauté génie logiciel » pour le domaine de l'embarqué.

2.2.3 PECOS

Le modèle de composants PECOS [Müller *et al.*, 2001a] a été défini dans le cadre d'un projet IST du même nom. L'objectif du projet est de fournir un environnement pour la définition, la composition, la configuration et le déploiement de systèmes embarqués à base de composants. Il a pour cible les périphériques industriels tels que les capteurs (température, pression, etc.) ou les actionneurs. Deux projections du modèle existent pour les langages Java et C++.

Modèle de composant. De manière classique, PECOS repose sur les trois entités principales qui sont le composant, le port et le connecteur.

Un **composant** est identifié par un nom et possède des propriétés, des ports et un comportement. Les propriétés permettent d'associer des métadonnées aux composants, comme par exemple des échéances, des périodicités ou temps d'exécution au pire cas. Trois personnalités de composants sont définies : passif, actif et événement. Contrairement au **composant passif**, un **composant actif** est associé à une activité qui lui est propre. Un **composant événément** est un composant actif dont le comportement est déclenché par l'occurrence d'un événement en provenane de l'environnement. Généralement, ce type de composant est utilisé pour modéliser les éléments matériels du système générant périodiquement des événements. PECOS définit un modèle hiérarchique dans lequel des composants composites peuvent contenir d'autres composants. Les composants composites jouent un rôle particulier dans PECOS puisqu'ils définissent des sortes de super-structures pour l'exécution de leurs sous-composants.

Un **port** est identifié par un nom, le type des données qu'il peut transmettre, un intervalle de valeurs légales pour ces données et une direction. Trois directions sont envisagées : entrée, sortie, entrée-sortie. Un port ne peut être connecté qu'à un port acceptant le même type de données et de direction complémentaire. Notons que l'approche proposée par PECOS est fortement dirigée par les données puisque les systèmes sont conçus en terme de valeurs échangées via les ports des composants.

Un **connecteur** est identifié par un nom, le type de données qu'il supporte et une liste de ports qu'il connecte. Les types de ceux-ci doivent être compatibles avec le type du connecteur.

Modèle d'exécution. La sémantique d'un connecteur PECOS s'apparente à celle d'un partage de données. Diffuser de l'information sur un connecteur revient ainsi à écrire une donnée, tandis que récupérer une information sur un connecteur consiste en la lecture d'une donnée. Chaque donnée est définie dans un espace. Selon la nature du composant, la localisation de cet espace varie. Ainsi les composants actifs et événements possèdent leur propre espace de données, tandis que les composants passifs utilisent l'espace de données de leur super-composant (ou plus précisement du premier composant, en remontant dans la hiérarchie d'imbrication, qui en possède un).

La figure 2.3 illustre un assemblage de composants PECOS correspondant à une horloge. Le composant `Clock` fournit l'heure courante qui est affichée par deux composants : sous forme textuelle par le composant `Display` et sous forme graphique par le composant `Digital Display`. Ce dernier utilise la librairie de *widgets* SWT, qui est basée sur une boucle d'événements (clic souris, événement déclenchant le réaffichage des fenêtres, etc.) qui est prise en charge par le composant `EventLoop`. Finalement, `Device` est le composite de plus haut niveau qui représente le système dans son ensemble.
Les composants `Device` et `EventLoop` sont actifs, tandis que les autres sont passifs. Il y a donc deux espaces de données (un pour chaque composant actif) et les composants passifs `Clock`, `Display` et `Digital Display` utilisent l'espace de `Device`. Une connexion entre un composant actif et un composant passif (comme par exemple entre `EventLoop` et `Digital Display`) entraînera la duplication de la valeur associée au connecteur : `EventLoop` écrira une valeur dans son espace d'adressage, tandis que `Digital Display` tentera de la lire dans celui de `Device`. Pour que la communication soit possible, il faut que les espaces de données soit synchronisés. Cette tâche est prise en charge par le modèle d'exécution de PECOS .

Chaque composant actif (ou événement) doit fournir un plan d'exécution pour ses sous-composants et lui-même. Ce plan définit également la priorité et la périodicité de la tâche du composant actif. À titre d'exemple, les plans d'exécution des composants `Device` et `EventLoop` sont les suivants.

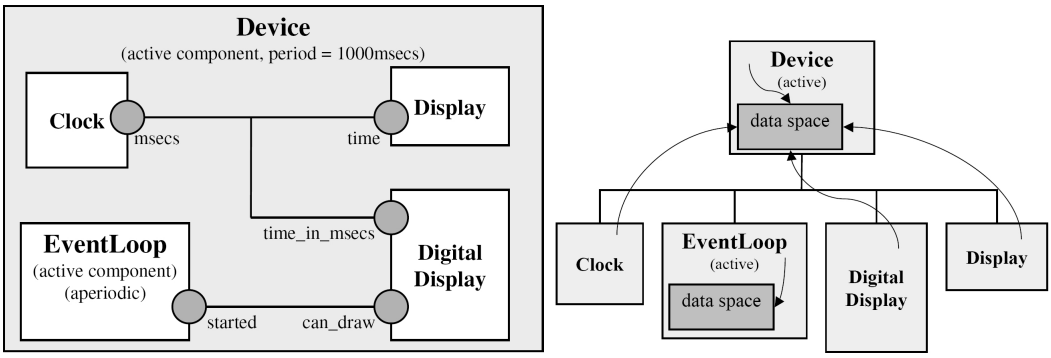


Figure 2.3 – Assemblage de composants PECOS et espaces de données associés.

```
schedule sched of Device
 every 1000 at 10 {
  {
    sync eventLoop;
    exec clock;
    exec display;
    exec digitalDisplay;
  } at 0;
}
```

```
schedule eventTask of Device.eventLoop
 at 5 {
  {
    exec;
  } at 0;
}
```

Dans cet exemple, la priorité (`at 10`) de la tâche `sched` associée au composant `Device` est le double de celle associée au composant `EventLoop` (`at 5`). Il y aura donc une période d'exécution de la tâche `eventTask` pour deux périodes de la tâche `sched`. Les tâches commencent toutes au temps relatif 0 (`at 0`[6]). Le comportement de la tâche `sched` débute par une synchronisation de son espace de données avec celui du composant `EventLoop`), puis se poursuit avec l'exécution des composants passifs `Clock`, `Display` et `Digital Display`. La figure 2.4 illustre une telle exécution.

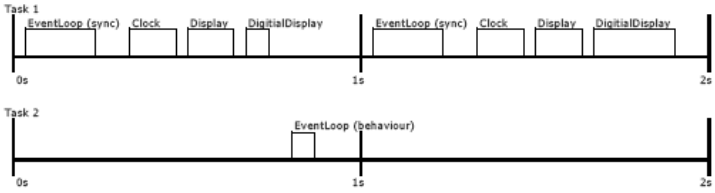


Figure 2.4 – Exécution d'un assemblage de composants PECOS .

Résumé - points marquants. Le modèle de composant PECOS adresse la conception de périphériques industriels de type capteurs/actionneurs (il s'agit de fonctionnalités que l'on peut qualifier de « bas-niveau »). Les interactions entre composants se font exclusivement par lecture/écriture de données typées. Le modèle vise les systèmes temps réel dur, proposant notamment des mécanismes pour capturer précisément les aspects temporels des composants.
L'originalité du modèle de composants PECOS réside dans le rôle dévolu aux composites : ils constituent des conteneurs qui dirigent l'exécution de leurs sous-composants. La politique d'exécution est définie par le développeur qui fournit un plan d'exécution pour chaque composant

[6] La syntaxe choisie par les concepteurs de PECOS utilise malheureusement le même mot clé pour définir la priorité et la date relative de début d'une tâche.

actif (ce plan cadence de façon explicite l'exécution des sous-composants passifs ou actifs). L'intérêt de cette approche réside dans la facilité qu'il y a à composer des politiques d'exécution par assemblage de composants : l'imbrication des activités respecte la hiérarchie de composants. L'inconvénient réside dans le caractère exhaustif de la politique d'exécution : l'ajout d'un nouveau sous-composant dans un composite impose de revoir le plan d'exécution.
La vue composant n'est uniquement utilisée à la phase de développement. Après l'étape de compilation, la structure modélisée est perdue au profit d'un binaire monolithique.

2.2.4 CLARA

CLARA (*Configuration LAnguage for Realtime Applications*) [Durand, 1998, Faucou *et al.*, 2004] est un ADL pour la spécification et l'implantation de systèmes réactifs, temps réel et asynchrones, développé par l'IRCCYN. Il s'agit d'un langage orienté flot de contrôle entre activités concurrentes pour lesquelles il est possible de spécifier des interactions complexes conçernant leurs lois d'activation et politiques de synchronisation.

Présentation du langage. CLARA définit cinq types d'entités (composants) : les **activités**, les **générateurs d'occurrences**, les **ressources** et les **variables partagées** et l'entité de plus haut niveau, le **système**.
Les activités spécifient les entités actives de l'architecture et encapsulent les comportements applicatifs. Elles sont systématiquement attachées à deux ports de contrôle `start` et `end`, respectivement un port d'entrée utilisé pour attacher une loi d'activation à l'activité et un port de sortie pour matérialiser la complétude du traitement lui correspondant. Au cours de son exécution (non réentrante), l'activité peut interagir avec son environnement par l'intermédiaire de ports fonctionnels d'entrée et de sortie pour assurer le transfert de données ou de signaux ou par des ports d'accès à des ressources ou variables partagées. Les générateurs d'occurrences modélisent des sources de données ou de signaux périodiques ou sporadiques susceptibles de stimuler les activités de l'application.
Les liens entre les composants s'établissent entre ports de sortie et d'entrée et se construisent à partir d'un ensemble d'objets de base (qui sont les protocoles, connecteurs et les opérateurs). Sans entrer dans les détails, **ces constructions permettent de spécifier de nombreuses stratégies d'interaction et de synchronisation entre producteurs et consommateurs**, pour des liaisons simples ou des liaisons complexes, c'est-à-dire impliquant de multiples producteurs et/ou consommateurs.
La figure 2.5 illustre un exemple de configuration CLARA issu de [Faucou *et al.*, 2004]. Il s'agit d'un simple contrôleur qui à partir de données (scrutéees périodiquement) en provenance de deux capteurs, effectue un traitement (activité `control`), soumet une donnée à un actionneur et rafraîchit une interface graphique (activité `HMI`). Le symbole « & » caractérise une liaison complexe de type *broadcast*.

Éxécutif. Les travaux menés dans [Faucou, 2002] proposent la construction d'une « vue implémentation d'une architecture opérationnelle », c'est-à-dire la projection des entités de haut niveau de l'ADL CLARA sur un support d'exécution. Une couche intergicielle a été développée au-dessus d'un système d'exploitation OSEK qui implante les spécificités du langage. Les concepts du langage ne sont pas réifiés à l'éxécution, à ce niveau d'abstraction, il s'agit de manipuler des tâches, des variables partagées, des messages, etc.

Résumé - points marquants. Le langage CLARA propose un ensemble d'abstractions pour la conception de systèmes réactifs concurrents – l'objectif étant de permettre au concepteur de se consacrer à l'architecture de son application sans se soucier des détails d'implantation inhérents

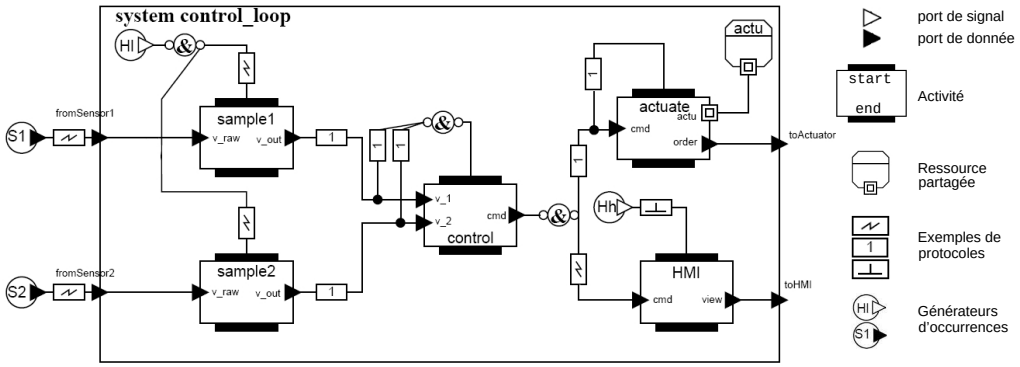


Figure 2.5 – Exemple de configuration CLARA (syntaxe graphique).

à ce domaine applicatif.
Le langage met également l'accent sur les interactions entre les activités en permettant d'en exprimer une grande variété, des plus élémentaires au plus complexes, couvrant alors la majorité des besoins des applications temps réel pour l'échange de données ou de signaux.
Notons également que les aspects de validation temporelle des architectures à un niveau de granularité fin (tenant compte du support logiciel d'exécution comme décrit dans [Faucou, 2002]) est un objectif prioritaire de l'approche.

2.2.5 SAVECCM

SAVECCM [Hansson *et al.*, 2004, Carlson *et al.*, 2006] est un modèle de composant qui a été développé dans le contexte du projet SAVE[7], pour la conception et l'implantation d'applications de contrôle embarquées dans l'automobile. Il s'agit d'un modèle volontairement simple et restrictif (donc peu flexible) dans l'objectif de faciliter l'analyse de contraintes temps réel et de propriétés de sûreté. Il s'agit de développer une méthode de conception et une infrastructure pour applications configurées statiquement pour lesquelles les contraintes temporelles doivent être formellement vérifiées et s'exécutant dans un environnement fortement contraint par les ressources. Les propriétés temporelles qu'il doit être possible de vérifier concernent les temps de réponse de bout-en-bout, la fraîcheur des données (« un composant effectue un calcul sur une donnée échantillonnée il y a moins de 35ms »), la simultanéité (« deux capteurs enregistrent une donnée dans un interval inférieur à 2ms ») et être en mesure de borner la gigue sur les traitements.

Présentation du langage. Trois éléments structuraux sont définis dans SAVECCM , les ***composants***, les ***switches*** et les ***assemblies***. Les composants sont classiquement les entités de première classe qui encapsulent un comportement. Ils exportent des interfaces fonctionnelles nommés **port d'entrée** et **de sortie** : il peut s'agir de port de donnée (type lecteur/écrivain), d'événement (pour contrôler l'activation des composants) ou une combinaison des deux.
Une architecture SAVECCM se spécifie selon le style architectural *pipes & filters* et caractérise des flots de contrôle pour lesquels les exécutions des composants sont déclenchées par des horloges ou des événements externes. Le modèle d'exécution associé aux composants se découpe en trois phases : la lecture de toutes ses entrées (c'est-à-dire données ou occurrences d'événements), l'exécution des traitements internes (dans un temps borné) puis la génération de ses sorties.
Les *switchs* permettent de spécifier au niveau de l'architecture des transferts de données et/ou

[7] *Component Based Design of Safety Critical Vehicular Systems*, `www.mrtc.mdh.se/SAVE`

d'événements de manière conditonnelle. Ils permettent de configurer des assemblages de composants, à l'étape de l'initialisation de l'application ou pour spécifier des modes de fonctionnement au cours de son exécution (un patron de connexion est choisi en fonction des informations disponibles à un instant donné sur les ports d'entrée du *switch* et de gardes basées sur ces entrées).
Récemment [Hakansson, 2007], la notion de **composite** a été introduite. Sémantiquement, il s'agit de rendre atomique l'exécution de l'ensemble de ses sous-composants : les traitements associés à ces derniers ne commencent que lorsque l'ensemble des ports d'entrée du composite sont activés, les occurrences de sortie du composite ne sont générées qu'après leur complétude.
Les *assemblies* offrent simplement un mécanisme pour structurer de manière abstraire une configuration de composants et de *switchs*.
La figure 2.6 donne la syntaxe graphique définie par SAVECCM pour spécifier une configuration. Elle illustre un assemblage `ModeC`, qui caratérise deux modes (alors déterminés par l'environnement) : le premier spécifie que l'algorithme de contrôle `A` doit fonctionner à 10Hz, le deuxième, l'algorithme de contrôle `B` à 100Hz.

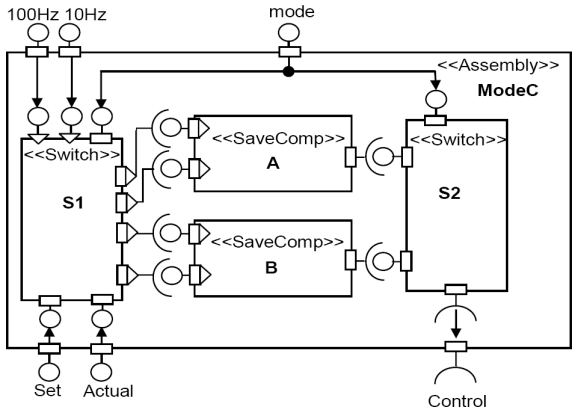


Figure 2.6 – Exemple de configuration SAVECCM (syntaxe graphique).

Résumé - points marquants. SAVECCM définit un langage volontairement peu flexible pour en faciliter les étapes d'analyse, notamment temps réel. Il se démarque des autres ADLs dans la proposition d'exprimer par l'intermédiaire d'une entité de première classe (avec l'élément *switch*) la configuration de l'architecture (à l'étape de la configuration ou dynamiquement).
Le modèle de composant SAVECCM n'est utilisé qu'à l'étape de la modélisation. Il adresse des applications fortement contraintes par le temps, définies statiquement, la prise en charge de l'achitecture à l'exécution n'est pas l'objectif du modèle.

2.2.6 AADL

AADL (*Architecture Analysis and Design Language*) est un ADL initialement dédié aux systèmes avioniques et basé sur la modélisation d'applications temps réel embarquées critiques [Feiler *et al.*, 2004, SAE, 2004]. Il est issu d'une première initiative, METAH [Hon, 1998], proposée par Honeywell au début des années 90. Il est actuellement développé et standardisé par le SAE[8] et supporté par un large ensemble de partenaires industriels (Lockheed Martin, EADS, etc).

Les applications visées sont fortement contraintes par les ressources, le temps et les interactions matériel/logiciel. Le langage est utilisé pour décrire la structure d'une application sous forme

[8] *Society for Automotive Engineers.*

d'un assemblage de composants logiciels et matériels. Les composants interagissent par l'intermédiaire d'interfaces dites fonctionnelles qui décrivent des flots de contrôle et de données. La spécification d'aspects non fonctionnels importants tels que les exigences temporelles, les fautes et erreurs, le partitionnement temporel et spatial et les propriétés de sûreté et de certification sont pris en compte.

Présentation du modèle. Les trois concepts principaux définis au sein du modèle de composant sont ceux de « **type de composant** », « d'**implantation de composant** » et de « **catégorie de composant** ».
Un « **type** » définit les interfaces fonctionnelles du composant, elles sont caractérisées par des points d'interactions de contrôle et de données, et par un ensemble de propriétés spécifiées sous forme d'attributs. À ce niveau d'abstraction, il est également possible de décrire le comportement observable du composant en précisant les flots de contrôle susceptibles de le traverser.
Une « **implantation** » d'un certain « type » précise la structure interne pour ce composant sous forme d'une configuration de sous-composants. Une telle configuration décrit la façon dont ces sous-composants échangent des données et la façon dont les flots de contrôle s'écoulent parmi eux.
En plus d'être typé, chaque composant appartient à une « **catégorie** » prédéfinie qui représente la nature des éléments manipulés (dix au total). Ainsi, un composant peut être considéré comme une abstraction de la plate-forme d'exécution : une `mémoire`, un `processeur`, un `bus` ou encore un `périphérique`, ou une entité logicielle : une `donnée` (une zone de donnée statique partagée), un `sous-programme` (un bloc de code séquentiel), un `thread` ou un `processus` (modélisant une entité de partitionnement spatial de mémoire). Enfin, le concept de `système` est proposé : il permet d'encapsuler des configurations de composants et notamment de spécifier l'allocation des composants logiciels sur les composants matériels. Un `système` peut définir différents modes de fonctionnement, chacun pouvant être associé à la configuration d'un assemblage particulier (en terme de composants et de leurs connexions). C'est le composant au sommet de la hiérarchie.

Une configuration permet de spécifier les connexions entre composants par l'intermédiaire de leurs points d'interactions, nommés `ports`. Il s'agit de points d'entrée et/ou de sortie d'un composant. Trois modèles de communications sont proposés au sein du langage : par événements (type action/réaction), par données (lecture/écriture), ou une combinaison de ces deux sémantiques (alors proche d'un envoi de message).
À partir d'une telle configuration, il s'agit également de spécifier comment les composants logiciels sont alloués sur les composants matériels. À titre d'illustration, la figure 2.7 présente la notation graphique d'une configuration d'un `processus` AADL contenant cinq `threads` alloués sur une machine multi-processeur à mémoire partagée.

Il est possible d'associer des propriétés à chaque élément de la description architecturale, notamment pour caractériser certains aspects d'ordre non fonctionnel. Pour chaque catégorie de composant, une bibliothèque de telles propriétés est prédéfinie. Par exemple, les `threads` disposent de propriétés temps-réel telles que la période (caractérisée sur la configuration donnée figure 2.7), l'échéance ou la durée d'exécution. Le langage propose également un mécanisme d'extension permettant au concepteur de définir les propriétés relatives à son domaine d'application et de contraindre leurs utilisations à une catégorie de composant spécifique.

Extensibilité. Une attention particulière a été portée sur la possibilité d'étendre le cœur du langage normalisé par le SAE, par l'intermédiaire d'`annexes`. Ces dernières permettent d'encapsuler les spécifications étendues qui seront alors fusionnées avec celles du standard. Elles permettent au concepteur d'enrichir ses modèles en fournissant des informations additionnelles structurées conformément à ces annexes. On peut notamment citer [Vestal, 2005] caractérisant un modèle d'erreur ou [França *et al.*, 2007] pour décrire le comportement d'une spécification AADL.

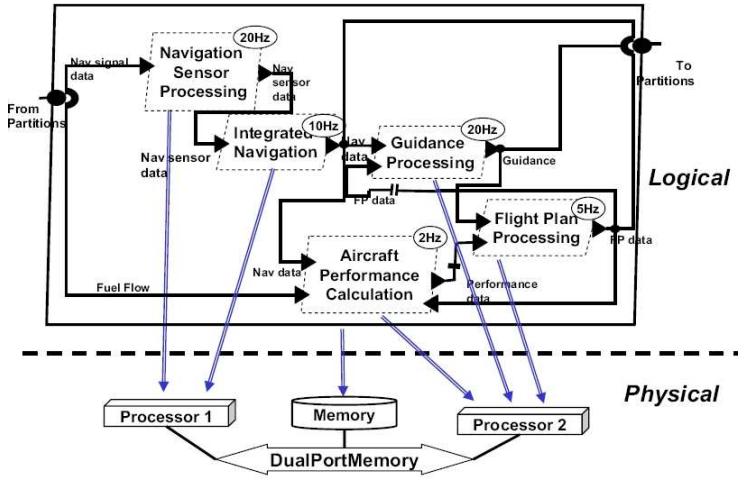


Figure 2.7 – Exemple de spécification d'une configuration AADL (notations graphiques).

Exécutif. Les spécifications donnent peu d'information sur le support d'un exécutif pour les applications spécifiées en AADL si ce n'est la génération d'une enveloppe spécifique (ou couche intergiciel) entre les implantations des composants écrit dans un langage de programmation impératif (en C ou en Ada – des règles de projection doivent alors être respectées entre les spécifications architecturales et l'implantation) et un système d'exploitation temps réel standard. Au sein de la suite d'outils OCARINA [HUGUES *et al.*, 2007] est implanté un compilateur permettant de générer un code source compilable pour applications AADL réparties.

Outils de conception. D'un point de vue syntaxique, AADL a été défini en premier lieu en tant que langage purement textuel, mais plus récemment s'est concrétisée la volonté d'exploiter des environnements de modélisation de plus haut niveau, notamment basé sur EMF [(SAE), 2005] ou UML (pour lequel un profil a été défini). La suite d'outils de référence nommée OSATE[9] a été développée pour l'environnement ECLIPSE qui fournit notamment des éditeurs textuel et graphique et des outils de vérification pour assurer la cohérence des spécifications architecturales.

Résumé - points marquants. Un atout indéniable d'AADL est d'avoir fédéré un ensemble d'industriels autour de ce standard, montrant notamment l'intérêt porté par l'industrie en faveur de démarche de modélisation orientée architecture pour le domaine du temps réel.
L'originalité d'AADL réside dans sa considération de plusieurs sémantiques au niveau du composant, notamment au travers de la description des éléments matériels rarement considérés comme entités de première classe dans les ADLs au même titre que les composants logiciels.
L'espace de conception défini par AADL (c'est-à-dire les concepts proposés pour structurer son applicatif) est similaire à celui couramment proposé au sein des ADLs pour l'embarqué : une architecture d'entités actives très orientée flot de donnée et de contrôle.
AADL est un langage très orienté mise en œuvre, et que l'on peut qualifier de « bas-niveau ». En effet, les entités manipulées sont proches des abstractions fournies généralement par le système d'exploitation (processus, threads, etc).

[9] `http ://la.sei.cmu.edu/aadl-wiki/`

2.2.7 AUTOSAR

AUTOSAR (*AUTomotive Open System ARchitecture*) est un consortium fondé en 2003 à l'initiative d'acteurs majeurs de l'industrie automobile [Fennel *et al.*, 2006, Heinecke *et al.*, 2006]. L'objectif est de faciliter l'intégration et la réutilisation de composants logiciels et matériels entre plates-formes de différents constructeurs et équipementiers du domaine. Il s'agit notamment de développer et d'établir un standard industriel ouvert pour les architectures électriques/électroniques dans l'automobile.

Constatant la complexité des architectures utilisées dans l'automobile (réseaux d'ECU[10] et de capteurs/actionneurs interconnectés) croissant de manière exponentielle et l'hétérogénéité des fonctionnalités logicielles qui y sont embarquées, l'objectif d'AUTOSAR est de définir une méthodologie adaptée et standardisée par un consortium regroupant plus d'une centaine d'entreprises de par le monde. Le standard se concentre essentiellement sur les aspects de modularité, de configurabilité et de portabilité de modules logiciels et sur la standardisation de leurs interfaces. Il s'agit d'un développement « orienté fonctionnalités » : les entités de première classe utilisées par le concepteur pour spécifier son application sont des modules encapsulant du code fonctionnel.
La version 2.0 des spécifications AUTOSAR (rendue publique en 2006) étant composée de plus de quatre vingt dix livrables, il est bien entendu impossible de présenter de manière exhaustive son étendue[11].

Un modèle en couche. AUTOSAR a adopté un modèle en couche pour la standardisation de différents niveaux d'abstraction, de l'application jusqu'aux services de bas-niveau au-dessus de la plate-forme matérielle. D'après les auteurs, cette approche est nouvelle dans le contexte des applications développées par l'industrie automobile. De manière relativement classique, trois couches ont été définies, elles sont représentées par la figure 2.8 : une couche applicative (AUTOSAR *software*), l'environnement d'exécution (*Runtime Environment*) et une couche regroupant les services de plus bas-niveaux (*Basic Software*).

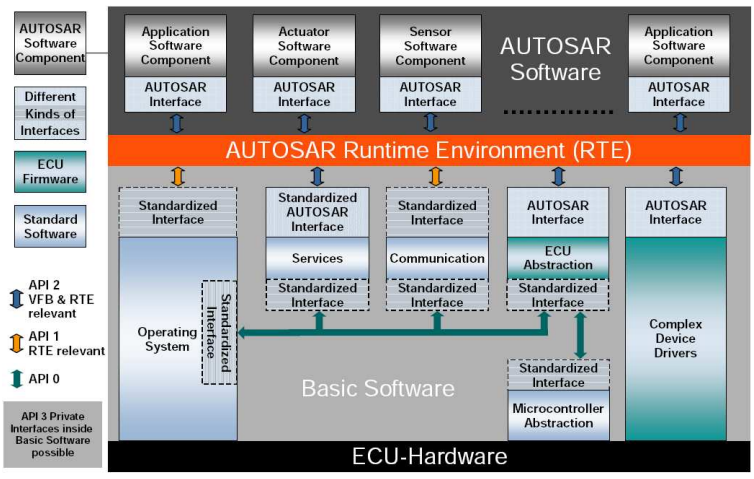


Figure 2.8 – Abstractions AUTOSAR , du matériel à l'applicatif.

Les spécifications définissent le VFB (*Virtual Functional Bus*), une couche d'abstraction permettant de séparer les applications (définies par un ensemble de composants interconnectés) et l'infrastructure d'exécution qui leur est nécessaire. Les composants applicatifs peuvent être répartis

[10] *Electronic Control Unit.*
[11] Les spécifications complètes sont consultables à partir du site `www.autosar.org`.

sur différentes ressources de calcul, cette couche fournissant alors les mécanismes de communication et les services standardisés pour ces composants. Elle permet notamment d'abstraire les implantations des composants applicatifs de l'infrastructure matérielle sous-jacente et de leur répartition. L'ensemble des environnements d'exécution (RTE, notamment représenté sur la figure 2.8) disposés sur les différents ECU sont alors considérés comme une implantation du VFB.
La couche d'abstraction la plus basse est le BSW (*Basic Software*), elle fournit les services d'ordre non fonctionnels (dans la terminoligie AUTOSAR). Une séparation claire des fonctionnalités est déterminée en fonction de différents critères (services indépendants ou non de la plate-forme matérielle sous-jacente, modules aux interfaces standardisables ou au contraire propriétaires, interfaces avec le système d'exploitation, etc).

Présentation du modèle applicatif. Une application AUTOSAR consiste en un ensemble de composants logiciels (*SoftWare Components* – SWC) interconnectés [GdR, 2006]. Trois niveaux de descriptions sont pris en compte au sein des spécifications des SWC :
(1) Le premier niveau, le plus abstrait, pour lequel les composants sont décrits au moyen de types de données et d'interfaces, de ports et de connexions entre eux, et de composants hiérarchiques. Plus précisément, les entités de premières classes sont les `AtomicSoftwareComponent` qui permettent l'encapsulation de fonctionnalités insécables. L'assemblage des fonctionnalités se fait par l'intermédiaire d'interfaces de type client/serveur (mode synchrone) ou de type émetteur/consommateur (mode asynchrone avec mémorisation). Les liens de dépendances entre les composants atomiques et les services élémentaires fournis par l'exécutif sous-jacent doivent être explicités par des interfaces annotées (un attribut nommé « *isService* »). Classiquement, la composition s'établit horizontalement par l'intermédiaire de connecteurs d'assemblage ou verticalement par connecteurs de délégation (au sein de structures composites).
(2) Au deuxième niveau, il s'agit de spécifier les interactions entre les SWC et le RTE (*Run Time Environment*) sous-jacent : il s'agit de préciser les entités exécutables internes aux composants atomiques (appelées *runnables*), de spécifier des « attributs de communications » (par exemple, la taille d'un tampon pour une connexion asynchrone) et de détailler certains comportements contraints par le temps. À ce niveau de description, les spécifications permettent de détailler finement les relations de concurrence entre les *runnables* (c'est-à-dire leurs accès aux données partagées) « encapsulés » au sein d'un composant atomique.
(3) Enfin, le dernier niveau conçerne la description concrète des implantations des composants atomiques (sous forme de code source ou compilé) permettant leurs déploiements et la caractérisation de leur consommation en ressources.

Exécutif. Les spécifications AUTOSAR définissent la méthodologie à adopter pour déployer un assemblage de SWCs. Il s'agit alors de configurer et de générer l'intergiciel (le RTE) à partir des *Software Component Descriptions* (qui regroupent l'ensemble des spécifications détaillées ci-dessus) et des *ECU Resource Descriptions* décrivant les ressources matérielles disponibles. Deux générateurs interviennent, l'un pour définir un placement des SWCs parmi les différents ECUs selon leurs liens de communication et besoins en ressources, l'autre pour générer concrètement le RTE sur chaque ECU, au-dessus d'un système d'exploitation qui implante les spécifications OSEK. À notre connaissance, il n'existe pour le moment pas d'implantation fonctionnelle de ce processus de génération.

Résumé - points marquants. Comme AADL, le consortium créé autour d'AUTOSAR rassemble de nombreux industriels de l'automobile témoignant d'une forte volonté et de la nécessité de partager un standard commun au sein des différents constructeurs et équipementiers du domaine.
Un aspect déterminant qui qualifie la « philosophie AUTOSAR » est la volonté de promouvoir une démarche de conception orientée fonctionnalités, l'applicatif embarqué étant avant tout considéré en tant qu'architecture fonctionnelle plutôt qu'opérationnelle, témoignant de l'introduction

de plus en plus massive du logiciel au sein du secteur automobile.
Un aspect intéressant de l'approche est également de considérer l'effort de standardisation d'un modèle en couche à différents niveaux d'abstraction, pratique nouvelle au sein de cette communauté industrielle. Cette architecture n'a pas encore fait ses preuves dans le domaine de l'automobile [de Oliveira, 2005] mais elle est l'orientation la plus forte pour le développement futur et son impact sur le dimensionnement temps réel est en cours d'évaluation.

2.3 Synthèse et positionnement

2.3.1 Espace de conception

Prise en charge des activités. La majorité des démarches orientées architectures pour le domaine du temps réel et de l'embarqué se focalisent sur une spécification opérationnelle de l'application. C'est en effet le concept d'activité qui est manipulé en tant qu'entité de première classe, comme c'est le cas de CLARA , SAVECCM ou d'AADL. Dans ces démarches, l'architecture modélise alors un assemblage de tâches concurrentes.
Selon nous, cette approche est réductrice et trop intrusive dans l'espace de conception dans le sens où elle ne permet pas de promouvoir une démarche constructiviste au regard des fonctionnalités métiers. Nous pensons que **les modules fonctionnels doivent constituer les entités de première classe** sur lesquelles doit être basée la réutilisabilité, la maintenabilité et dans une autre mesure les aspects de reconfiguration dynamique.
En premier lieu, il s'agit donc de raisonner sur une architecture fonctionnelle, au centre du cycle de conception, puis de manière découplée, d'exprimer les aspects multi-tâches.
Cette démarche est implicitement celle adoptée au sein des modèles de composants « issus du génie logiciel » (ou « modèles de composants génériques ») comme dans le contexte de FRACTAL ou de CCM : dans ces deux cas, l'architecture (et l'infrastructure) ne présuppose rien quant à une disposition des tâches sur le métier, ces aspects doivent être pris éventuellement en charge par le comportement interne du métier, donc par le développeur.
En contrepartie, nous pensons que ce contrôle ne doit pas être pris en charge par le code métier (comme c'est le cas des modèles proposés par CLARA , SAVECCM et AADL). Il s'agit en effet d'aspects d'ordre non fonctionnels qui n'ont pas à s'entremêler à ce dernier, qui doivent être pris en charge par l'infrastructure d'exécution et qui ne doivent pas être « enfouis » dans l'implantation mais au contraire extériorisés au niveau de l'architecture pour être en mesure de raisonner sur cette dimension dynamique.
Il s'agit donc de proposer un mécanisme pour **représenter la dynamique propre aux aspects multi-tâche au niveau de l'architecture**. La proposition d'AUTOSAR et de COMPARE vont d'une certaine manière dans ce sens, respectivement avec les notions de *runnable* (entité de concurrence atomique) et d'activité susceptible de traverser différents composants de l'architecture.

Sémantique des interactions. La prise en charge d'assemblage de fonctionnalités à des granularités arbitraires passe par la nécessité d'exprimer des dépendances fonctionnelles entre les différentes entités architecturales (modélisées par des interfaces requises/fournies comme dans le cas de FRACTAL, LW-CCM ou AUTOSAR). Les approches exclusivement orientées flot de donnée et de contrôle telles que CLARA et SAVECCM ne disposent naturellement pas d'une telle capacité d'expression (car basées sur un style architectural de type « *pipes & filters* »). Il est intéressant de noter que cette notion n'est pas non plus présente dans AADL mais que des travaux [Hamid *et al.*, 2005] ont été proposés pour répondre à ces besoins non initialement pris en charge.
Dans le contexte des applications embarquées, multi-tâches coopérantes, il est nécessaire de proposer des modèles de communications de type flot de contrôle et de données. À ce titre,

il faut relever qu'aucune approche présentée dans l'état de l'art ne considère la possibilité de faire cohabiter ces modèles avec un mode d'interaction de type requis/fourni exprimant les dépendances fonctionnelles.

Modèle d'exécution. Un point important de comparaison des approches concerne le modèle d'exécution attaché aux entités manipulées (cet aspect dépend notamment de la manière dont les aspects liés à la concurrence sont appréhendés au niveau de l'architecture).
PECOS utilise la notion de composite pour spécifier de manière statique l'ordonnancement de ses sous-composants (il s'agit d'une approche synchrone, l'ordonnancement est cadencé par une horloge globale), qui à chaque exécution, écrivent et/ou lisent des données sur les espaces correspondant à leurs ports. Une activité CLARA s'active par l'intermédiaire de son interface de contrôle *start* (elle est non ré-entrante) et est susceptible de se synchroniser en fonction des protocoles spécifiés sur les ports qu'elle exporte. Les composants SAVECCM sont exclusivement attachés à un modèle d'exécution plus basique, de type « consommations des occurrences d'entrée / traitements internes / productions des occurrences de sortie ». AADL se base essentiellement sur la spécification de threads périodiques et AUTOSAR sur un ensemble d'événements définis au niveau du RTE sous-jacent. Dans ce dernier cas, c'est alors au concepteur de modéliser l'ensemble des interactions entre certains de ces événements et ses *runnables* applicatifs.
Ces aspects sont importants dans le sens où ils contraignent fortement l'espace de conception manipulable par le concepteur de l'application.

Il en est de même concernant le degré d'encapsulation d'un composant relativement à son comportement interne. Par exemple, une activité CLARA , un composant SAVECCM ou PECOS ne peuvent être attachés qu'à une unique séquence de comportement, contrairement à FRACTAL ou LW-CCM dont les composants peuvent implanter un nombre arbitraire de méthodes sur leurs interfaces fournies, ou à AUTOSAR dont un SWC peut spécifier de multiples *runnables*.
De telle sorte à ne pas contraindre le concepteur, et lui laisser le choix quant à la granularité des fonctionnalités encapsulées au sein d'un composant, il semble important de considérer cet aspect.

Composition hiérarchique. D'un point de vue conceptuel, la prise en charge de la composition hiérarchique est une propriété intéressante permettant de manipuler en tant qu'entité de première classe une configuration architecturale. Cependant, la sémantique d'une telle composition dépend bien entendu des abstractions proposées par le langage d'architecture. Dans PECOS , il s'agit de spécifier un séquencement des sous-composants, au sein de SAVECCM (composite) et de CLARA (qui propose la notion d'activité composée), il s'agit d'une « encapsulation du contrôle » dans le sens où la composition possède (au vu de l'environnement) la même sémantique que les composants (ou activités) primitives. En effet, dans ces deux approches, l'exécution d'un composite n'est fonction que des stimuli de l'environnement agissant sur les interfaces externes de ce composite.
De par les différents composants définis dans AADL, la composition hiérarchique est utilisée pour spécifier des assemblages valides de composants en fonction de leur catégorie (par exemple un `processus` peut être composé par un ensemble de `threads` mais non l'inverse).
FRACTAL et AUTOSAR caractérisent la composition hiérarchique en tant qu'assemblages de fonctionnalités alors qu'une telle construction n'est pas supportée dans LW-CCM .

Dans les approches présentées, seule FRACTAL (en dehors de PECOS) laisse une « porte ouverte » quant à considérer le composite comme une potentielle entité qui superpose un contrôle spécifique à ses sous-composants. En effet, un composite peut être attaché à une membrane exerçant alors un contrôle a priori arbitraire sur son assemblage sous-jacent.

Reconfiguration dynamique. Dans le domaine des applications temps réel, pour lesquelles le respect de contraintes temporelles est un paramètre crucial, il n'est pas concevable d'imaginer des possibilités de reconfiguration dynamique d'une application sans qu'elles n'aient été prédéfinies a priori de l'exécution, c'est-à-dire statiquement. Il s'agit de la notion de « mode de fonctionnement » : dans les approches dirigées par l'architecture, il s'agit généralement de spécifier différentes configurations de composants et d'assurer le changement d'état de l'une à l'autre de ces configurations en fonction de paramètres et d'événements déterminés. Cette méthode est notamment employée dans AADL, SAVECCM (avec le concept de *switch*) et dans une moindre mesure dans AUTOSAR (dans ce-dernier, la notion de mode est interne à un SWC).
Les besoins en reconfiguration dynamique et son support à l'exécution sont des facteurs déterminants qui ont conduit la spécification du modèle FRACTAL. Cependant, il n'est pas du ressort du modèle de chercher à « cadrer » a priori les possibilités de reconfiguration. Une telle démarche est par exemple proposée dans [David, 2005].

Propriétés attachées aux entités architecturales. Un langage de description d'architecture s'accompagne généralement d'un ensemble de propriétés (au sens large, par exemple d'un contrat spécifié sous forme de langage dédié à une simple annotation) qui s'attachent aux entités de première classe définies par ce langage. Conceptuellement, de telles propriétés s'utilisent dans deux directions : dans le sens « *bottom-up* », il s'agit alors d'abstraire au niveau architectural des informations enfouies dans le comportement interne du composant (généralement spécifié par un langage de programmation), ou bien dans le sens « *top-down* », dans ce cas, il s'agit de proposer au concepteur de caractériser certains aspects de son architecture qui doivent être pris en charge par l'infrastructure d'exécution et non par son code métier.

Parmi les « propriétés *bottom-up* », on retrouve par exemple une abstraction du comportement de l'implantation interne des composants. Avec CLARA , il s'agit d'exprimer la séquence d'opérations effectuées sur les ports d'entrée et de sortie lorsque l'activité est dans l'état actif. AADL définit la notion de flot d'exécution qui s'écoule le long de l'architecture. La caractérisation de temps d'exécution de blocs de code est un autre exemple de telles propriétés.
Elles permettent généralement de faciliter certaines analyses menées au niveau des abstractions architecturales. Plutôt que de travailler à l'échelle de l'implantation, l'objectif consiste à abstraire les propriétés pertinentes et ainsi réduire l'espace d'exploration du processus d'analyse.

Les « propriétés *top-down* » capturent généralement les besoins du domaine visé et font partie intégrante de l'étape de conception de l'application (par exemple, spécifier la période d'une activité). Le concepteur utilise cette capacité d'expression pour construire son architecture et au sein d'une démarche outillée, l'objectif est de prendre automatiquement en charge ces propriétés à la phase de compilation. Sans entrer dans les détails, toutes les approches supportent à différents degrés cette démarche.

2.3.2 Support à l'exécution

Concepts architecturaux manipulables à l'exécution. Hormis les implantations FRACTAL (et dans une moindre mesure dans COMPARE , basé sur LW-CCM), **les approches présentées ne s'attachent pas à rendre concret les concepts architecturaux de haut niveau à l'exécution**, ils ne servent alors qu'à l'étape de la modélisation de l'application, il s'agit d'une représentation logique de la structure de l'application. Cette structure est alors perdue après l'étape de compilation. Il faut noter que dans le cas des implantations FRACTAL, le concept de composite est également réifié à l'exécution et ce, quelle que soit la profondeur de l'encapsulation.

Paradigme composant à plusieurs niveaux d'abstraction. Dans les approches présentées dans la section précédente, aucune ne s'intéresse à mettre en œuvre un support d'exécution lui-même implanté sous forme d'assemblages de composants. Pour cela, il faut bien entendu que les concepts de haut niveau proposés par l'approche soient suffisamment riches pour concevoir différents niveaux de fonctionnalités.
Cependant, cette volonté d'exploiter le paradigme composant pour la mise en œuvre d'infrastructures spécialisées se retrouve dans différents travaux. Pour la gestion des communications réparties, nous pouvons citer les canevas DREAM [Quéma, 2005] pour l'implantation d'intergiciels JAVA ou dans une moindre mesure au sein de POLYORB (notamment adapté à la construction d'intergiciels temps réel [Hugues, 2005, Vergnaud, 2007]).
D'autres démarches se focalisent aussi sur l'implantation de services d'ordre non fonctionnels sous forme d'assemblage de composants : SAIA [DeAntoni, 2007], basée sur la notion de connecteur pour abstraire les entrées/sorties un applicatif temps réel et la plate-forme d'exécution, [Polakovic *et al.*, 2006] pour la gestion de la reconfiguration dynamique pour les systèmes d'exploitation conçus avec THINK, ou encore [Seinturier *et al.*, 2006] proposant l'ingénierie des membranes FRACTAL sous forme d'assemblages de composants.

2.3.3 Importance d'un support outillé

Les langages de description d'architecture proposent des concepts de haut niveau pour spécifier les applications. De plus, la perspective architecturale est généralement au centre du processus de conception du logiciel. Une telle démarche doit nécessairement s'accompagner de nombreux outils d'aide à la conception, de vérification et de validation, de compilation, de mise au point, etc.
Il ne s'agit pas ici de détailler l'ensemble des outils proposés par les approches que nous avons présentées mais de simplement mettre en avant ce besoin fondamental au sein de telles démarches de conception.

Au sein de ce chapitre, différentes approches de conception pour le domaine de l'embarqué et centrées sur l'architecture ont été présentées. Il s'agissait ensuite d'en proposer une synthèse et d'énoncer les principes essentiels du travail de thèse. C'est à partir de ces derniers que nous introduisons dans le chapitre suivant la présentation générale de notre contribution.

Chapitre 3

Présentation générale

Sommaire

3.1 Proposition **29**
3.1.1 Définition de l'espace de conception TINAP 30
3.1.2 Patron de « conception/programmation orientée membrane » 31
3.1.3 Le paradigme composant à différents niveaux d'abstraction 32
3.1.4 Cadre méthodologique 33
3.2 Organisation de la contribution **34**

3.1 Proposition

Cette thèse s'articule autour de TINAP[12], dont les objectifs principaux peuvent se résumer par les points suivants (détaillés dans la suite de cette section) :

- La définition d'un espace de conception : il s'agit de spécifier un modèle – ou profil – de composant de haut niveau pour concevoir des applications multi-tâches à « contraintes temps réel embarquées » et d'assurer son support à l'exécution.
- L'utilisation du patron de « conception/programmation orientée membrane » pour implanter l'infrastructure d'exécution nécessaire à la mise en œuvre du modèle de composants.
- La volonté d'expérimenter le paradigme composant à différents niveaux d'abstraction, c'est-à-dire pour concevoir un système dans sa globalité.
- Ancrer notre proposition au sein d'un cadre méthodologique « dirigé par les modèles » et proposer des outils de « haut niveau » d'aide à la conception.

[12] **T**INAP **I**S **N**OT$_{only}$ **A** **P**ROFILE.

3.1.1 Définition de l'espace de conception TINAP

Domaine d'application visé. Dans le contexte de cette thèse, nous nous sommes intéressés à la conception d'applications qualifiées de « temps réel embarquées ». Cependant, cette dénomination est à considérer avec retenue. En effet, dans la littérature, elle demeure couramment employée pour caractériser pléthores de classes de systèmes dont les besoins et contraintes fonctionnels sont très variés et issus de communautés parfois bien différentes.
Dans notre cas, nous considérons des applications conçues dans un environnement multi-tâches, asynchrone, à contraintes temporelles souples, non critiques. Dans ce contexte, même si nous considérons que les aspects multi-tâches et temporels font partie intégrante de la logique métier des systèmes ciblés, notre approche a été conduite par la volonté d'adresser des applications dont la partie opérationnelle est prépondérante par rapport aux aspects de contrôle[13].

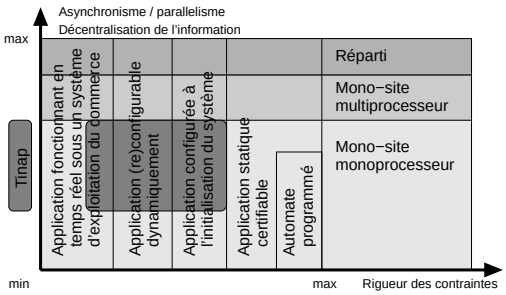


Figure 3.1 – Domaine d'application.

Contrairement à des applications « génériques », celles que nous visons présentent également de forts besoins en terme de synchronisation et de modèles de communications spécifiques. Elles se caractérisent également par de fortes dépendances avec les services du système d'exploitation sous-jacent et par un comportement déterministe. Dans le contexte de cette thèse, nous nous sommes restreint à la conception d'application centralisée sur monoprocesseur, les aspects liés à la répartition n'ont pour le moment pas été adressés. La figure 3.1 présente le référentiel de classes d'applications proposé dans [Trialog, 1992], et permet de situer le domaine visé par TINAP (ce positionnement sera détaillé par la suite).

Le paradigme composant. Selon nous, le premier intérêt de ce paradigme est de considérer qu'il est fortement intrusif dans l'esprit du concepteur, il oblige à « penser architecture », et donc à modulariser et à factoriser au maximum l'ensemble des fonctionnalités en amont du cycle de conception. L'objectif est également de réifier au niveau architectural les flots d'informations que les composants sont susceptibles de s'échanger et d'externaliser certaines propriétés internes sous forme abstraite (éventuellement non fonctionnelles) pertinentes pour le métier visé.
Ces aspects tendent à faciliter grandement la compréhension du fonctionnement de l'application mais aussi son implantation, sa maintenance, sa réutilisabilité et la capacité de raisonner sur les propriétés qu'externalise l'architecture. Ces éléments poussent naturellement à considérer l'intérêt d'un tel paradigme pour la conception d'applications embarquées, comme l'état de l'art le montre, mais aussi pour tenter de l'appliquer à différents niveaux d'abstraction comme nous le présentons ci-après (section 3.1.3).

Modèle. Conceptuellement, le modèle de composant que nous proposons se caractérise principalement par les points suivants :

- La volonté de faire cohabiter au sein du même modèle différentes possibilités d'interactions entre composants, aussi bien par appels de méthode que dirigées par les événements ou par flots de données. La prise en charge de cette « hétérogénéité » est une volonté de ne pas catégoriquement contraindre l'espace de conception à un modèle d'interaction spécifique.
- L'adoption d'un style architectural peu contraint et axé sur un assemblage « orienté fonctionnalité ».

[13] « Contrôle », au sens entendu dans les systèmes qualifiés de contrôle-commande.

- La promotion d'un langage de haut niveau, en cherchant notamment à ce que les concepts qui le définissent puissent se personnaliser par simples descriptions. Par exemple pour spécifier différents patrons d'interactions comportementaux pour une liaison entre deux composants ou pour « personnaliser » un composant à l'exécution (d'un point de vue implantation, il s'agit alors que ces aspects soient pris en charge automatiquement par l'infrastructure sans que le concepteur ait à s'en soucier).
- La volonté d'exploiter au maximum la description architecturale. En ce sens, considérant que celle-ci doit trouver sa place à l'« épicentre » du cycle de conception du logiciel, c'est sur les entités de première classe qu'un maximum de propriétés doivent être représentées et abstraites, notamment concernant les aspects dynamiques ou les contraintes temporelles par exemple. Cette démarche peut s'apparenter à la volonté d'utiliser les « concepts canoniques » des ADL (composant, interface, connecteur, etc) pour les décliner en entités spécifiques au domaine visé (pratique issue des « DSL », *Domain Specific Langage*).

L'objectif est alors de proposer un langage au sein duquel les préoccupations propres au domaine d'application visé sont (idéalement) exhaustivement capturées au niveau de la description de l'architecture. En d'autres termes, seule la logique fonctionnelle doit être implantée par les comportements des composants, tous les autres aspects propres au domaine visé doivent être pris en compte au niveau de la description architecturale.

Support à l'exécution. Le support à l'exécution du modèle de composant TINAP a fortement influencé notre proposition. Il nous semble notamment pertinent que les concepts proposés dans le modèle conceptuel trouvent leurs réalisations (« par bijection ») à l'exécution. Cette exigence n'est pas absolument nécessaire pour le déploiement final d'un système totalement statique mais trouve tout son sens pour des systèmes nécessitant un certain degré d'adaptation dynamique et s'avère obligatoire pour les étapes de prototypage ou de mise au point du logiciel. Dans ce dernier cas, l'objectif est alors d'assurer une fine traçabilité entre la description architecturale et son « modèle exécutif ». En d'autres termes, on peut dire que les concepts modélisés en amont doivent facilement trouver leur réalisation à l'exécution : il s'agit par exemple de pouvoir raisonner sur l'architecture, d'être en mesure de manipuler les concepts de l'ADL à l'exécution, à travers le « prisme d'une API ». Ces concepts doivent alors faire partie intégrante du modèle de programmation proposé.

FRACTAL/THINK. Les concepts proposés dans le cadre de TINAP peuvent être perçus comme une potentielle personnalité (un « profil ») du modèle de composants FRACTAL. En effet, ce dernier se veut particulièrement générique, donc facilement adaptable pour des besoins spécifiques à un domaine dédié[14]. La particularité du modèle FRACTAL demeure aussi dans la spécification d'un modèle d'exécutif comme nous venons de la caractériser ci-dessus, nous confortant à « l'adopter » comme socle conceptuel. La volonté de se baser sur THINK s'est imposée d'elle-même, notamment parce qu'il s'agit d'une implantation pour le langage C et que la plate-forme s'accompagne d'une chaîne d'outils fonctionnelle (permettant notamment, relativement à nos besoins, de prototyper nos expérimentations jusqu'aux étapes de déploiement et de reconfiguration) et d'une bibliothèque de composants implantant une bonne part de services élémentaires du système d'exploitation. Nous détaillons dans la suite de cette section d'autres facteurs qui caractérisent la « philosophie FRACTAL ».

3.1.2 Patron de « conception/programmation orientée membrane »

Séparer les préoccupations. Le modèle de composant FRACTAL est en partie basé sur le concept de membrane qui permet le découplage entre le code dit « fonctionnel » et le code dit de

[14] À titre d'exemple, PROACTIVE [Baduel *et al.*, 2006] est également une adaptation de FRACTAL pour le calcul parallèle à large échelle.

« contrôle »[15]. En ingénierie logicielle, cette volonté de découplage entre aspects fonctionnels est maintenant courante. On la retrouve par exemple dans les intergiciels qui servent d'intermédiaire de communication entre applications complexes, généralement réparties, ou dans les approches à conteneurs qui implantent un ensemble de « services techniques » requis par les composants [DeMichiel and Keith, 2006, OMG, 2002b, Microsoft, 2002]. Il en est de même pour les différentes implantations des spécifications FRACTAL et les développements de canevas dédiés à des domaines spécifiques basées sur ces spécifications [Seinturier *et al.*, 2006, Bruneton *et al.*, 2006, Quéma, 2005] : ils montrent toujours cette volonté de découpler la prise en charge des aspects non fonctionnels de la logique applicative.

Membrane. Cependant, l'originalité de la membrane FRACTAL réside dans le fait qu'elle est ouverte et distribuée[16]. En ce sens, les aspects non fonctionnels, plutôt qu'implantés statiquement, demeurent totalement programmables et configurables (donc extensibles et de complexité arbitraire) ; plutôt qu'implantés de manière monolithique et centralisée pour l'ensemble de l'application, ils sont distribués au sein des membranes et générés automatiquement autour des composants métiers (donc déployables de façon hétérogène en fonction des besoins non fonctionnels intra-application). Il ne s'agit pas d'aspects purement théoriques, différentes implantations de chaînes d'outils FRACTAL les mettent concrètement en œuvre. Il est ainsi possible d'implanter des politiques arbitraires de contrôle qui se superposent aux logiques métiers, et ce à un niveau de granularité quelconque. Par extension, considérant que le modèle structurel FRACTAL est hiérarchique, il est possible d'attacher un contrôle arbitraire à un groupe de composants (qui forment alors le contenu du composite soumis à une couche de contrôle commune). La maturité de cette pratique et des outils qui lui sont associés pourrait nous encourager à la qualifier de « paradigme de conception ». En premier lieu, les membranes FRACTAL implantent dans la pratique les aspects non fonctionnels du modèle de composant défini par les spécifications, c'est-à-dire les capacités d'introspection, de configuration et d'instanciation dynamiques.

Dans le contexte de notre approche, nous nous basons sur ce paradigme pour implanter les aspects non fonctionnels propres à notre modèle de composant. Il s'agit notamment d'implanter des membranes que l'on pourrait qualifier « de domaine », qui par exemple, mettent en œuvre les personnalités (de haut niveau) des composants et connecteurs TINAP. Dans ce cas, les membranes ne se contentent pas seulement de fournir des capacités de contrôle supplémentaires, mais implantent l'infrastructure d'exécution nécessaire aux composants.

3.1.3 Le paradigme composant à différents niveaux d'abstraction

La contribution de cette thèse s'inscrit dans une démarche globale dont l'objectif est de proposer une approche orientée architecture pour concevoir un système dans sa globalité, c'est-à-dire aussi bien pour le niveau applicatif, mais aussi pour celui de l'infrastructure d'exécution, de l'intergiciel (pour la gestion des communications réparties) ou encore pour les services élémentaires implantés par le système d'exploitation. Par infrastructure d'exécution, nous entendons l'implantation des concepts du modèle de composant (aussi bien les concepts structurels que ceux relatifs aux aspects non fonctionnels) nécessaires à l'exécution.
Cette intention est motivée par la volonté d'expérimenter le paradigme composant à large échelle et d'être en mesure de raisonner à différents niveaux d'abstraction en utilisant un ensemble de concepts communs.

Ce « noyau de concepts » partagé par tous les niveaux d'abstraction au sein de notre démarche peut se caractériser en premier lieu en deux points essentiels :

[15] « Contrôle » au sens FRACTAL du terme, c'est-à-dire qualifiant l'implantation des « non fonctionnalités » du canevas à composants.

[16] Bien qu'une membrane puisse théoriquement être répartie (sur plusieurs nœuds), « distribuée » signifie ici « éclatée ».

1. Un modèle structurel générique (canonique) d'assemblage de modules logiciels (les composants), couplé à un modèle d'abstraction de leurs comportements.
2. Le patron de « conception orientée membrane » présenté ci-dessus : il s'agit alors de proposer des extensions sémantiques « de domaine » à ce modèle et gérer leurs implantations de manière transparente au sein de l'infrastructure d'exécution.

Une conception orientée composants à différents niveaux d'abstraction peut être schématisée comme illustré par la figure 3.2.

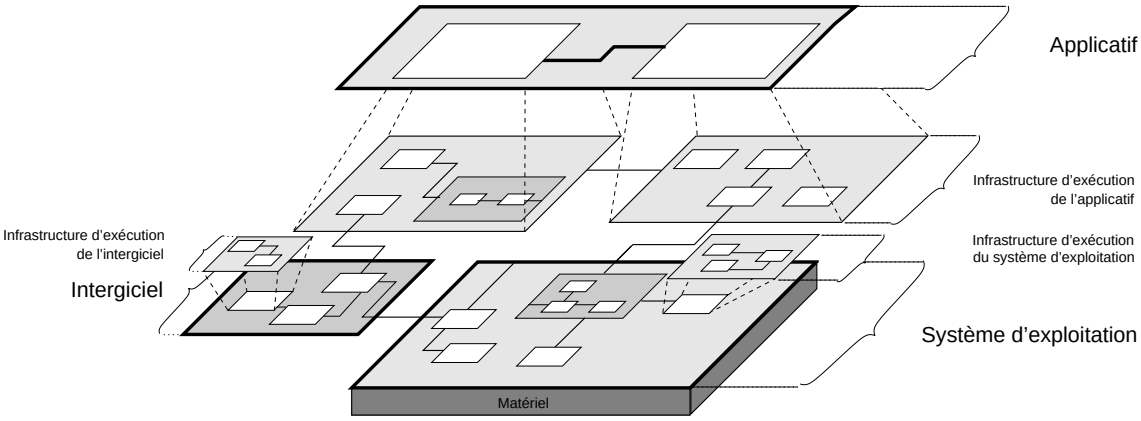


Figure 3.2 – Paradigme composant à différents niveaux d'abstraction d'un système.

Sur cette figure quatre niveaux d'abstraction sont représentés : l'applicatif, l'intergiciel, le système d'exploitation et l'infrastructure d'exécution.
Ces niveaux partagent donc le même ensemble de concepts structuraux, mais les besoins fonctionnels n'y sont pas identiques. Par exemple, il est intéressant de proposer au concepteur de l'applicatif de spécifier différents modèles d'exécution et de communication pour ses composants alors que ces aspects n'ont pas nécessairement de sens au niveau des fonctionnalités offertes par le système d'exploitation. D'une certaine manière, chacun de ces niveaux dispose d'un espace de conception qui lui est propre et les besoins d'ordre non fonctionnels leurs sont relatifs. Ces derniers sont implantés par la couche d'infrastructure d'exécution (comme représenté sur la figure).

Comme nous l'évoquions, nous nous sommes restreint à la conception d'applications centralisées sur mono-processeur.

L'objectif de cette démarche est donc d'être en mesure de raisonner avec un ensemble de concepts communs sur l'ensemble d'un système et de les spécialiser en fonction des besoins relatifs à un niveau d'abstraction donné. De plus, en cherchant à clairement spécifier les interactions entre ces niveaux d'abstraction, nous offrons la possibilité de « naviguer » au sein de ces niveaux dans une démarche de raffinement.

3.1.4 Cadre méthodologique

De manière transverse aux objectifs de notre proposition que nous avons détaillés, nous nous sommes intéressés à connecter l'espace de conception de TINAP au sein d'une démarche « dirigée par les modèles ». L'objectif est alors de favoriser au maximum le processus de développement pour le concepteur final, notamment en proposant une « édition de haut niveau » du

langage TINAP. Cet aspect est donc davantage tourné sur les besoins en terme d'outils qu'il est nécessaire de considérer lorsqu'il s'agit d'assister certaines étapes de la conception/modélisation du logiciel, de faciliter la réutilisation de l'existant, ou encore de contraindre autant que possible le concepteur à définir des spécifications cohérentes relativement à la sémantique donnée aux entités du langage.

3.2 Organisation de la contribution

L'espace de conception TINAP se divise en quatre vues, qui concrétisent différentes perspectives du logiciel. Pour chacune d'elle, un ensemble de concepts canoniques sont définis, caractérisant ainsi le « noyau » utilisé à différents niveaux d'abstraction. Ces aspects sont schématisés sur le tableau 3.1, sur laquelle sont aussi représentés certaines extensions de ce « noyau » définies pour certains niveaux d'abstraction.

<table>
<tr><th></th><th colspan="2">Chapitre 4</th><th colspan="2">Chapitre 5</th></tr>
<tr><th></th><th>Concepts canoniques</th><th>Niveau applicatif</th><th>Niveau infrastructure</th><th>Niveau OS</th></tr>
<tr><td>Vue structurelle</td><td colspan="4">Notions élémentaires de composant / composite - interfaces client/serveur – liaisons horz. / vert.</td></tr>
<tr><td></td><td></td><td colspan="2">Interfaces et liaisons de flot de données et d'événements</td><td></td></tr>
<tr><td></td><td></td><td>Interfaces de flot de données audio (chapitre 6)</td><td>Composants d'infrastructure : contrôleurs et intercepteurs (interfaces spécifiques)</td><td></td></tr>
<tr><td>Vue dynamique et contrôle</td><td colspan="4">Introspection et reconfiguration (Fractal)</td></tr>
<tr><td></td><td></td><td colspan="2">Modèle de concurrence</td><td></td></tr>
<tr><td></td><td></td><td>Contraintes temporelles</td><td></td><td></td></tr>
<tr><td>Vue implantation</td><td colspan="4">Abstraction de la structure interne des primitifs et accès aux variables partagées</td></tr>
<tr><td>Vue comportementale</td><td colspan="4">Abstraction du comportement interne des primitifs</td></tr>
</table>

TAB. 3.1 – Concepts TINAP canoniques et extensions nécessaires à différents niveaux d'abstraction.

Au sein du chapitre 4, nous nous attachons à présenter ces concepts canoniques couplés aux extensions que nous proposons pour caractériser le « niveau applicatif ». Le chapitre 5 présente ce que nous avons qualifié de « niveau infrastructure » et de « niveau OS ». Nous y présentons également comment le patron de « conception orientée membrane » est utilisé pour implanter les concepts de haut niveau proposés au niveau applicatif. Enfin, le chapitre 6 présente deux études de cas dont l'objectif est d'évaluer notre approche à ces différents niveaux d'abstraction.

Chapitre 4

Espace de conception TINAP

Sommaire

4.1 Introduction 36
4.1.1 Spécification des « Vues TINAP » 36
4.1.2 Méta-modélisation des concepts du langage 36
4.1.3 Plan du chapitre 38
4.2 Vue structurelle : le socle de l'approche 39
4.2.1 Composants, interfaces et liaisons 39
4.2.2 Langage de définition d'interface 45
4.2.3 Représentation graphique de la vue structurelle 47
4.2.4 Contraintes associées à la vue structurelle 48
4.2.5 Annotations supplémentaires attachées aux aspects structurels 49
4.2.6 Bilan sur la vue structurelle 51
4.3 Vue dynamique et de contrôle 52
4.3.1 Précisions sémantiques relatives aux aspects dynamiques 52
4.3.2 Description des aspects de contrôle 53
4.3.3 Représentation graphique de la vue dynamique 62
4.3.4 Sémantique et contraintes associées à la vue dynamique 62
4.3.5 Bilan sur la vue dynamique et de contrôle 64
4.4 Vues implantations 65
4.4.1 Vue implantation externe 65
4.4.2 Vue implantation interne 68
4.4.3 Bilan sur les vues implantation 69
4.5 Vues comportementales 72
4.5.1 Exemple d'architecture TINAP 73
4.5.2 Comportement de niveau implantation 73
4.5.3 Comportement de niveau composant primitif 74
4.5.4 Comportement de niveau architecture 77
4.5.5 Bilan sur les vues comportementales 84
4.6 TINAP à différents niveaux d'abstraction 85
4.7 Spécification de contraintes d'ordonnancement 86
4.7.1 Par priorité 86
4.7.2 Par échéances temporelles 86
4.7.3 Support des contraintes d'ordonnancement à l'exécution 87
4.8 Conclusion du chapitre 87

4.1 Introduction

Ce chapitre est dédié à la présentation de « l'espace de conception TINAP ». Il s'agit donc de détailler les concepts qui caractérisent le profil de composant que nous proposons.

4.1.1 Spécification des « Vues TINAP »

La spécification d'une architecture TINAP est associée à différentes vues [IEEE, 2000] qui permettent de caractériser le système selon une certaine perspective (ou aspect). Elles sont énumérées ci-dessous, et détaillées au sein de ce chapitre :

1. la **vue structurelle**, qui explicite la structure logicielle selon les concepts architecturaux définis (composants, interfaces typées syntaxiquement, interconnexions entre composants).
2. la **vue dynamique**, qui spécialise la vue structurelle en y intégrant des informations propres aux aspects dynamiques de l'application.
3. la **vue implantation**, qui caractérise la manière dont les composants métier sont implantés.
4. et la **vue comportement**, qui permet de raisonner sur les aspects comportementaux de l'application.

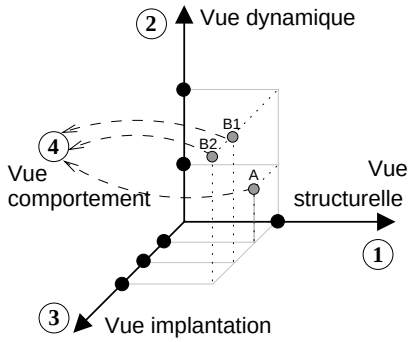


Figure 4.1 – Liens entre vues TINAP.

Certaines de ces vues sont obtenues de manière automatique en fonction des informations spécifiées par les autres vues. C'est par exemple le cas de la vue comportement (cf. figure 4.1), en fonction des informations spécifiées par le concepteur pour les vues structurelle, dynamique et implantation.
Comme illustré sur cette figure, ces vues représentent (idéalement) des aspects orthogonaux. À titre d'exemple, il est possible de caractériser différentes vues dynamiques pour une même vue structurelle (cas des applications `A` et `B`), ou encore pour les applications `B1` et `B2` qui correspondent à une structure et dynamique identique mais à deux implantations distinctes.
Comme nous le détaillerons dans ce document, à chacune de ces vues est attaché un ensemble de propriétés à valider pour s'assurer de la bonne cohérence des spécifications de l'application. Elles peuvent être également spécialisées en « sous-vues ».

4.1.2 Méta-modélisation des concepts du langage

Contexte. Comme nous l'avons déjà souligné, notre démarche s'inscrit dans un cadre méthodologique dont l'objectif est de fournir au concepteur un environnement de conception dirigé par les modèles.
Pour répondre à ce besoin, nous nous sommes attaché à « formaliser » les concepts du langage que nous proposons. Par « formalisation », nous entendons la définition précise de la syntaxe et de la sémantique de ces concepts en utilisant un sous-ensemble des capacités d'expression du MOF (*MetaObject Facility* [OMG, 2006a], basé sur les diagrammes de type entité-relation). Il s'agit d'un sous-ensemble car nous n'utilisons que les relations de base entre méta-classes, suffisantes pour caractériser notre langage : la généralisation (ou héritage), la composition et l'association dirigée.
Définir ces concepts sous forme de méta-modèles nous permet également de les implanter et

de les intégrer au sein d'outils de modélisation et de conception de « haut niveau » facilitant un processus assisté de développement pour le concepteur final.

Dans ce chapitre, il nous a semblé intéressant de présenter TINAP en nous appuyant sur des extraits de ces méta-modèles, facilitant notamment la compréhension de l'articulation du langage. Un méta-modèle permet en effet de décrire d'un point de vue syntaxique la structure des concepts de modélisation spécifique à un domaine particulier. Cependant, spécifier cette structure ne suffit pas toujours pour implanter des éditeurs totalement fonctionnels, c'est-à-dire qui contraignent de manière totale l'utilisateur à spécifier des modèles cohérents d'un point de vue des entités du langage défini. Dans ce chapitre, plutôt que de spécifier formellement ces contraintes (par l'utilisation d'un langage dédié de type OCL [OMG, 2006b] par exemple) nous nous contenterons de les expliciter en langage naturel lorsqu'elles nous paraissent nécessaires[17].
Il en est de même pour la sémantique des concepts que nous proposons à l'utilisateur final que nous nous attachons à décrire précisément dans le contexte de ce chapitre.

Structuration des concepts. Dans le chapitre 3, nous avons caractérisé un « noyau de concepts » partagé communément par différents niveaux d'abstraction au sein d'un système. La volonté d'isoler ce « noyau » se concrétise également à ce stade de spécification des méta-modèles. En effet, nous avons cherché à isoler les entités du langage partagées par les différents niveaux d'abstraction. La structuration des principaux paquetages est donnée figure 4.2 et met en évidence deux aspects :

- Le découplage entre les concepts du noyau au sein du paquetage `core` et d'extensions définies dans le paquetage `extensions`.
- Le découpage sous forme de vues : les paquetages `*_adl` et `*_idl` pour les méta-entités définies au sein de la vue structurelle, `*_control_decl` pour la vue dynamique, `implementation` pour la vue implantation et enfin le paquetage `behavior` pour la vue comportementale.

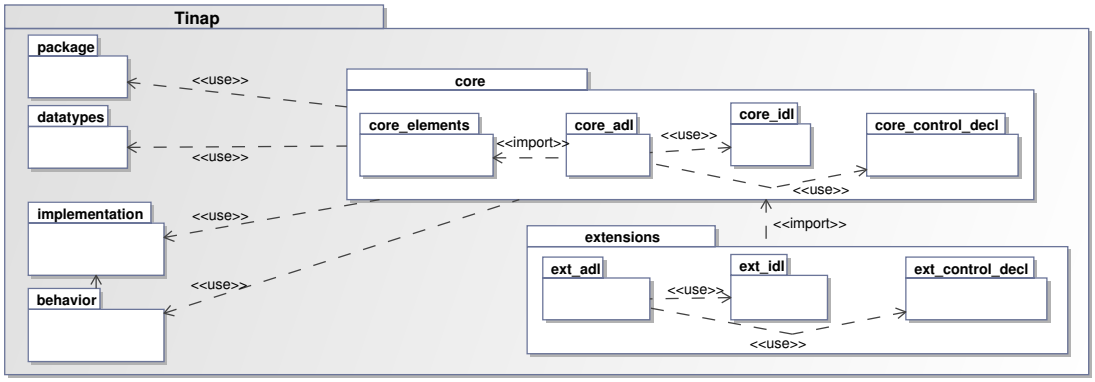


Figure 4.2 – Principaux paquetages des concepts TINAP.

Le paquetage `core` définit donc les entités principales du langage (composants primitif et composite, interfaces, liaisons, typage des interfaces) telles que nous les retrouverons à tous les niveaux d'abstraction. Les concepts de ce paquetage sont notamment ceux que l'on retrouve couramment dans les différentes implantations des spécifications FRACTAL[18].

Les concepts définis dans le paquetage `extensions` enrichissent ceux du noyau et correspondent à des besoins spécifiques au sein de l'espace de conception du niveau applicatif et de

[17] Leur implantation au sein d'un outil de modélisation sera succintement évoquée en annexe B.

[18] En ce sens, ce paquetage peut se présenter comme une méta-modélisation des concepts structurels proposés par FRACTAL.

celui de l'infrastructure d'exécution.
En effet, les besoins dans l'espace de conception de l'applicatif sont par exemple plus riches que ceux nécessaires pour définir les fonctionnalités élémentaires du système d'exploitation : par exemple, au sein de ce dernier, l'unique modèle de communication est l'appel de méthode. Dans ce cas, le chemin de communication réifié[19] par le concept de liaison, est directement implanté côté client par une référence vers une structure mémoire permettant l'accès aux implantations fournies côté serveur[20]. Cependant, au niveau applicatif, certains modèles de communication moins élémentaires sont nécessaires comme les liaisons de flots (de données et d'événement) que nous avons définies. Ces liaisons sont implantées par des talons (ou intercepteurs) spécifiques qui fournissent des fonctionnalités liées à des mécanismes de *bufferisation* et de synchronisation. Ces modèles de communication doivent être spécifiables en tant qu'entités de première classe pour le langage de niveau applicatif.

Le paquetage `datatypes` spécifie les types de données manipulables. Le paquetage `package` définit simplement un mécanisme pour structurer certaines entités du modèle (les définitions de composants et les signatures d'interfaces) en différents espaces de nommage.

Au sein de ce chapitre, nous présentons la plupart des concepts à partir d'extraits de méta-modèles « à plat », mais l'appartenance aux paquetages est parfois explicitement représentée sur les méta-classes lorsque la distinction s'avère nécessaire.

4.1.3 Plan du chapitre

Dans le contexte de ce chapitre, nous présentons dans un premier temps les concepts de TINAP que nous avons qualifiés d'« étendus » et les présentons en nous appuyant sur des extraits de méta-modèles. Ainsi, la prochaine section présente les concepts de la **vue structurelle**. La section 4.3 (page 52) ceux de la **vue dynamique**. Les sections 4.4 (page 65) et 4.5 (page 72) présentent respectivement les **vues implantations** et **comportementales** spécifiées par TINAP. Après une présentation de la manière dont TINAP s'appréhende à différents niveaux d'abstraction (section 4.6), la section 4.7 présente succintement la manière dont des contraintes d'ordonnancement peuvent être spécifiées par le concepteur au niveau applicatif.

[19] Dans ce document, nous utilisons couramment le terme « réifier ». Nous l'employons au sens large, « *transformer en chose ; donner le caractère d'une chose* » [Robert, 2002]. Il s'agit de « rendre concret », « manipulable » un concept.

[20] C'est notamment cette sémantique qui est attachée au sein des différentes implantations du modèle de composant FRACTAL.

4.2 Vue structurelle : le socle de l'approche

Cette section est dédiée à la présentation des aspects structurels (ou « vue structurelle ») que l'on peut considérer comme le « socle » du langage de description d'architecture de notre démarche.

4.2.1 Composants, interfaces et liaisons

4.2.1.1 Composants

Les concepts structurels relatifs aux composants sont schématisés figure 4.3 et sont spécifiés dans le paquetage `core` (cf. figure 4.2).
Certaines conventions graphiques sont utilisées pour faciliter la compréhension : les noms des classes abstraites sont représentés en italique sur fond blanc, les classes concrètes sur fond grisé.

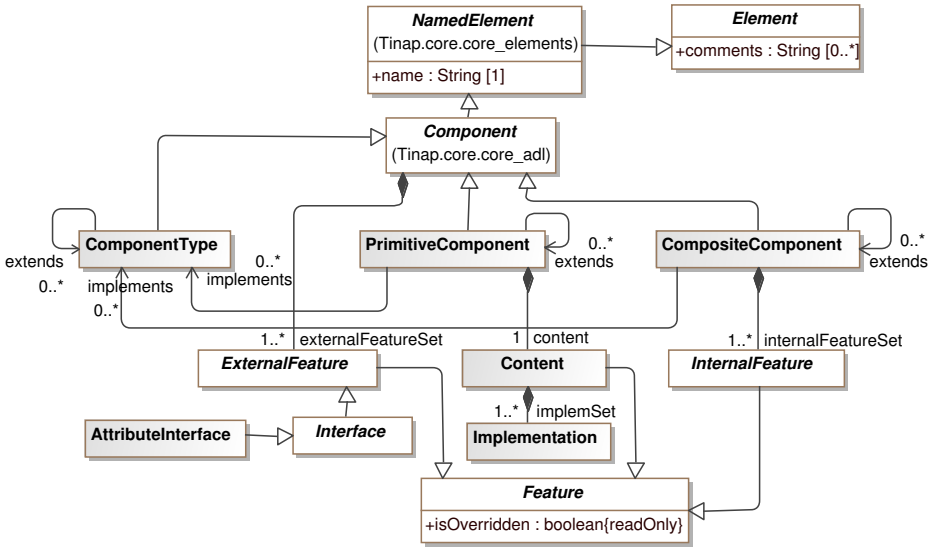


Figure 4.3 – Extrait du méta-modèle relatif aux définitions de composants.

Un composant est caractérisé de manière unique par un nom. Trois « natures de composants » sont définies : les `types`, les `primitifs` et les `composites`. Ces entités ont un point commun, celui d'exporter des `propriétés externes` (*ExternalFeature*), par exemple, un ensemble d'interfaces et/ou d'attributs (méta-classe *AttributeInterface*[21]). Un type de composant est un composant pour lequel seules ces propriétés sont spécifiées. L'héritage (cf. paragraphe suivant, éventuellement multiple, spécifié par les références *extends*) est défini pour chaque nature de composant.

Un primitif et un composite peuvent être considérés comme des implantations (notamment par l'intermédiaire de la référence *implements*) d'un type de composant. Une implantation d'un primitif sera définie par le code qui lui sera associé (matérialisée par la méta-classe *Content*). Cette implantation est alors attachée à de multiples fichiers textes (ou binaires méta-modélisés par la classe *Implementation*). La multiplicité attachée à ce lien de composition entre contenu et implantation ne permet pas d'associer de multiples implantations d'une même spécification de primitif, mais simplement d'associer plusieurs fichiers dont l'agrégation implante cette spécification.

[21] Comme nous le détaillerons plus loin (cf. figure 4.7, page 43), les attributs se spécifient eux aussi au sein d'une interface.

Sémantique : Composant primitif

1. Dans notre démarche, nous utilisons la notion de composant au sens de [Szypersky *et al.*, 2002], c'est-à-dire une entité logicielle qui encapsule du code[22] (source ou compilé) et qui externalise certaines propriétés à son environnement. Il s'agit d'une unité de composition.
2. Les « fonctionnalités » fournies par un composant primitif sont arbitraires. Une implantation contient du texte (algorithmes) et/ou des données. Un composant qui encapsule des données est « **avec état** » contrairement à un composant « **sans état** ».
3. À l'exécution, la structure architecturale des composants définie par le concepteur à ce niveau d'abstraction est réifiée.

Liens d'héritage et d'implantation. Ces liens sont explicités sur la figure 4.3. Le lien d'héritage est spécifié par les références *extends* : l'héritage ne peut donc se faire qu'entre composants de même « nature ». L'héritage peut être multiple, dans ce cas, un composant fils hérite de toutes les *features* de l'ensemble de ses pères. La surchage est également proposée, un composant fils peut alors redéfinir une *feature* déclarée initialement par le père (renseignée par l'attribut en lecture seule *isOverridden* de la figure 4.3). La surcharge s'obtient lorsque le nom de la *feature* fils est identique à celui du père (par exemple le nom d'un sous-composant).
Le lien d'implantation est spécifié par la référence *implements* entre un primitif ou un composite et un type de composant.

Sémantique : Liens d'héritage et d'implantation

1. Les notions d'héritage et d'implantation de type de composant ne sont pas réifiées à l'exécution. Il s'agit simplement d'opérateurs logiques qui favorisent la réutilisation, la factorisation et la compréhension des définitions architecturales spécifiées par l'utilisateur à l'étape de la conception.

Composites. Un composite, quant à lui, fournit en guise d'implantation un ensemble de `propriétés internes` détaillées figure 4.4.
La composition interne d'un composite se spécifie alors par des `sous-composants` (qui consti-

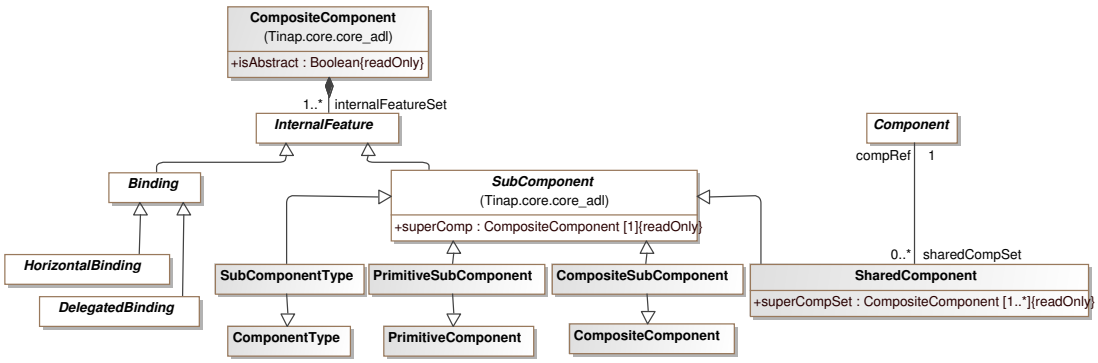


Figure 4.4 – Structure interne d'un composite (paquetage `core`).

tuent son contenu) reliés par un ensemble de `liaisons`. Une telle configuration est définie par des `liaisons horizontales` (pour lier les composants d'un même niveau d'encapsulation) et des `liaisons de délégation` (pour assurer la composition hiérarchique entre les interfaces d'un composite et les interfaces de ses sous-composants). Une liaison entre composants est toujours dirigée et se définit entre deux interfaces.

[22] Dans notre approche, il s'agit du C.

Un composite est une entité de composition hiérarchique dont la structure interne est une arborescence de sous-composants dont la profondeur est arbitraire et dont les feuilles sont des primitifs. Nous parlerons de « composite abstrait » (attribut *isAbstract* de la classe *CompositeComponent*) si au moins un sous-composant de son contenu est un `type` de composant ne fournissant donc pas d'implantation. Bien entendu dans ce cas, le composite ne pourra pas être instancié. Ce mécanisme permet de factoriser les définitions de composites : un composite `B` peut par exemple hériter d'un composite abstrait `A` puis surcharger (cf. paragraphe « liens d'héritage », page 40) le `type` de sous-composant et devenir ainsi instanciable. Cet attribut est en lecture seule (*readOnly*), c'est-à-dire qu'il est renseigné automatiquement par l'outil de modélisation et ce de manière transparente pour l'utilisateur.
Le composant partagé (*SharedComponent*) est une notion issue des spécifications FRACTAL. Il s'agit d'un composant singleton dont l'instance (et son implantation) est partagée par plusieurs composites (cela est matérialisé par la multiplicité attachée à l'attribut *superCompSet* de la classe *SharedComponent*).
À l'exécution, un tel composant partage son implantation (texte) et son état (données internes) à l'ensemble des composites de l'application qui le contiennent.

En premier lieu, la notion de composite offre un degré supplémentaire d'encapsulation pour un ensemble de fonctionnalités et matérialise une configuration de composants (ses sous-composants liés) en tant qu'entité de première classe. Le composite peut être également perçu comme une unité de composition (au même titre que la relation de composition issue des langages orientés objets). À ce titre, cet « opérateur d'assemblage » peut être associé à des sémantiques plus avancées, notamment pour caractériser la notion de domaine (de désignation, de ressources[23], de défaillance, de sécurité, etc), ou plus généralement pour fournir un ensemble de services d'ordre non fonctionnel à un groupe de sous-composants de son contenu.

Sémantique : Composite

1. Un composite est une entité de première classe permettant d'assurer, à un degré arbitraire, la composition hiérarchique.
2. Il peut matérialiser un espace de fonctionnalités soumises à un contrôle (d'ordre non fonctionnel) commun.
3. Il s'agit d'une entité réifiée, et donc manipulable, à l'exécution[24].

4.2.1.2 Interfaces

Une définition précise du concept d'interface est donnée figure 4.5.

Une interface est caractérisée par un nom associé à l'espace de nommage du composant auquel elle est attachée. Différentes « **natures d'interfaces** » sont définies dans le profil de composants : les ***interfaces de service***, les ***interfaces de flot*** et les ***interfaces d'attribut***.
Les interfaces de service correspondent aux services requis et fournis par un composant.
Les interfaces de flot concrétisent un modèle de communication pour lequel l'information sera échangée par flots entre un producteur en sortie et un consommateur en entrée.
Ces interfaces de flot sont spécialisées en deux natures d'interfaces, celles pour assurer le transport des données (*DataFlowInterface*) et celles pour assurer une communication événementielle entre composants (*EventFlowInterface*). La sémantique d'un événement est proche de celle associée aux primitives de signalisation fournies dans les systèmes d'exploitation.

[23] En terme de contrôle sur l'utilisation de ressources, on peut citer le projet FIRST [Lipari and Bini, 2005] ou le modèle PECOS, qui d'une certaine manière, utilise le composite en tant qu'entité d'encapsulation temporelle (sous forme de budget sur l'utilisation du processeur).
[24] Il ne s'agit pas d'un « simple » opérateur de composition logique utilisé à l'étape de la modélisation (comme c'est le cas des approches présentées dans l'état de l'art).

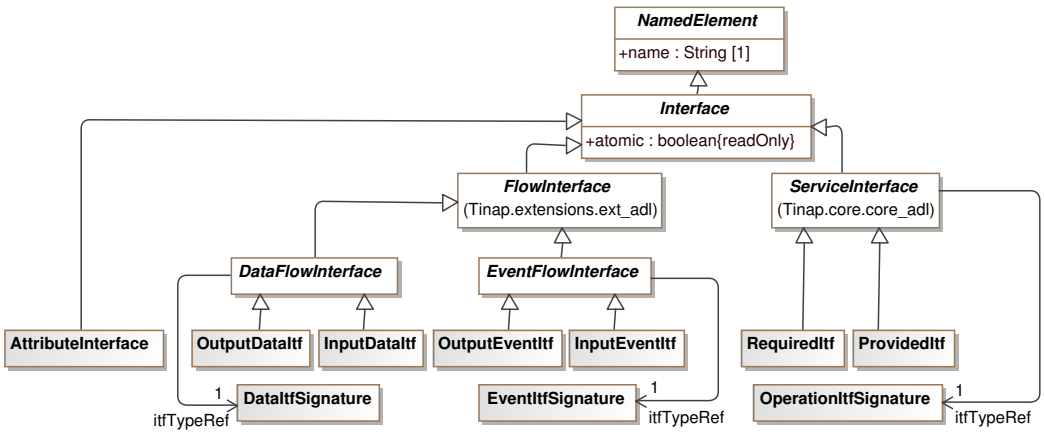


Figure 4.5 – Définition des interfaces.

Toutes les interfaces sont typées par une signature (meta-modélisée par la référence *itfTypeRef* de la figure 4.5) : un ensemble de données typées, d'événements nommés ou de signatures de méthodes (d'opérations) selon leur nature. L'attribut *atomic* attaché aux interfaces donne une information supplémentaire sur les signatures : il sera placé à `vrai` dans le cas où la signature associée à l'interface ne définit qu'un seul élément (il s'agit d'un attribut en lecture seule, renseigné automatiquement par l'outil de modélisation). Le langage de définition des interfaces est détaillé dans la section 4.2.2.
Les interfaces de service sont définies dans le paquetage `core/core_adl` (cf. figure 4.2 page 37, il s'agit en effet du concept d'interface utilisé à tous niveaux d'abstraction d'un système, correspondant à un appel de méthode élémentaire), les interfaces de flots sont définies dans le paquetage `extension/ext_adl` (et les définitions de leurs signatures dans `extension/ext_idl`).

Au sein de notre démarche, la nécessité d'introduire ces différentes natures d'interface se justifie pour plusieurs raisons :

D'un point de vue sémantique, les interfaces de service et les interfaces de flot ne mettent pas en œuvre les mêmes patrons de communication. Dans le premier cas, il s'agit de proposer un modèle de type client/serveur entre deux composants. Ce modèle concrétise un lien de dépendance fonctionnelle entre les composants et permet d'échanger des informations dans les deux directions de la liaisons (client vers serveur et la réciproque). Dans le deuxième cas, il s'agit de proposer un modèle de type producteur/consommateur où l'information est distribuée de manière asynchrone (et unilatéralement) quel que soit le nombre de protagonistes impliqués dans les échanges (communication de type *broadcast* par exemple). Dans ce type d'interaction, un producteur ne connaît pas l'identité du ou des potentiels consommateurs.
Différencier les trois natures d'interfaces (service, donnée, événement) se justifie aussi par le fait qu'il sera possible d'attacher différents comportements aux liaisons entre interfaces (cf. section 4.3.2.3, page 58) qui ne sont pas communs.

Une seconde hiérarchisation des concepts d'interfaces est donnée figure 4.6. Elle permet de faire la distinction entre interfaces ***d'entrée*** et ***de sortie***. Dans le contexte de ce document, nous utiliserons ces deux nomenclatures (figures 4.5 et 4.6), pour qualifier les interfaces ***de flot***, ***fournie***, ***d'entrée d'événement***, ***de sortie***, ***de service*** (la sémantique de l'attribut *internalInteraction* sera précisée section 4.2.3, page 47).

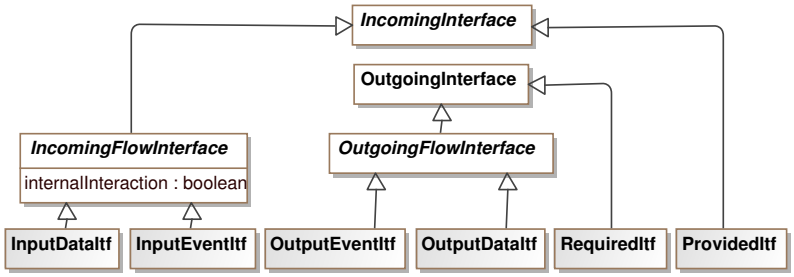


Figure 4.6 – Caractérisation des « interfaces d'entrée » et « de sortie ».

L'interface d'attribut est une interface particulière représentée par la figure ci-contre (figure 4.7).

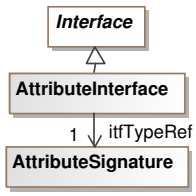


Figure 4.7 – Interface pour définir des attributs.

Elle permet d'externaliser un ensemble de propriétés pour configurer/caractériser le composant fonctionnel auquel elle est attachée. Avant l'étape de compilation de l'application, le concepteur doit paramétrer les attributs de ces interfaces en leur assignant une valeur initiale (cf. figure 4.13, page 47).
Une telle interface peut être définie pour un composant primitif aussi bien que pour un composite.

Sémantique : Interfaces

1. À l'étape de conception, l'interface est une entité de regroupement d'informations liées logiquement.
2. À l'exécution, la notion d'interface est réifiée (rendue concrète et donc « manipulable » au sein du langage de programmation sous-jacent).

4.2.1.3 Liaisons

Types. Deux types de liaisons sont définies, les `liaisons horizontales` et les `liaisons de délégation`.
Une liaison horizontale permet de lier deux composants d'un même niveau d'encapsulation et ne peut s'établir qu'entre couple d'interfaces de même nature et de « polarité opposée », c'est-à-dire : entre une interface de sortie de données (*OutputDataItf*) et une interface d'entrée de données (*InputDataItf*), entre une interface de sortie d'événement (*OutputEventItf*) et une interface d'entrée d'événement (*InputEventItf*) et enfin entre une interface requise (*RequiredInterface*) et une interface fournie (*ProvidedInterface*).
Typiquement, une liaison de délégation (ou liaison verticale) permet de lier les interfaces d'un composite avec les interfaces de ses sous-composants permettant l'assemblage hiérarchique. Elle s'établit aussi entre deux interfaces de même nature mais de « même polarité », par exemple entre une interface fournie par un composite et une interface fournie par l'un de ses sous-composants.

À titre d'illustration, la spécification d'une liaison horizontale et d'une liaison de délégation (entre deux interfaces requises) est donnée figure 4.8 (les structures des méta-classes pour exprimer les liaisons entre interfaces de donnée et d'événement sont similaires).

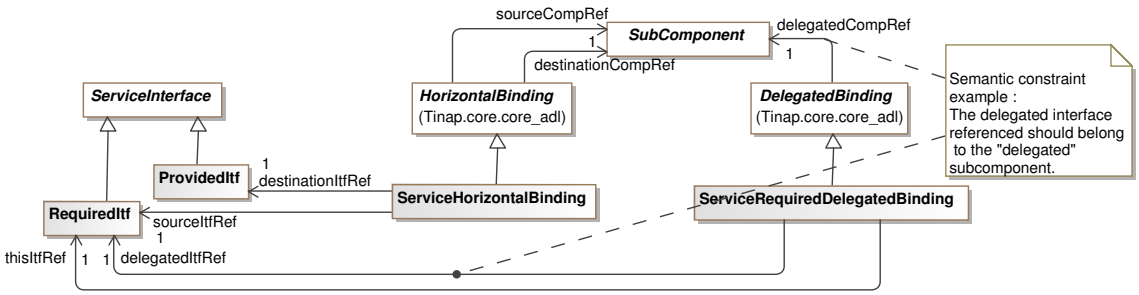


Figure 4.8 – Extrait de méta-modèle relatif aux liaisons de service.

La figure 4.9 schématise une liaison particulière que nous avons spécifiée pour les interfaces d'attributs. Seules les liaisons de délégation sont définies pour cette nature d'interface. Il s'agit donc d'un simple mécanisme qui permet de « faire remonter » les interfaces d'attribut d'un sous-composant au niveau de son composite parent. Cette spécificité s'avère intéressante pour externaliser au niveau du composite des attributs internes tout en préservant l'encapsulation.

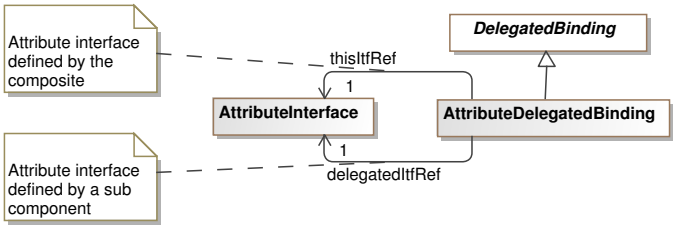


Figure 4.9 – Liaison de délégation pour les interfaces d'attributs.

Interface multi-liée. Une même interface de composant peut être attachée à différentes liaisons, pour partager des services entre composants (pour les interfaces de service) ou pour attacher un consommateur à plusieurs producteurs (et réciproquement, pour les interfaces de flot). Dans ce cas, nous parlerons d'une **interface multi-liée***[25] impliquée dans une **liaison multiple***. Une liaison multiple permet à l'utilisateur de définir des schémas d'interconnexion de type `1-N`, `N-1` (et également de type `N-M`) entre interfaces de même nature. D'un point de vue dynamique, les sémantiques implicites attachées aux interfaces multi-liées sont détaillées page 52.

Au sein du méta-modèle, nous utilisons également une seconde hiérarchisation pour faire la distinction entre les liaisons en fonction de la nature des interfaces qu'elles lient (représentée figure 4.10). Nous parlerons donc de « nature de liaison » pour qualifier les *`liaisons de service`*, les *`liaisons de donnée`* et les *`liaisons d'événement`*.

Sémantique : Liaisons

1. Les liaisons sont des canaux logiques de communication. Elles concrétisent les flux d'interaction qui s'opèrent entre entités logicielles.
2. À l'exécution, les communications ne peuvent se faire que par leur intermédiaire.

[25] Dans l'ensemble du document, les expressions suivies d'un astérisque correspondent aux entrées du glossaire (page 152).

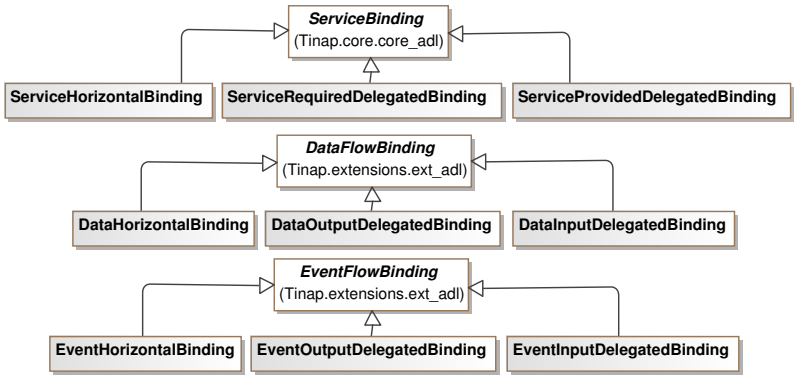


Figure 4.10 – Caractérisation des liaisons en fonction de la nature des interfaces liées.

4.2.2 Langage de définition d'interface

IDL. Comme nous l'avons déjà mentionné, les interfaces sont typées par une signature. Selon la nature de l'interface (c'est-à-dire `de donnée`, `d'événement`, `de service` ou `d'attribut`), les langages de définition d'interface (*IDL*) sont partiellement explicités sur la figure 4.11.

En premier lieu, une structure de méta-classes est définie pour spécifier les types de données (dans le paquetage `datatypes`). La figure 4.11 n'en donne qu'un aperçu sommaire, mais l'essentiel est de retenir qu'un ensemble de types primitifs y sont définis (« entier », « flottant », etc), et qu'il est offert à l'utilisateur la possibilité de définir ses propres types complexes, scalaires ou structurés (de type structure C).

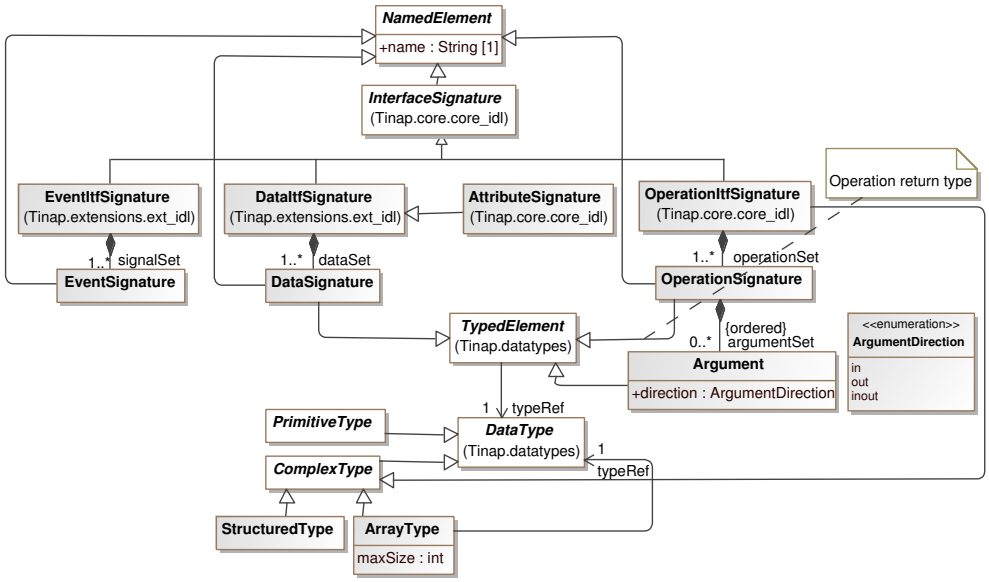


Figure 4.11 – Langage de définition d'interface et types de données manipulables (extrait).

Les signatures des interfaces se définissent donc de la manière suivante :

- La signature associée à une **interface de service** est classiquement définie par un ensemble d'opérations (elles-mêmes définies par un ensemble de paramètres d'entrée/sortie et une valeur de retour). Au sein de notre démarche, nous n'avons pas considéré la prise en charge d'exception.
- La signature associée à une **interface de données** permet de spécifier quels seront les types des données échangées (types primitifs ou complexes).
- La signature associée à une **interface d'événement** est simplement identifiée par un ensemble de noms.
- La signature d'une **interface d'attributs** se définie de manière identique à une signature d'interface de données.

On peut également noter que nous considérons la signature d'une opération comme un type de données complexes. En effet, à l'implantation, nous considérons qu'il est légal de passer des références d'interfaces en paramètre d'appel d'opération.

Points d'interactions atomiques. Dans notre démarche, le concept d'interface permet de définir un ensemble de signatures de données, de signaux ou de méthodes regroupées par un lien logique (ou conceptuel). Cependant, les composants communiquent concrètement non par les interfaces mais par les « ***points d'interaction atomique*** » (*AtomicInteractionPoint*, figure 4.12). À titre d'exemple, la signature textuelle[26] de l'interface de service ci-dessous,

```
OperationInterface calculItf {
   int mutl (...);
   int sous (...);
}
```

définit deux points d'interaction atomiques. Comme nous l'avons explicité page 42 (figure 4.5) une « interface atomique » est une interface qui définit un unique point d'interaction.

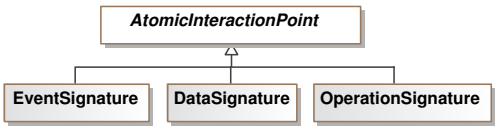


Figure 4.12 – Points d'interactions atomiques.

Initialisation des attributs. À l'étape de conception de l'application, les valeurs initiales des attributs doivent être renseignées. Un extrait du méta-modèle qui permet ce paramétrage est donné figure 4.13. L'outil de modélisation s'assure alors de la bonne cohérence entre les types de *ValueSpecification* et ceux des attributs définis au sein de l'interface correspondante.

[26] Par « **signature textuelle** », nous faisons référence à la projection d'une signature spécifiée dans l'outil de modélisation dont la structure est conforme à celle donnée figure 4.11 vers une représentation sérialisée dans un fichier texte.

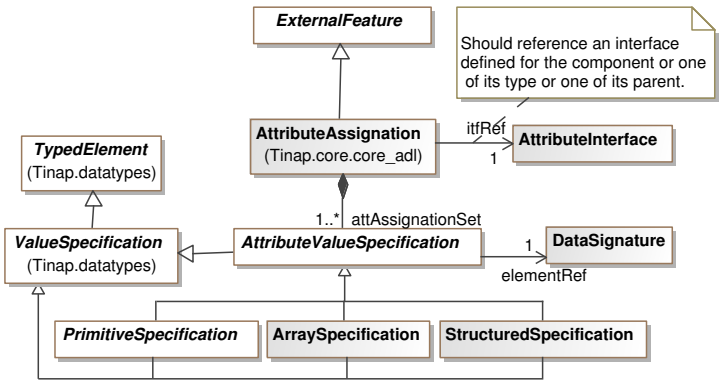


Figure 4.13 – Paramétrages initiaux pour les interfaces d'attributs (extrait).

4.2.3 Représentation graphique de la vue structurelle

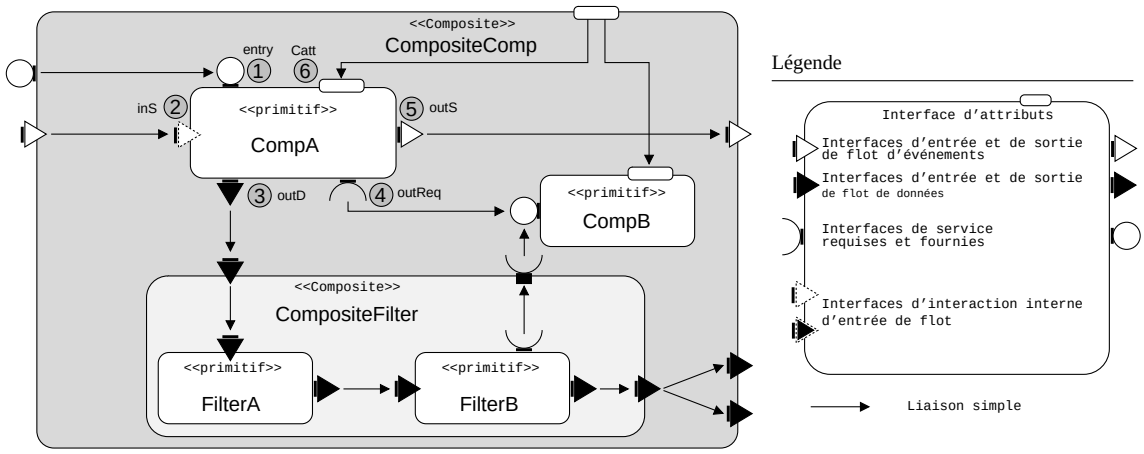
Dans le contexte de ce document, nous serons amenés à représenter des assemblages de composants sous forme graphique. Un exemple de description architecturale est donné par la figure 4.14 (avec les signatures textuelles des interfaces du composant `compA`) : il s'agit d'une représentation de la vue structurelle (définie section 4.1.1, page 36) de niveau applicatif.

Les interfaces de flot sont représentées par des triangles (blancs pour les interfaces d'événement, noirs pour les données) dont l'orientation spécifie la direction du flux. Pour les interfaces de service, nous utiliserons la « notation CCM » (facettes et réceptacles). Les interfaces d'attribut sont représentées par un rectangle blanc. Les liaisons sont représentées par un simple trait dirigé.

Sur cette architecture sont définies des interfaces multi-liées : l'interface fournie de `CompB` dont les services sont partagés par le composant `CompA` et `CompositeFilter`, l'interface de sortie de donnée du composite `CompositeFilter` et l'interface d'attributs du composant de plus haut niveau `CompositeComp` lié par une liaison complexe avec les interfaces d'attributs de `CompA` et `CompB`.

Interfaces d'interaction interne. L'attribut *internalInteraction* défini pour les interfaces d'entrée de flot (cf. figure 4.6, page 43) externalise au niveau de l'architecture une propriété du comportement interne de l'implantation d'un primitif. En effet, pour un tel composant, nous faisons la différence entre une interface dont les points d'interaction atomique sont des points d'entrée pour un fil d'exécution (typiquement l'implantation d'une méthode) et les interfaces dont les points d'interaction sont de simples points de communication (internes à cette implantation de méthode) avec l'environnement. Par exemple, une donnée ou un événement peut être consommé au cours de l'exécution d'une méthode via de telles interfaces. Pour insister sur cette distinction, nous parlerons respectivement d'***`interface implantée`**** et d'***`interface d'interaction interne`**** :

- Une interface fournie (d'un primitif) est systématiquement une *`interface implantée`*. En effet, elle externalise les services fournis par le composant.
- Une interface d'entrée de flot peut être **soit** une *`interface implantée`*, **soit** une *`interface d'interaction interne`*. C'est donc pour faire la distinction entre ces deux cas que l'attribut *internalInteraction* doit être renseigné pour les interfaces d'entrée de flot.



Signatures textuelles des interfaces de CompA :

```
1. Interface fournie entry
     OperationInterface provItf {
       void mA (...);
       void mB (...); }
2. Interface d'entrée d'événement inS
     EventInterface inSignItf
       {inNotifyEvt;}
3. Interface de sortie de donnée outD
     DataInterface outDataItf
       {MyStructType s;}
4. Interface requise outReq
     OperationInterface reqItf {
       int mCa (...);
       String mCb (...); }
5. Interface de sortie d'événement outS
     EventInterface outSignItf {
       EventA;
       EventB; }
6. Interface d'attribut Catt
     AttributeInterface compAAtts
       {boolean b;}
```

Figure 4.14 – Conventions graphiques utilisées pour représenter la vue structurelle de niveau applicatif.

Graphiquement, nous faisons la distinction entre *`interface implantée`* et *`interface d'interaction interne`* pour les interfaces d'entrée de flot (dont les contours sont schématisés respectivement en trait continu ou en pointillés sur la figure 4.14).

Nous serons amenés à revenir sur ce point dans dans la section 4.4 (page 65) mais ces explications sont tout de même nécessaires à la bonne compréhension de certains aspects de la vue dynamique présentée en section 4.3.

4.2.4 Contraintes associées à la vue structurelle

4.2.4.1 Contrats syntaxiques sur les liaisons

Au sein du langage TINAP, nous avons spécifié (cf. section 4.2.1.3, page 43) comment les liaisons s'établissent entre interfaces de même nature. Il s'agit donc d'une contrainte exprimée directement de par la structure du méta-modèle, le concepteur ne peut donc pas définir de liaisons entre interfaces de natures différentes.

Cependant, à ce niveau de description (la vue structurelle), nous n'autorisons pas au concepteur de définir une liaison entre interfaces dont les signatures associées ne sont pas totalement compatibles (c'est-à-dire de même type).

Cette contrainte est relâchée dans le cas des liaisons de délégation pour les interfaces d'attribut (cf. figure 4.9, page 44). Dans ce cas, l'ensemble des attributs défini au sein de l'interface du composite doit inclure l'ensemble des attributs définis par les interfaces d'attributs déléguées des sous-composants. Nous ne détaillons pas dans le contexte de ce document la définition précise de

cet opérateur d'inclusion, nous donnons le simple exemple ci-dessous pour illustrer ce propos, qui définit une interface d'attributs d'un composite liée à deux interfaces d'attributs de ses sous-composants A et B dont les signatures sont les suivantes :

Pour un composite : `AttributeInterface composite_Atts { float a; int b; }`
Pour son sous-composant A : `AttributeInterface subCompA_Atts { float a; }`
Pour son sous-composant B : `AttributeInterface subCompB_Atts { int b; }`

4.2.4.2 Contraintes sémantiques de domaine

Comme nous l'évoquions en introduction de ce chapitre, certaines contraintes imposées à l'étape de modélisation pour concevoir une architecture cohérente ne peuvent toutes s'exprimer à partir de la simple structure du méta-modèle. Cette sous-section présente ces contraintes.

Espaces de nommage et unicité des noms. Différents espaces de nommage s'appliquent à certaines entités du langage, par exemple pour la signature d'une interface sur l'ensemble des noms des points d'interactions qui y sont définis, pour un composant primitif sur l'ensemble des *ExternalFeatures* nommées qui lui sont associées, pour un composite sur l'ensemble des sous-composants nommés attachés à son contenu, etc.
Pour chaque espace de nommage, l'unicité des noms doit être garantie par l'outil de modélisation pour éviter les conflits.

Relativement aux articulations des concepts. À ce niveau de description de l'application, la majeure partie des contraintes sémantiques sur l'articulation entre les entités du langage sont explicitées à même la structure du méta-modèle. Par exemple, les liens d'héritage qui ne peuvent s'établir qu'entre composants de même type (cf. figure 4.3, page 39), les définitions d'entités de première classe telles le *ClientServerHorizontalBinding* ou le *DataInputDelegatedBinding* (cf. figure 4.6, page 43) qui permettent de contraindre les liaisons en fonction de leur types et des natures d'interfaces impliquées.
Cependant, d'autres contraintes sémantiques ne peuvent s'expliciter de la sorte sans complexifier terriblement la structure du méta-modèle. C'est le cas des propriétés liées à l'encapsulation du modèle structurel. Par exemple, une liaison horizontale ne s'établit qu'entre sous-composants de même niveau d'encapsulation ou une liaison verticale qu'entre une interface d'un composite et une interface d'un « sous-composant direct ».
La mise en œuvre de ces contraintes (respect des contrats syntaxiques et contraintes sémantiques) au sein d'un prototype d'éditeur implanté dans le contexte de cette thèse sera succintement traitée dans l'annexe B.

4.2.5 Annotations supplémentaires attachées aux aspects structurels

4.2.5.1 Mécanisme d'annotation

D'un point de vue méthodologique et de manière générale, nous considérons que les annotations regroupent des informations qui permettent de caractériser ou de paramétrer précisément (sous forme de simples attributs au sein du méta-modèle, ou sous forme plus complexe) certains aspects spécifiques de domaine et s'ajoutent aux entités du langage que nous avons définies.
Cependant, il faut noter qu'elles n'ont pas toutes le même impact sur le cycle de développement de l'application. En effet, nous pouvons faire la distinction entre les annotations utilisées :

– à l'étape de conception uniquement. C'est par exemple le cas pour celles nécessaires à un processus de vérification ou de validation.

– pour paramétrer la génération de l'infrastructure d'exécution.
– à l'étape de l'exécution. Dans ce cas, elles sont réifiées au sein de l'implantation.

Au sein du modèle, nous pouvons également faire la distinction entre les annotations renseignées par le concepteur et celles renseignées par un processus automatique en ligne couplé à l'outil de modélisation.

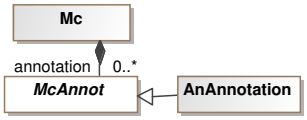


Figure 4.15 – Annotations.

Dans l'ensemble de ce document, nous utilisons le patron présenté figure 4.15 pour spécifier les annotations qui s'ajoutent aux méta-classes définies dans notre langage. À chaque méta-classe (nommée `Mc`) est associée, par une relation de composition, une classe abstraite dont le nom est celui de la méta-classe suffixé par « *Annot* ». Les annotations proposées pour cette méta-classe héritent alors de cette dernière. Ce simple mécanisme nous a semblé pertinent, pour (1) méta-modéliser un découplage entre informations qualifiées d'annotations et méta-classes du langage, pour (2) faciliter une éventuelle hiérarchisation des annotations et pour (3) faciliter les extensions du méta-modèle.

4.2.5.2 Annotations sur les interfaces

Relativement aux aspects structuraux que nous avons présentés jusqu'à présent, seules les annotations représentées sur la figure 4.16 sont définies.

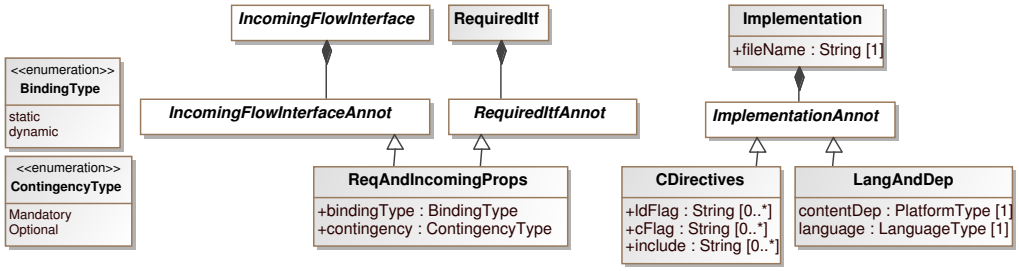


Figure 4.16 – Annotations supplémentaires sur les interfaces et le contenu.

Pour les interfaces d'entrée de flots et les interfaces requises. Les attributs pour ces interfaces sont issus des spécifications FRACTAL :

– *bindingType* : pour spécifier que la référence vers cette interface est connue seulement à l'étape de l'exécution (*Dynamic*) et non à l'étape de la compilation.
– *contingency* : pour spécifier que l'interface doit obligatoirement être liée avant l'étape de l'éxécution de l'application ou non. Dans ce dernier cas, l'attribut permet par exemple de spécifier qu'un service requis par un composant peut ne pas être présent à l'exécution.

Pour les implantations de composants.

– *contentDep* : cet attribut permet de spécifier si l'implantation attachée au composant est dépendante d'une cible matérielle donnée (PowerPC, ARM, ...) ou s'il s'agit d'une implantation générique.
– *language* : pour spécifier le langage de l'implantation du contenu (langage des composants primitifs THINK , assembleur, ...).
– *ldFlag*, *cFlag*, *include* : des directives supplémentaires pour la compilation C.

4.2.6 Bilan sur la vue structurelle

Dans cette section ont été présentés les concepts architecturaux caractérisant le socle de l'espace de conception de TINAP.
Hormis les interfaces de flot (et les liaisons associées), les notions présentées dans cette section sont définies au sein du « noyau de concept » (cf. page 37), il s'agit des entités architecturales canoniques sur lesquelles nous nous basons à tous niveaux d'abstraction d'un système.
En premier lieu, il s'agit donc de spécifier les assemblages de composants fonctionnels qui constituent la sémantique globale du logiciel à concevoir. C'est en ce sens que nous qualifions notre approche d'« orientée fonctionnalité ». Cette description architecturale est alors utilisée de manière centrale au processus de conception, notamment pour spécifier certains aspects dynamiques comme nous le détaillons dans la prochaine section.

4.3 Vue dynamique et de contrôle

L'objectif de notre approche est d'utiliser la représentation de l'architecture logicielle telle que nous l'avons présentée dans la section précédente comme modèle de base pour le processus de conception du logiciel. Cependant, la simple spécification de l'architecture statique ne suffit pas pour caractériser de manière précise les propriétés du domaine d'application visé.

La section qui suit présente les mécanismes proposés au concepteur pour spécifier les aspects dynamiques et les aspects de contrôle de son application. Cette étape consiste à annoter – de manière déclarative – la structure logicielle pour y ajouter des informations caractérisant le modèle d'exécution (ou modèle de concurrence) et les aspects de reconfiguration dynamique des différents composants de l'application.

4.3.1 Précisions sémantiques relatives aux aspects dynamiques

Occurrence entrante et sortante. À l'exécution, pour indiquer l'évolution de l'état du système, nous parlerons d'occurrence d'événement* (au sens large) d'entrée et de sortie.
Au niveau d'abstraction du langage défini dans la section précédente, nous parlerons donc d'une **occurrence entrante** pour qualifier :

- un appel de méthode côté appelé sur une interface de service fournie,
- l'arrivée d'une donnée sur une interface d'entrée de données,
- ou l'arrivée d'un événement notifié sur une interface d'entrée d'événement.

Réciproquement, une **occurrence sortante** qualifie :

- un appel de méthode côté appelant sur une interface de service requise,
- la production (ou l'écriture) d'une donnée sur une interface de sortie de données,
- ou la levée d'un événement sur une interface de sortie d'événement.

Sémantiques implicites attachées aux interfaces multi-liées. Lors de la description des aspects structuraux, nous avons parlé d'interface multi-liée* (cf. page 44) pour qualifier des interfaces impliquées dans de multiples liaisons. Il est ainsi possible de lier une interface d'entrée à différentes interfaces de sortie et réciproquement. D'un point de vue dynamique, une sémantique est attachée à ces schémas d'interconnexion :

- **Côté interface d'entrée :** Pour les interfaces d'entrée de flot, les occurrences en provenance de sources distinctes sont mémorisées (cette « fonctionnalité » est assurée par l'infrastructure d'exécution).
  Pour les interfaces fournies, nous parlerons de partage de services, l'ensemble des opérations définies par l'interface multi-liée sont alors partagés par tous les clients.
- **Côté interface de sortie :** Pour les interfaces de sortie de flots, la **sémantique par défaut** définie à l'exécution est celle d'un *broadcast*. Ainsi, une occurrence sortante sur une telle interface (la production d'une donnée ou la levée d'un événement) sera alors propagée à l'ensemble des consommateurs liés.
  Pour les interfaces requises, la sémantique attachée à l'interface est celle définie par les spécifications FRACTAL, les interfaces *collection*. À l'implantation, cette notion se traduit par des conventions de nommage permettant de différencier les références vers les interfaces fournies. C'est donc au niveau de l'implantation du code métier que l'appel vers l'une des référence est déterminée.
  Il est également intéressant de noter que d'autres sémantiques auraient pu être attribuées

à une interface requise multi-liée, notamment pour prendre en compte des propriétés de sûreté de fonctionnement. Par exemple, pour certaines applications critiques, il est parfois de mise d'invoquer un même service vers différentes implantations dans le but de comparer les écarts entre les résultats ou d'en faire une moyenne. Une telle sémantique peut très bien être prise en charge automatiquement par l'infrastructure d'exécution et de manière transparente pour l'utilisateur.

4.3.2 Description des aspects de contrôle

L'une des principales particularités du « paradigme composant » est d'externaliser et d'abstraire certains détails d'implantation au niveau du langage de description de l'architecture. C'est par exemple le cas de la notion d'interface qui caractérise, par exemple, les services offerts et requis par le composant sous forme de signatures de méthodes.
Dans le contexte de notre démarche, et comme nous l'avons énoncé dans le chapitre 3, l'objectif est d'appliquer ce même principe mais pour caractériser des propriétés liées à certains aspects dynamiques du domaine d'application visé (du « métier visé »). Ces aspects se spécifient au niveau de l'architecture, et illustrent la volonté de définir un langage d'architecture pour lequel les préoccupations propres au domaine visé par TINAP sont réifiées.

Nous appelons « **descripteurs de contrôle*** » les propriétés liées à la dynamique que nous proposons de réifier au niveau architectural. Le mécanisme pour ajouter de telles annotations sur les composants est représenté sur la figure 4.17. Sur cette figure, nous avons fait apparaître les noms des paquetages au sein desquels les méta-classes sont définies.

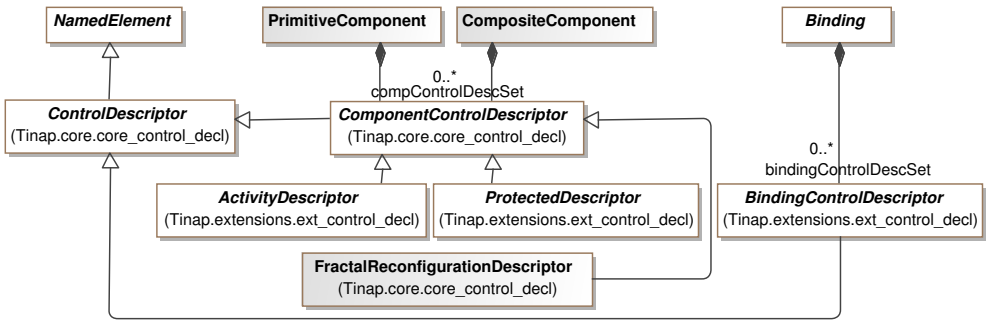


Figure 4.17 – Descripteurs de contrôle.

Les descripteurs de contrôle que nous proposons (*ActivityDescriptor*, *ProtectedDescriptor* et *BindingDescriptor*) permettent de spécifier un **modèle de concurrence** à appliquer sur les concepts architecturaux que nous avons définis. Nous employons ce terme général pour qualifier :

1. les modèles d'exécution, qui correspondent à la manière dont les aspects multi-tâches sont appréhendés au sein du processus de développement et les règles d'activation de ces tâches (ou contrôleurs d'activation),
2. les modèles (ou protocoles) de communication et de synchronisation,
3. et la protection de sections critiques (liée généralement à des mécanismes de synchronisation).

Ces descripteurs s'attachent aux composants ou aux liaisons. Dans le cas des composants, ils permettent de définir une « personnalité » qui leur sera associée : nous parlerons de « **composant actif** » lorsqu'un composant est attaché à un *ActivityDescriptor*, et de « **composant passif** » dans le cas contraire. Lorsqu'un composant passif est attaché à un *ProtectedDescriptor*, nous parlerons

de « **composant protégé** » . Dans le cas des liaisons, ils permettent de préciser les protocoles de communication et de synchronisation qui leur seront attachés.

De façon connexe, les composants que nous envisageons sont potentiellement introspectables et reconfigurables, selon les modalités définies dans les spécifications FRACTAL. Ces propriétés sont configurées par l'intermédiaire de la méta-classe *FractalSpecsDescriptor*.

Dans cette section, nous décrivons les descripteurs élémentaires que nous avons proposés et présentons en détails leurs sémantiques. Il sera alors à la charge de l'infrastructure d'exécution de les préserver au cours du cycle de vie de l'application.

4.3.2.1 Descripteurs de contrôle FRACTAL

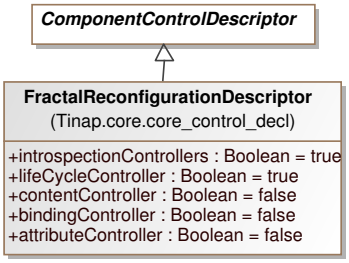


Figure 4.18 – Contrôleurs FRACTAL.

Les capacités d'introspection et de reconfiguration dynamique proposées par les spécifications FRACTAL sont énumérées au sein du descripteur de contrôle de la figure 4.18. Il peut s'appliquer à un composant primitif ou à un composite (à quelques contraintes près, notamment pour le *contentController*).
Dans notre démarche, il faut noter que l'ensemble de ces contrôleurs sont optionnels. En effet, il n'est pas obligatoire que l'infrastructure d'exécution fournisse tous ces services d'ordre non fonctionnel à l'ensemble des composants d'une application. Il est donc du ressort du concepteur de spécifier les capacités de contrôle à attacher à ses composants métier.

Ce descripteur de contrôle est défini au sein du paquetage `core`, nous considérons donc qu'il fait partie du « noyau de concepts » que nous avons caractérisé, il s'emploie potentiellement à tous les niveaux d'abstraction d'un système.

4.3.2.2 Descripteurs de contrôle pour la gestion des activités

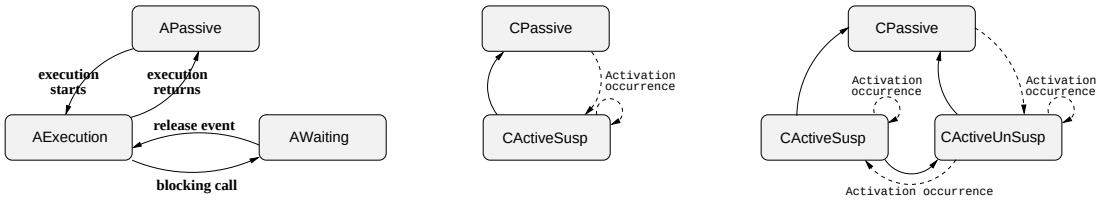
Les descripteurs de contrôle que nous présentons dans ce paragraphe permettent au concepteur d'exprimer à partir de l'architecture définie, quelles seront les entités actives de l'application. En d'autres termes, il s'agit de lui proposer une approche déclarative pour spécifier comment les activités[27] sont projetées sur son architecture métier et leurs règles d'activation. Un composant auquel est attaché un tel descripteur de contrôle sera nommé « **composant actif** ».

Catégories de composants actifs. Il est possible de proposer différents modèles d'exécution à partir des concepts architecturaux que nous avons définis. En premier lieu, nous pouvons faire la distinction entre deux catégories de composant actif que nous nommerons **mono-actif** et **multi-actif**. Un composant mono-actif est un composant attaché à une unique activité. Dans ce cas, à l'exécution pour un instant donné, seul un unique fil d'exécution est susceptible de « traverser son implantation ». Les problèmes de corruption de l'état interne du composant induits par de potentiels accès concurrents sont donc impossibles. En revanche, nous qualifions de multi-actif, un composant pour lequel plusieurs activités lui sont attachées et qui peuvent potentiellement s'activer concurremment.

[27] Dans le contexte de ce document, nous employons le terme « activité » (ou fil d'exécution) au niveau applicatif, c'est-à-dire une suite d'instructions séquentielles. Au niveau système d'exploitation, elles sont alors projetées sur des « tâches » (entités ordonnançables par le système).

Activation. D'un point de vue dynamique, nous définissons la notion d'**interface d'activation*** (*ActivationInterface*) qui permet de spécifier quelle(s) occurrence(s) entrante(s) (cf. page 52) en provenance de l'environnement est susceptible de provoquer une activation de l'activité qui sera attachée au composant métier.

Les graphes informels d'états d'une activité et d'un composant actif sont représentés sur la figure 4.19.



(a) Activity states (b) Mono−active component states (c) Multi−active component states

Figure 4.19 – Graphe d'états (a) d'une activité, (b) d'un composant mono-actif et (c) d'un composant multi-actif.

C'est en fonction des occurrences entrantes d'une interface d'activation (ou plus simplement des demandes d'activation) que s'opère la transition de l'état `APassive` à l'état `AExecution` pour une activité (partie gauche de la figure). Lorsque la suite d'instruction à exécuter retourne, celle-ci est alors de nouveau disponible pour le composant actif auquel elle est attachée. Cette suite d'instructions peut potentiellement effectuer des appels bloquants (état `AWaiting`).
Un composant actif (partie droite de la figure) est dans l'état `CPassive` si l'activité (ou toutes les activités dans le cas d'un composant multi-actif) est (ou sont) dans l'état `APassive`. Si l'activité attachée à un composant mono-actif commence son exécution, celui-ci sera dans l'état `CActiveSusp` (pour *ActiveSuspend*). Si d'éventuelles demandes d'activation surviennent dans cet état, elles seront mises en attente. Il en est de même pour un composant multi-actif, mais qui pourra cependant accepter de nouvelles exécutions d'activités dans la limite de la borne fixée par le concepteur (état `CActiveUnSusp`).
Les activités seront exécutées dans le contexte de tâches éventuellement préemptables du système d'exploitation sous-jacent.Les états des composants actifs sont gérés automatiquement par l'infrastructure d'exécution.

Une interface d'activation est à opposer à une ***`interface passive`***. En effet, le code implanté par une interface passive est alors invoqué dans le contexte d'une activité appelante. Il faut noter que les trois natures d'interface définies (de service, de flot de données ou d'événements) peuvent être utilisées comme interfaces passives.
Le caractère « passif ou actif » d'une interface ne s'applique qu'aux *`interfaces implantées`* (et donc uniquement pour qualifier des interfaces d'entrée).
Dans la suite du document, nous emploierons donc le terme « d'*`interface passive`* » **uniquement pour qualifier une interface d'entrée implantée**, à opposer à une *`interface d'entrée de flot d'interaction interne`*. Nous emploierons de manière similaire les termes « d'*`interface d'activation`* » et « d'*`interface active`* ».

Présentation des descripteurs. Trois exemples de descripteurs de contrôle élémentaires sont donnés figure 4.20. Ils sont structurés de manière identique mais leurs sémantiques diffèrent comme nous le détaillons ci-dessous.

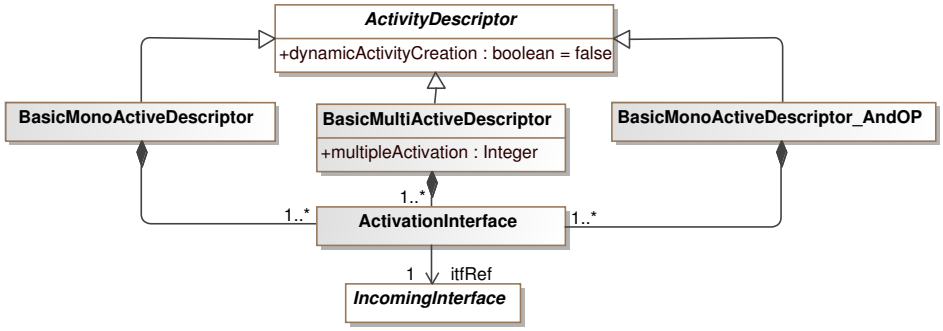


Figure 4.20 – Descripteurs de contrôle élémentaires pour composants mono-actif et multi-actif.

Un tel descripteur permet de définir une ou plusieurs interfaces d'activation qui référencent (association *itfRef*) une interface entrante du composant auquel il est attaché. Il faut donc noter qu'un composant actif est susceptible d'être activé par des occurences entrantes de différentes natures en fonction de ses interfaces d'activations.
Toutes les activités d'une application sont instanciées avant l'étape d'exécution du système [28].

1. Le descripteur de contrôle ***BasicMonoActiveDescriptor*** spécifie qu'à l'exécution, une unique tâche du système d'exploitation sera assignée au composant pour exécuter le(s) bloc(s) de code associé(s) aux interfaces d'activation. Il s'agit donc d'une sémantique « *run-to-completion* » et les demandes d'activation seront ordonnancées par défaut selon FIFO. En d'autres termes, il n'y a pas d'exécution concurrente possible pour le (ou les) bloc(s) de code implanté(s) par le composant.

2. Comme nous l'avons présenté, il est également possible d'attacher plusieurs activités à un même composant (donc multi-actif). Il s'agit du ***BasicMultiActiveDescriptor***.
Il diffère du précédent par l'attribut *multipleActivation* qui permet de spécifier le nombre d'activités attachées au composant actif, ou en d'autres termes le seuil d'exécutions concurrentes (ou parallèles) qui peuvent s'opérer à un instant donné. Ce descripteur possède une sémantique proche d'un *pool de threads* attaché au composant métier.
Il faut noter que dans ce cas, les activations concurrentes peuvent compromettre la cohérence de l'état interne du composant s'il est « avec état »[29].
D'un point de vue conceptuel, remarquons également qu'une telle allocation d'activité « casse » la vision architecturale inhérente au concept de composant logiciel. En un sens, une vue purement structurelle ne suffit pas pour appréhender la dynamique de l'implantation d'un composant : une vue « orientée activité » est nécessaire.

[28] Nous avons tout de même considéré la possibilité pour que ces activités soient instanciées dynamiquement en fonction des occurrences entrantes (attribut *dynamicActivityCreation*). Dans un contexte applicatif non temps réel, cette propriété peut avoir du sens (notamment pour un composant multi-actif), dans le cas contraire, son utilisation doit être interdite.

[29] À ce stade, nous pouvons noter qu'au sein de notre démarche, toutes les dépendances d'un composant avec des fonctionnalités de son environnement doivent être explicitées. Notamment, un composant applicatif ne peut être implanté avec des appels système – pour sérialiser des blocs de code utilisant un mutex par exemple – sans caractériser ce requis au niveau de l'architecture. Nous reviendrons sur cet aspect section 4.6.

À titre d'illustration, la figure 4.21 schématise la différence entre deux traces d'exécution possibles pour un même composant attaché aux deux descripteurs de contrôle présentés ci-dessus.
Nous reprenons l'exemple du composant primitif `CompA` donné figure 4.14 (page 48) et nous considérons que son interface `provItf` est spécifiée comme interface d'activation. Une occurrence entrante sur cette interface correspond donc à une demande d'exécution du bloc de code correspondant (les implantations des méthodes `mA` ou `mB`). La première partie de la figure correspond à une trace d'exécution des méthodes en fonction des occurrences entrantes lorsque le composant est mono-actif, la deuxième partie lorsque le composant est multi-actif dont l'attribut *multipleActivation* est fixé à `2`.

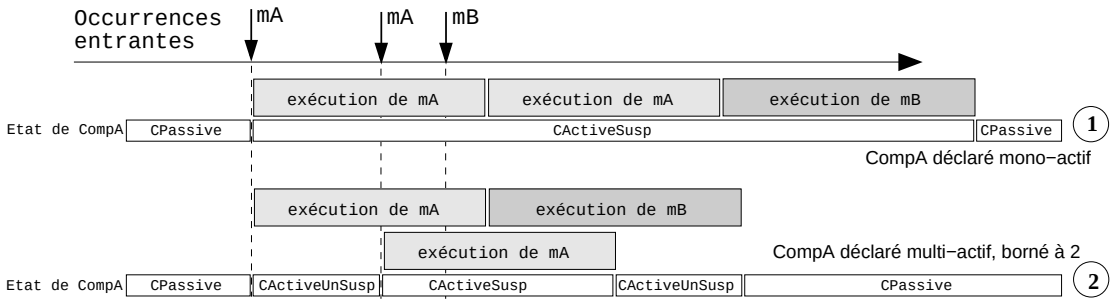


Figure 4.21 – Traces d'exécutions associées à un composant **(1)** lorsqu'il est attaché à un descripteur mono-actif et **(2)** à un descripteur multi-actif.

3. Le troisième descripteur présenté sur la figure 4.20 (***BasicMonoActiveDescriptor_AndOP***) permet aussi de spécifier un composant mono-actif. Cependant, contrairement aux deux précédents, l'activité attachée au composant sera activée par l'infrastructure d'exécution uniquement lorsqu'une occurence entrante pour chaque point d'interaction des interfaces d'activation sera mémorisée. En d'autres termes, la règle d'activation correspond à un opérateur de type *`et logique`* [30]. Un tel descripteur permet par exemple de définir une synchronisation entre un consommateur et plusieurs producteurs.

Les descripteurs de contrôle précédents permettent de définir un composant actif dont l'activation dépend des occurrences entrantes en provenance de l'environnement. Cependant, on peut être amené à définir un composant dont l'activation est demandée périodiquement. Le descripteur de contrôle de la figure 4.22 permet donc d'annoter un composant (dans ce cas mono-actif) pour lequel l'utilisateur peut spécifier une période en paramètre.

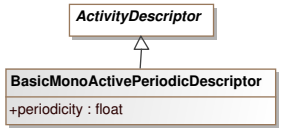


Figure 4.22 – Descripteur simple pour activation périodique.

[30] Dans ce cas, le composant sera attaché à une unique implantation de méthode dont la signature du point d'entrée devra respecter certaines conventions que nous détaillerons dans le chapitre 5.

4.3.2.3 Descripteurs de contrôle pour attacher un comportement aux liaisons

Les liaisons telles que nous les avons décrites (cf. section 4.2.1.3, page 43) permettent d'interconnecter structurellement les composants entre eux par l'intermédiaire des interfaces. Cependant, il est possible de préciser les comportements (ou les patrons d'interaction) qui leurs sont associés en fonction de leurs natures, notamment :

- en terme de blocage ou non blocage côté interface d'entrée et côté interface de sortie (synchronisme/asynchronisme),
- en terme de perte possible ou non des occurrences,
- dans le cas de communications asynchrones, pour limiter la propagation des occurrences au sein de l'architecture en fixant éventuellement des seuils à ne pas dépasser,
- pour préciser leurs sémantiques.

Pour caractériser les comportements des liaisons de flot, nous avons repris certaines propositions CLARA [Faucou *et al.*, 2004] (présenté dans l'état de l'art) qui définit cinq types de comportement pour les liaisons d'événement et cinq types de comportement pour les liaisons de données. C'est parce que cette nomenclature capture pertinemment les besoins des applicatifs en terme de modèles de comportement que nous avons décidé de nous en inspirer.
Les comportements qui s'attachent aux liaisons de service sont celles que l'on retrouve classiquement dans la littérature (synchronisme, asynchronisme, rendez-vous).

A. Comportements des liaisons de flot d'événements

CLARA définit cinq types de « protocoles de signalisation » fixés pour l'émission et la réception d'événements (qui s'appliquent donc aux liaisons d'événement au sein de notre approche) :

- (1) incrémenté et (2) incrémenté borné (asynchrones sans perte),
- (3) mémorisé et (4) fugace (asynchrone avec pertes potentielles),
- et (4) rendez-vous (synchrone).

Protocoles incrémentés : ils permettent de mettre en œuvre une interaction asynchrone entre une interface de sortie et une interface d'entrée. Ils s'implantent par l'intermédiaire d'un simple compteur qui est incrémenté de `1` lorsque l'occurrence d'un signal se produit sur l'interface de sortie et est décrémenté de `1` lorsque l'occurrence est consommée sur l'interface d'entrée. Lorsque ce protocole est non-borné, le fil d'exécution du producteur n'est jamais bloqué. Lorsqu'il est borné, le producteur est bloqué dans le cas où le compteur a atteint la valeur fixée par la borne. Côté interface d'entrée, le consommateur est bloqué lorsque le compteur a une valeur nulle. Les chronogrammes associés à ces protocoles sont donnés figure 4.23.

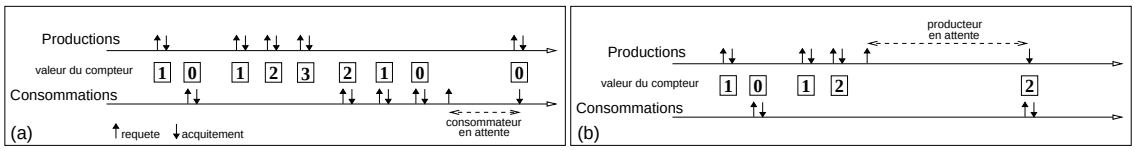


Figure 4.23 – Chronogrammes pour les protocoles incrémentés (a) non borné et (b) borné (à 2) pour une liaison d'événement.

Protocole fugace : pour ce protocole, les occurrences d'événements ne sont pas mémorisées, les pertes sont donc possibles. Le producteur n'est jamais bloqué à l'inverse du consommateur qui l'est toujours jusqu'à ce qu'une production d'événement sur son interface d'entrée se produise. Une occurrence de production est perdue si le consommateur n'est pas effectivement en attente.

Protocole rendez-vous : Il s'agit d'une interaction synchrone entre producteur et consommateur. Ils doivent être tous les deux en attente pour que l'interaction s'établisse.

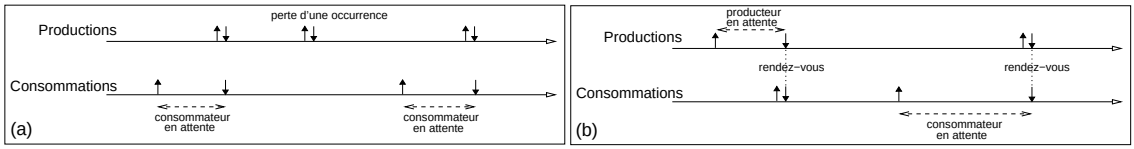


Figure 4.24 – Chronogrammes pour les protocoles (a) fugace et (b) rendez-vous.

B. Comportements des liaisons de flot de données

CLARA définit cinq types de « protocoles de transfert » fixés pour l'émission et la réception des données (qui s'appliquent donc aux liaisons de données au sein de notre approche) :

- (1) boîte aux lettres non bornée et (2) boîte aux lettres bornée (asynchrones sans perte),
- (3) rendez-vous (synchrone),
- rafraîchi (4) avec consommation et (5) sans consommation (asynchrones avec pertes potentielles).

Protocoles boîte aux lettres : les fonctionnements de ces deux protocoles sont exactement similaires à ceux explicités pour les protocoles incrémenté et incrémenté borné. Au lieu d'un simple compteur, les occurrences sont mémorisées dans une boîte aux lettres qui permet de stocker les données produites sur l'interface d'entrée.

Protocoles rafraîchis : ces deux protocoles ne mémorisent qu'une unique donnée côté interface d'entrée. Les pertes sont donc possibles si par exemple deux opérations de production se déclenchent successivement sans opération de consommation intermédiaire. Ils ne sont jamais bloquants pour le producteur. Dans le cas du protocole « avec consommation », le consommateur devient le propriétaire de la donnée qui ne sera alors pas disponible pour les consommations suivantes, à la différence du protocole « sans consommation » pour lequel le consommateur effectue une simple lecture de la donnée, qui reste alors disponible pour d'autres opérations de consommation.

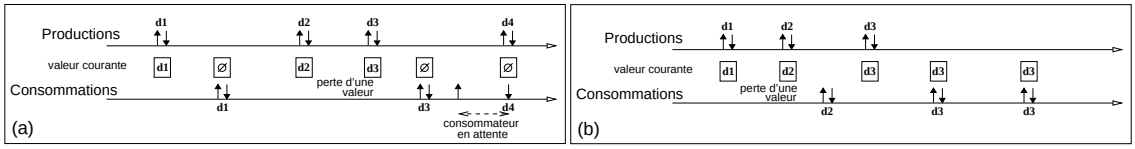


Figure 4.25 – Chronogrammes pour les protocoles rafraîchis (a) avec consommation et (b) sans consommation (la notation ∅ représente le non-rafraîchissement de la variable).

Protocole rendez-vous : il est similaire au protocole rendez-vous pour une liaison d'événement.

Les chronogrammes présentés dans ces deux paragraphes sont repris de [Faucou, 2002], document auquel on peut se reporter pour une présentation plus détaillée et exhaustive de ces protocoles pour les liaisons de flots.

C. Comportements des liaisons de service

Les protocoles définis au sein de TINAP pour les liaisons de service sont les suivants :

- (1) asynchrone non borné et (2) borné,
- (3) synchrone,
- et (4) rendez-vous.

Protocoles asynchrones : comme pour les protocoles incrémenté non borné et borné pour les événements, le client n'est jamais bloqué, sauf dans le cas du protocole borné lorsque la borne fixée est atteinte.

Protocole synchrone : le client est bloqué jusqu'à ce que l'invocation du service côté serveur se termine (retourne).

Protocole rendez-vous : le client est alors bloqué jusqu'à ce que le service invoqué côté serveur commence son exécution.

Dans le cas des protocoles asynchrones et rendez-vous, les méthodes spécifiées par les interfaces ne peuvent définir de paramètre de sortie ni de valeur de retour.

Les interactions de service reposent sur un modèle de communication de type client/serveur, et une « liaison de service », d'un point de vue conceptuel, permet d'expliciter en premier lieu une relation de dépendance fonctionnelle entre services (représentés sous forme de signature de méthodes avec éventuellement des paramètres d'entrée/sortie). Pour cette raison, nous considérons que les occurrences sont toutes mémorisées et que les pertes sont donc impossibles.
Dans le cas d'interaction client/serveur entre deux activités distinctes, la sémantique de ce modèle de communication s'apparente (dans une certaine mesure) à celle d'un envoi de message, ou encore d'un LRPC[31].

Un autre protocole couramment défini pour les interactions client/serveur est le synchrone retardé qui peut s'utiliser pour des méthodes avec des paramètres de sortie. Dans ce cas, le client n'est jamais bloqué lors de son invocation et peut continuer le fil de son exécution tant qu'il n'accède pas à une variable utilisée en paramètre de sortie de cette invocation et dont la valeur dépend de sa terminaison. Lorsque c'est le cas, il se retrouve alors bloqué jusqu'au retour effectif de l'invocation.
Dans notre approche, nous n'avons pas considéré cette possibilité car elle implique, après une analyse statique du code de l'implantation du client, une transformation de celui-ci (si l'on veut que l'implantation de l'interaction soit absolument transparente pour le concepteur final).

D. Spécification des descripteurs d'un point de vue méthodologique

À titre indicatif, l'extrait du méta-modèle associé au descripteur de contrôle permettant de décrire le comportement associé aux liaisons est donné par la figure 4.26

Elle illustre simplement la possibilité d'annoter les liaisons en leur attribuant un comportement spécifique. Pour simplifier, nous considérons qu'un protocole attaché aux liaisons s'applique implicitement à tous les points d'interactions atomiques définis par les interfaces impliquées dans la liaisons.

L'attribut *boundSize* permet de spécifier le seuil maximum d'occurrences qui ne devra pas être dépassé (la valeur `-1` pour illimité) et n'a bien sûr de sens que pour les protocoles bornés que nous avons explicités ci-dessus.

[31] *Lightweight Remote Procedure Call.*

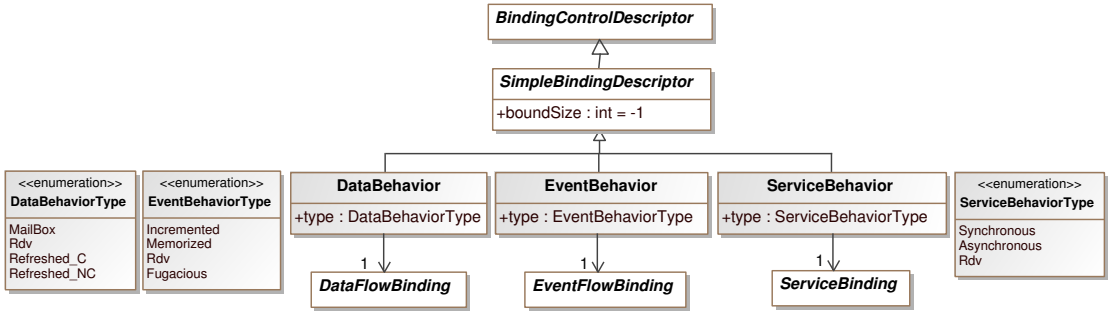


Figure 4.26 – Extrait du méta-modèle permettant d'annoter un comportement spécifique aux liaisons.

Si l'utilisateur ne spécifie pas de comportement spécifique pour une liaison, un comportement par défaut lui sera alors attaché : le protocole « boîte aux lettres » pour une liaison de donnée, le protocole « incrémenté » pour une liaison d'événement et le protocole synchrone pour une liaison de service.

4.3.2.4 Descripteurs de contrôle pour la protection des accès

Un descripteur de contrôle simple défini figure 4.27 permet de sérialiser les accès à un « **composant passif** ». Un composant passif est un composant auquel n'est pas attaché d'activité. Ainsi, les fonctionnalités qu'il implante et qu'il exporte par l'intermédiaire de ses interfaces d'entrées ne peuvent être exécutées que dans le contexte d'une activité attachée à un composant actif appelant.
Nous parlons de « **composant protégé** » lorsque ce descripteur est attaché à un composant.
Dans ce cas, l'infrastructure d'exécution s'assure de protéger les accès à ce composant en assurant qu'un unique (ou un certain nombre fixé par l'utilisateur par l'attribut *accessValue*) fil d'exécution est autorisé à invoquer une opération de ce composant (l'infrastructure d'exécution met donc en œuvre un verrou placé autour du composant qui fait office de ressource partagée protégée).

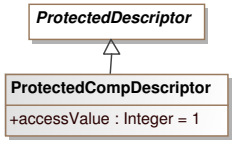


Figure 4.27 – Descripteur pour protéger les accès à un composant passif.

Au sein de cette section, nous avons présenté certains aspects dynamiques « canoniques » pour le domaine d'applications visées par le langage TINAP (activités et leurs règles d'activation, modèles de communication pour les interactions, etc). Il s'agit alors de représenter ces aspects liés à la dynamique du système sur les entités de première classe du langage, c'est-à-dire sur la description architecturale.
Dans la prochaine section, nous présentons de manière informelle les conventions graphiques sur lesquelles nous nous basons dans le contexte de ce document pour représenter ces aspects.

4.3.3 Représentation graphique de la vue dynamique

La figure 4.28 donne les conventions graphiques utilisées dans le contexte de ce document pour représenter une architecture à laquelle sont ajoutés les descripteurs de contrôle qui lui sont attachés (la vue structurelle de cet exemple est donnée figure 4.14, page 48).

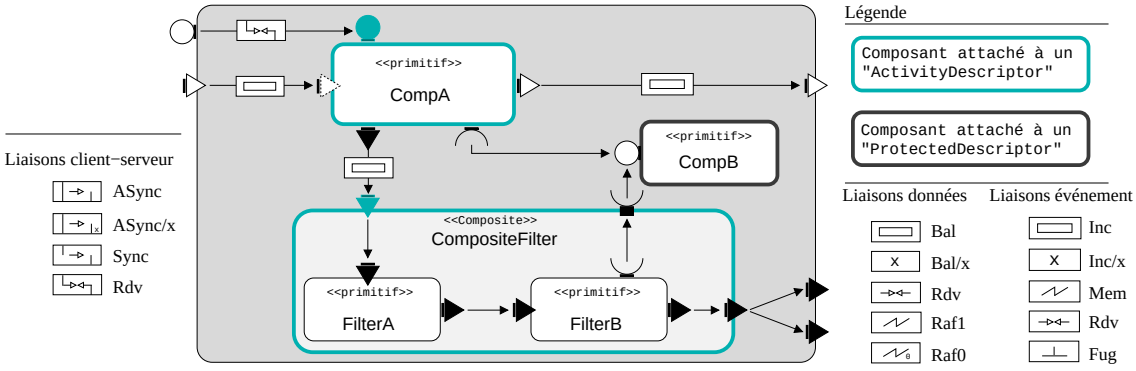


Figure 4.28 – Conventions graphiques utilisées pour représenter la vue dynamique de l'architecture.

Il n'est pas toujours aisé de représenter graphiquement toutes les informations des descripteurs de contrôle. Nous nous contentons de schématiser les éléments les plus importants :

- Les composants auxquels des descripteurs de contrôle sont attachés sont représentés par un contour épais de couleur.
- Par convention nous représentons les interfaces d'activation des composants actifs sur la partie supérieure du composant.
- Les protocoles des liaisons sont représentés par les symboles donnés en légende (ceux des liaisons de flot sont issus de CLARA).

Vocabulaire. Dans la suite de ce document, pour qualifier plus facilement un descripteur de contrôle particulier attaché à un composant, nous employons parfois les termes, en parlant d'un composant : « actif », ou plus précisément « mono-actif », « multi-actif », « périodique », « actif `et logique` », ou encore « protégé ».

4.3.4 Sémantique et contraintes associées à la vue dynamique

Dans cette section, nous cherchons à préciser la sémantique de certains concepts de la vue dynamique et de pointer du doigt certaines contraintes sur leurs utilisations. Elles ne peuvent être énumérées et détaillées de manière exhaustive, l'objectif est plutôt d'engendrer une réflexion sur les aspects qui nous semblent les plus pertinents.

Composant passif et interface. Un composant passif n'est pas attaché à une activité qui lui est propre (et il ne définit donc pas d'interface active).
Un tel composant est donc attaché à au moins une *`interface passive`* (il peut aussi interagir avec son environnement par l'intermédiaire d'*`interfaces d'entrée de flot interne`* et/ou d'interfaces de sorties).
Nous avons déjà évoqué qu'une interface de flot peut être une interface implantée passive

(comme c'est le cas pour les composants `FilterA` et `FilterB` de la figure 4.28 ci-dessus – dans cet exemple, c'est le `CompositeFilter` qui est actif et utilise les interfaces passives de ses sous-composants). En effet, comme nous l'expliquerons dans le chapitre 5, au niveau de l'infrastructure d'exécution, ces interfaces sont simplement projetées sur des interfaces de type client/serveur (pris en charge de manière automatique par le processus de génération de l'infrastructure d'exécution). Dans ce cas, le fil d'exécution appelant exécute le code implanté par ces interfaces comme un simple appel de méthode. La figure 4.29 illustre ces remarques.

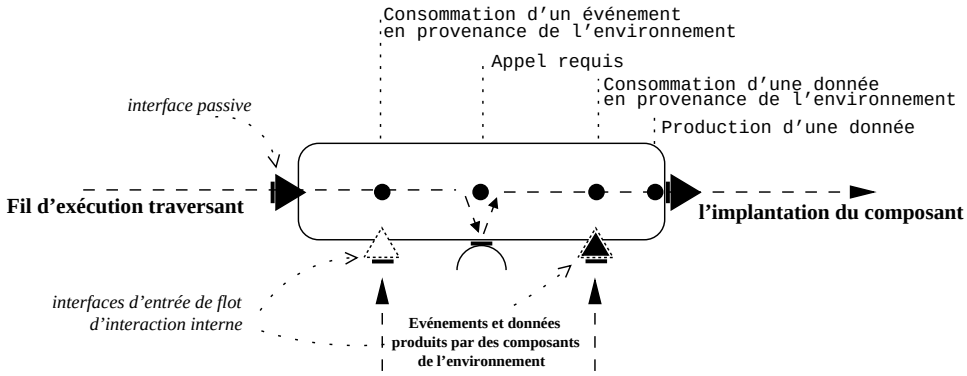


Figure 4.29 – Composant passif et interactions avec son environnement.

Descripteurs de liaisons et interfaces. Les descripteurs de liaisons ne peuvent s'attacher qu'aux liaisons dont l'interface destination est une interface active ou une interface d'entrée de flot d'interaction interne. Dans tous les autres cas, comme nous venons de l'expliquer ci-dessus, une interaction se fait par un simple appel de méthode dans le contexte de l'appelant, il s'agit alors d'une interaction purement synchrone.

Composant actif et interface passive. Un composant actif définit des interfaces d'activation (interfaces actives). Cependant, il peut être également attaché à des interfaces implantées passives. Une telle spécification n'est pas interdite, mais peut potentiellement poser des problèmes de cohérence interne que nous évoquerons notamment dans la section 4.5.

Composant « actif hiérarchique ». Les descripteurs de contrôle d'activité s'appliquent aussi bien aux composants primitifs qu'aux composites. Il est alors concevable d'imaginer un composite actif encapsulant un primitif actif. Même si ce genre d'assemblage est pris en charge par l'infrastructure d'exécution, il n'en reste pas moins que son utilité s'avère hasardeuse dans l'espace de conception métier.

Encapsulation hiérarchique. La remarque précédente amène à considérer cet aspect au sein de TINAP. En considérant la vue structurelle, l'encapsulation est préservée. En effet, de manière classique, un composite exporte ses interfaces à l'environnement sans présenter sa structure interne qui peut alors être omise (ou abstraite – il s'agit d'une approche « boîte noire »). Mais cette considération n'est viable qu'au sein d'un contexte applicatif pour lequel l'ajout d'un composite n'a aucune incidence sur une propriété globale du système en cours de conception. Dans ce cas, seule la compatibilité des interfaces fonctionnelles contraint la composabilité des composants, et non des assertions sur des propriétés telles que la performance, l'occupation des ressources, le respect de contraintes temporelles, etc.

Dans le contexte d'applications temps réel embarquées, pour lequel il est nécessaire de raisonner sur le système en cours d'assemblage en fonction de ces propriétés, une approche « boîte noire » n'est pas envisageable. C'est notamment le cas de TINAP : les aspects dynamiques qu'il est possible d'attacher aux sous-composants d'un composite (spécification de sous-composants actifs ou protégés par exemple) ne sont pas caractérisables au niveau de ce dernier (approche « boîte grise »).

Multiplicité des descripteurs de composants. Les descripteurs de contrôle qu'il est possible d'attacher aux composants ne se combinent pas forcément.
Sur la figure 4.17 (page 53), la multiplicité entre composants et *ComponentControlDescriptor* est fixée à « `0..*` ». Cependant,

1. seul un unique « descripeur FRACTAL » et/ou « protégé » peut s'attacher à un composant donné.
2. Le « descripteur FRACTAL » est orthogonal aux descripteurs « actif » et « protégé ».
3. Les descripteurs « actif » et « protégé » sont exclusifs.
4. Il est possible cependant de définir plusieurs descripteurs « actif » à un même composant. Par exemple la définition de deux descripteurs « mono-actif » attachés à des interfaces d'activation différentes (dans ce cas, la sémantique est proche d'un composant multi-actif, c'est-à-dire auquel est attaché deux activités qui peuvent s'exécuter de manière concurrentes, mais avec des points d'entrées différents) ; ou encore un descripteur « périodique » et un un descripteur « actif », etc. Il faut cependant noter, que même si ces combinaisons sont supportées sous certaines conditions, et qu'elles peuvent s'avérer nécessaires dans des cas simples, leur utilisation dans le cas général résulte d'une mauvaise conception en terme « d'encapsulation fonctionnelle ».

Cas général. D'autres contraintes sémantiques subsistent sur la manière de personnaliser les structures TINAP en les annotant par les descripteurs présentés. De façon générale, même si les descripteurs de contrôle proposés sont élémentaires, il peut s'avérer difficile de « valider » toutes les combinaisons possibles. Cela vient notamment du fait que le modèle de structure TINAP est hiérarchique et ne contraint pas le concepteur à une encapsulation fonctionnelle déterminée.

4.3.5 Bilan sur la vue dynamique et de contrôle

L'objectif de notre démarche est de placer la vue structurelle de l'architecture à l'« épicentre » du cycle de conception du logiciel. C'est donc sur ces entités de première classe, manipulées à ce niveau d'abstraction (composants, composites, interfaces, liaisons, etc), que nous nous sommes intéressés à représenter certains aspects dynamiques de la logique applicative. Ces propriétés qualifient notamment le « modèle de concurrence » (cf. section 4.3.2) proposé par TINAP.
Ces aspects ne sont manipulés par le concepteur que de manière déclarative, il s'agit d'annoter la vue structurelle pour obtenir un « modèle dynamique de l'architecture ». Ce modèle dynamique est transverse au modèle structurel (qui exprime les aspects fonctionnels) – une vue comportementale (présentée page 77) permet à ce titre de représenter les spécifications du concepteur sous forme d'un modèle de tâches. Il lui est aussi découplé : d'un point de vue conceptuel, différents « modèles dynamiques » peuvent être spécifiés à partir d'un même modèle structurel et d'un point de vue pratique, ces modèles sont mis en œuvre automatiquement autour du code fonctionnel. Par extension, ce patron de conception consistant à ajouter des propriétés sur les entités architecturales de telle sorte à personnaliser la structure logicielle pourrait s'étendre à différents domaines applicatifs, cela consiste à utiliser les concepts canoniques des ADL pour les décliner en entités spécifiques au domaine visé.

4.4 Vues implantations

Dans cette section, nous présentons ce que nous qualifions de « vues implantations » (présentées section 4.1.1, page 36) pour les composants primitifs. Elles sont caractérisées par :

1. une description abstraite de la structure de l'implantation en fonction des entités définies par le composant au sein de la vue structurelle. Il s'agit de la « vue implantation externe » présentée dans la prochaine section (4.4.1).
2. l'implantation elle-même des primitifs (définie en langage C), appelée « vue implantation interne » et présentée section 4.4.2 (page 68).

4.4.1 Vue implantation externe

4.4.1.1 « Encapsulation primitive » et implantation, définitions

Un composant primitif est un composant qui encapsule du code – textes et données – écrit dans un langage de programmation et de granularité quelconque. Cette granularité dépend du degré de modularité recherché par le concepteur et se caractérise par une forte cohésion interne au composant en termes de fonctionnalités.

Un composant exporte à son environnement, par l'intermédiaire de ses interfaces d'entrées, l'ensemble des points d'interactions atomiques avec lesquels il est susceptible d'interagir.
Dans le cas d'un composant primitif, nous parlerons plus précisément de ***points d'entrées de séquence**** fournis par son implantation (ou plus simplement de ***points d'entrée*** pour les fils d'exécutions, typiquement un ensemble d'implantations de méthodes). Une ***séquence locale**** est un bloc de code insécable pour le composant qui l'encapsule, il correspond à une suite séquentielle d'instructions[32].

Nous faisons la différence pour les composants primitifs entre une ***implantation mono-séquence**** et une ***implantation multi-séquence****. Dans le premier cas, le composant est implanté par un unique bloc de code séquentiel. Dans le deuxième cas, le primitif est implanté par de multiples blocs de code, et fournit alors de multiples points d'entrée.

Bien entendu, une séquence est susceptible d'interagir avec son environnement par l'intermédiaire des interfaces de sorties, pour invoquer une méthode requise, écrire une donnée ou encore produire un signal. Par ailleurs, comme nous l'avons explicité dans le chapitre précédent, une séquence peut aussi interagir par l'intermédiaire des *interfaces d'entrée de flot d'interaction interne*, pour consommer une donnée ou un signal. Il faut noter qu'au sein même d'une séquence, ces points d'interaction peuvent devenir des ***points de synchronisation**** avec l'environnement en fonction des comportements attachés aux liaisons.
Au sein de la vue architecture, nous avons fait la distinction entre interfaces « implantées » et « d'interaction interne ». À ce niveau d'abstraction, nous préserverons la distinction entre ces deux types de points d'interactions : *les interfaces d'interaction interne* se projettent en ***points d'interaction interne à une séquence**** et les *interfaces implantées* se projettent en *points d'entrée de séquence*.
Ces aspects sont schématisés figure 4.30 (à partir de l'exemple du primitif `CompA` de l'exemple donné page 62).

[32] Une séquence peut s'implanter avec les structures de contrôle courantes (boucles, tests).

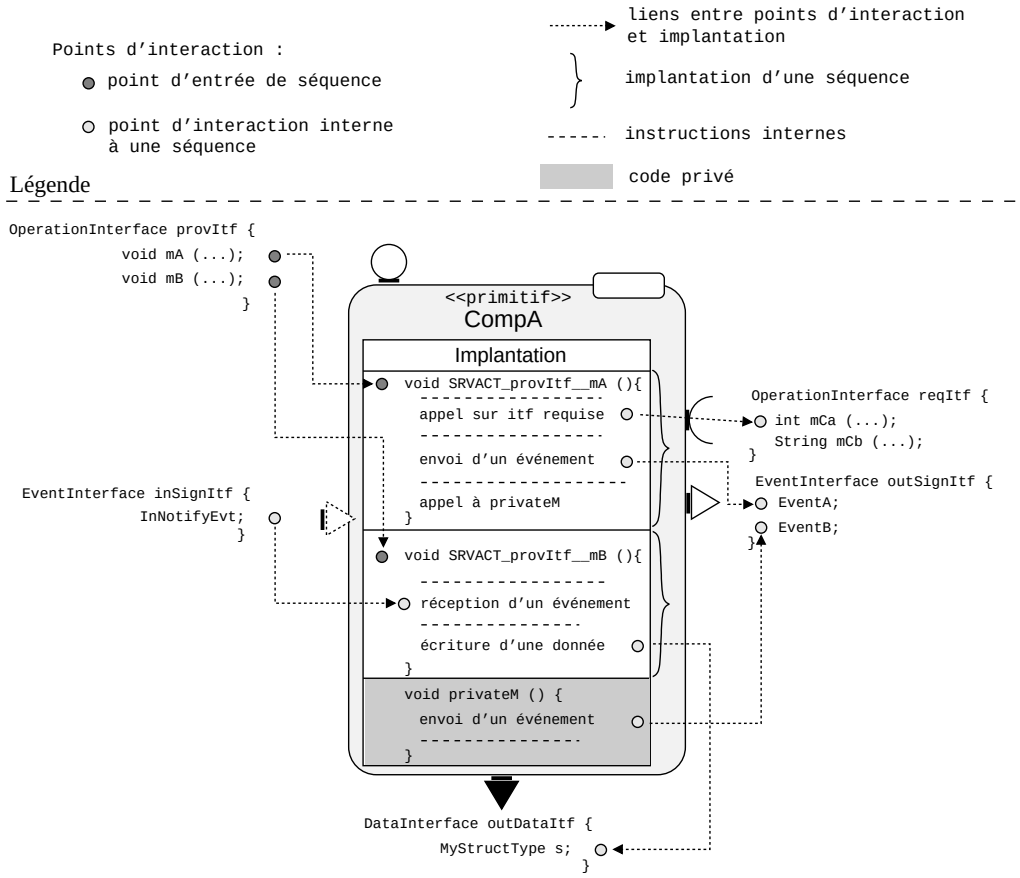


Figure 4.30 – Exemple de composant primitif encapsulant une implantation multi-séquence : projection des points d'interactions atomiques en points d'entrée et d'interaction interne de séquence.

À ce stade de la description de l'implantation d'un primitif, nous pouvons préciser quelques définitions :

- D'un point de vue comportemental, nous qualifions une ***séquence puits****, une séquence qui n'invoque aucune méthode d'une interface requise et ne produit aucun signal ou donnée d'une interface de sortie de flot à destination de son environnement (l'exemple ci-dessus ne présente pas de séquence puits).
- Toujours d'un point de vue comportemental, nous définissons une ***séquence autiste****, une séquence qui ne communique en aucune manière avec son environnement. Une telle séquence est forcément une séquence puits mais non l'inverse (une séquence puits peut être attachée à des points d'interactions internes entrants – réception d'une donnée ou consommation d'un événement).
- D'un point de vue dynamique, nous parlerons aussi de ***séquence basique**** et de ***séquence étendue**** (pour reprendre la terminologie des tâches OSEK [OSEK, 2003]). Une séquence basique, à la différence d'une séquence étendue, est une séquence qui n'invoque aucun service susceptible de provoquer une attente de l'activité qui l'exécute (un blocage dans l'attente d'une donnée sur une interface d'entrée par exemple). Cela correspond à une activité dans l'état `AWaiting` représenté figure 4.19 (page 55). Bien entendu, cette caractérisation dépend des descripteurs de contrôle attachés aux liaisons.

Ces définitions se transposant également à l'échelle d'une interface et d'un composant, nous parlerons par exemple d'*`interface puits`* ou de *`composant puits`* si toutes les séquences qu'elle ou il implante sont des séquences puits.

4.4.1.2 Méta-modélisation de la vue implantation externe

À partir des explications et définitions formulées ci-dessus, nous nous sommes attachés à représenter sous forme abstraite la structure de l'implantation du contenu des composants primitifs. L'extrait du méta-modèle est donné figure 4.31.

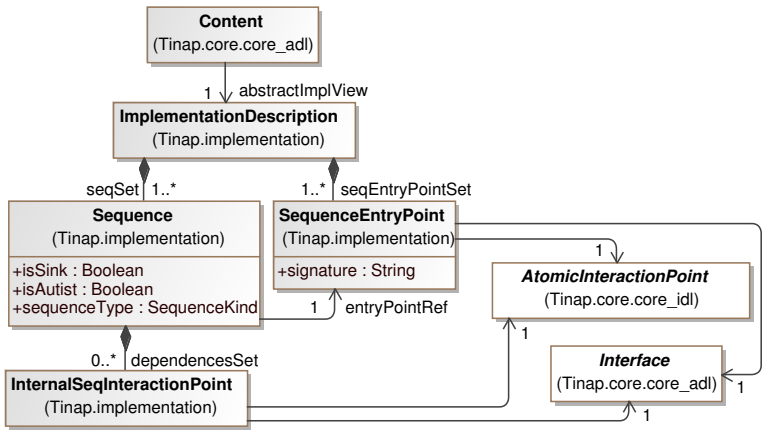


Figure 4.31 – Extrait du méta-modèle pour abstraire la structure interne d'une implantation.

Les informations ainsi représentées ne sont pas renseignées par le concepteur de l'application mais par un processus automatique à partir d'une analyse du code source des implantations du contenu du primitif. Les détails concernant ce processus seront explicités page 73.

Cet extrait de méta-modèle[33]permet simplement de représenter une vue abstraite de l'implantation des composants en prenant en compte les informations telles qu'elles sont schématisées sur la figure 4.30. Elle permet de faire le lien entre l'implantation (de bas-niveau) et les concepts architecturaux (la vue structurelle, de haut niveau). À ce niveau de description, la structure de l'implantation est donc représentée par un ensemble de séquences dont chacune d'elle est attachée à un point d'entrée (*SequenceEntryPoint*). Les interactions avec l'environnement du primitif pour une séquence donnée sont matérialisées par la méta-classe *InternalSeqInteractionPoint*. Le lien entre l'implantation et la vue structurelle est simplement représenté par les liaisons dirigées vers les points d'accès atomiques et les interfaces attachées au composant.

Implicitement, cette représentation permet aussi de modéliser les dépendances potentielles qui subsistent entre les interfaces du composant, ou encore, plus précisément, entre points d'interactions. Par exemple, à partir de l'implantation schématisée figure 4.30, pour exprimer la dépendance entre `mB` et `InNotifyEvent` ou encore l'indépendance entre `mB` et les interfaces `reqItf` et `outSignItf`.

Il est également important de noter que les termes de « séquence » et de « point d'entrée de séquence » ne s'appliquent pas aux méthodes privées qui sont susceptibles d'être implantées par le contenu. Ainsi, à ce niveau de description de l'implantation, ces méthodes ne sont pas représentées, seuls les points d'interactions avec l'environnement le sont. Par exemple, toujours en rapport avec la figure 4.30, nous considérons que l'implantation de la méthode `privateM` est

[33]D'autres informations sont définies pour cet extrait de méta-modèle. Elles serons spécifiées sur la figure 4.37.

« incluse » dans la séquence de `mA`.
À ce niveau de description, on ne cherche pas non plus à représenter le comportement interne des séquences, cet aspect est représenté par la « vue comportement » (présentée section 4.5).

4.4.2 Vue implantation interne

4.4.2.1 Correspondance entre l'ADL FRACTAL et implantation des « primitifs NUPTSE »

On peut qualifier le modèle de composant FRACTAL comme un modèle « intrusif » relativement aux implantations des composants. En effet, une spécification architecturale d'un primitif ne peut être totalement découplée du code source qui l'implante. Sans entrer dans les détails, cela se traduit par la nécessité de faire correspondre les noms des méthodes spécifiées au sein de l'ADL avec leurs implantations ou encore pour la mise en œuvre de certains aspects de contrôle que le développeur du code métier doit prendre en charge.

Au sein de la chaîne de compilation THINK NUPTSE [Lobry and Polakovic, 2008], l'implantation d'un composant primitif repose sur l'utilisation de certaines conventions d'écriture (ou encore de mots clefs, mais qui respectent la syntaxe du C) sous le terme PPL[34]. Elles permettent d'assurer le lien (et une certaine lisibilité pour le concepteur) entre l'implantation et la spécification architecturale du primitif et se traduisent de la manière suivante :

1. Implantation d'une méthode d'une interface fournie :
   `[typeRetour] SRV_[nomItf]__[nomMéthode] ([listeParamètres]) {}`
2. Appel de méthode d'une interface requise :
   `[typeRetour] CLT_[nomItf]__[nomMéthode] ([listeParamètres]);`
3. Accès à un attribut attaché au composant :
   `ATT_[nomAttribut]`
4. Appel à une méthode privée implantée par le composant :
   `[typeRetour] PRV_[nomMéthode] ([listeParamètres]);`

4.4.2.2 Correspondance entre concepts TINAP et implantation des primitifs

Dans notre approche, en plus des interfaces client-serveur utilisées classiquement dans THINK , nous avons défini les interfaces de flot ainsi que la notion d'interface d'activation.
Nous avons aussi introduit un ensemble de mots clef, en se basant sur le même principe que PPL pour assurer le lien entre l'implantation d'un composant TINAP et ces concepts architecturaux. Les préfixes ainsi définis sont représentés sur la tableau 4.1.

Ainsi, les points d'interactions internes à une séquence pour les interfaces de flots se projettent sur de simples appels de méthodes à l'implantation, préfixés par `CLTREAD`, `CLTCONSUME`, `CLTWRITE` et `CLTSEND` respectivement pour lire une donnée ou consommer un événement en provenance de l'environnement et pour écrire une donnée ou lever un événement à destination de l'environnement. Pour ces mots clef, les conventions d'écriture se définissent de la manière suivante :

5. `void CLTWRITE_[nomItf]__[nomDonnée] ([paramètreDonnée]);`
6. `void CLTREAD_[nomItf]__[nomDonnée] ([paramètreDonnée]);`
7. `void CLTSEND_[nomItf]__[nomEvénement] ();`
8. `void CLTCONSUME_[nomItf]__[nomEvénement] ();`

[34] *Primitive component Programming Language.*

	Interfaces de sortie	Interfaces d'entrée	
		d'interactions interne	implantées
Interfaces services	CLT (PPL)		SRV (PPL) **SRVACT**
Interfaces données	CLTWRITE	CLTREAD	SRVREAD **SRVACTREAD**
Interfaces événements	CLTSEND	CLTCONSUME	SRVCONSUME **SRVACTCONSUME**
Cas particuliers			**SRVACTPERIODIC** **SRVACTANDOP**

TAB. 4.1 – Préfixes des « mots clefs » utilisés pour l'implantation des primitifs TINAP.

Les préfixes CLTREAD et CLTCONSUME correspondent à l'entrée d'une donnée ou la consommation d'un événement sur les *interfaces d'entrée de flot d'interaction interne*. Il s'agit donc d'une demande explicite de la part du code métier pour lire ou consommer une occurrence entrante. À ce niveau d'abstraction, il s'agit donc d'un appel de méthode à l'infrastructure d'exécution du composant qui implante la liaison (cet aspect sera traité dans le chapitre 5).

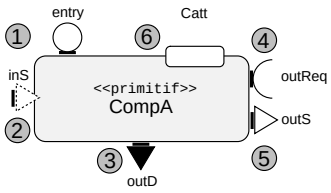
SRVREAD et SRVCONSUME caractérisent des points d'entrée de séquence (donc de simples signatures d'implantation de méthode) pour des interfaces passives d'entrée de données ou d'événements.
Par exemple, le composant FilterB (figure 4.28, page 62) est un composant (passif) attaché à une interface d'entrée de donnée passive. Dans ce cas, le composant implante autant de méthodes SRVREAD qu'il y a de données définies sur son interface d'entrée et leurs implantations seront invoquées dans le contexte d'une activité d'un composant actif appelant. SRVACT, SRVACTREAD et SRVACTCONSUME caractérisent des implantations de méthodes attachées à des points d'interaction définies pour des interfaces d'activation[35].
SRVACTPERIODIC et SRVACTANDOP sont des cas particuliers pour caractériser des points d'entrée, respectivement pour un « primitif actif périodique » et pour un point d'entrée associé à un composant attaché à un descripteur d'activité « `et logique` » (page 57).

À titre d'illustration, la vue implantation interne du primitif CompA est donnée figure 4.32.
Il y a donc des conventions que l'implantation doit respecter en fonction des concepts architecturaux[36]. Elles sont toutes vérifiables automatiquement comme nous l'expliquerons dans le chapitre suivant.

4.4.3 Bilan sur les vues implantation

Dans notre approche, nous promouvons une démarche « orientée langage de programmation » pour implanter le comportement des composants. Le choix du langage C est justifié par le domaine de l'embarqué qui nécessite d'implanter des fonctionnalités de bas-niveau. Il s'agit également d'un langage largement utilisé dans l'industrie.
Cependant, nous nous sommes attaché à définir une forte corrélation entre le comportement interne d'un composant et les concepts de plus haut niveau issus de l'architecture, réduisant ainsi l'écart sémantique entre ces deux niveaux d'abstraction.

[35] Ces mots clef sont employés pour assurer une certaine « lisibilité » au sein du code mais le choix entre interface active ou passive se fait au niveau de la description architecturale et non au niveau de l'implantation.
[36] Des conventions sont également établies pour la projection des spécifications IDL à l'implantation.



Interfaces définies par CompA et leurs signatures associées :

1. Interface **entry**

```
OperationInterface provItf {
  void mA (...);
  void mB (...); }
```

2. Interface **inS**

```
EventInterface inSignItf
  {inNotifyEvt;}
```

3. Interface **outD**

```
DataInterface outDataItf
  {MyStructType s;}
```

4. Interface **outReq**

```
OperationInterface reqItf {
  int mCa (...);
  String mCb (...); }
```

5. Interface **outS**

```
EventInterface outSignItf {
  EventA;
  EventB; }
```

6. Interface **Catt**

```
AttributeInterface compAAtts
  {boolean b;}
```

```
1  // Point d'entrée de séquence de "l'opération active" mA
2  void SRVACT_entry__mA (...) {
3    /* -- Bloc de code métier A -- */
4    for (...) {
5      // Appel sur une interface requise
6      CLT_outReq__mCa (...);
7    }
8    // Envoi d'une occurrence d'événement
9    // de type EventA
10   CLTSEND_outS__EventA();
11   /* -- Bloc de code métier B -- */
12   // Appel à une méthode privée
13   PRV_privateM ();
14 }
15 // Point d'entrée de séquence de
16 // "l'opération active" mB
17 void SRVACT_entry__mB (...) {
18   /* -- Bloc de code métier D -- */
19   // Consommation d'une occurrence
20   // d'événement de type InNotifyEvent
21   CLTCONSUME_inS__InNotifyEvt();
22   /* -- Bloc de code métier E -- */
23   // Ecriture d'une donnée de type s
24   CLTWRITE_outD__s (...);
25 }
26 void PRV_privateM() {
27   // Lecture d'un attribut
28   if (ATT_b == TRUE)
29     CLTSEND_outS__EventB();
30   else
31     /* -- Bloc de code métier C -- */
32 }
```

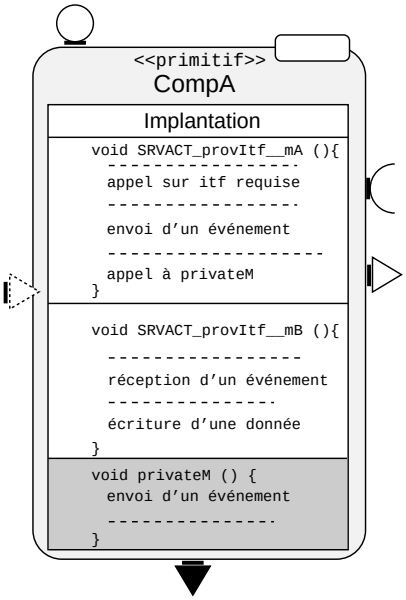


Figure 4.32 – Vue structurelle du primitif `CompA` (partie supérieure) et vue implantation interne de son contenu (partie inférieure).

Nous avons également cherché à qualifier plus précisément certaines propriétés de l'implantation à l'aide d'un vocabulaire. Certes, il ne s'agit que de mots (c'est-à-dire qu'ils ne jouent pas de rôle dans le processus de conception), mais cette démarche prend également son sens dans la volonté de préciser la sémantique d'une implantation relativement aux concepts architecturaux (ou de caractériser un code encapsulé dans un composant).
Enfin, la « vue implantation externe » offre une abstraction de l'implantation des composants primitifs. On pourrait également qualifier d'« interface » (au sens général du terme) ces informations[37] nécessaires et suffisantes pour intégrer dans une application un composant primitif dont

[37] Le méta-modèle associé à cette vue, présenté figure 4.31, ne donne pas toutes les informations que nous avons décidé d'abstraire. Nous y revenons à la prochaine section.

le code est compilé.

L'intérêt d'une démarche orientée composant est d'être en mesure d'abstraire les détails d'implantation des primitifs sous forme de propriétés réifiées au niveau de l'architecture. Il s'agit alors d'être en mesure de raisonner sur ces abstractions du système. Dans la prochaine section, nous présentons une manière de représenter sous forme abstraite le comportement des composants.

4.5 Vues comportementales

Les aspects comportementaux d'une application TINAP se caractérisent à différents niveaux d'abstraction :

- au **niveau implantation**, il consiste à extraire les chemins d'exécutions qui transitent au sein de l'implantation.
- au **niveau composant primitif**, il s'agit alors de représenter deux informations sous forme de traces d'exécution : (1) l'ensemble des interactions pour lesquelles le composant est impliqué avec son environnement via ses interfaces, et (2), les blocs de code internes susceptibles d'être exécutés entre ces interactions.
- au **niveau architecture**, l'objectif est d'être en mesure de raisonner sur certains aspects du comportement global de l'application en tenant compte des spécifications des vues structurelle et dynamique.

Les extractions des informations à ces différents niveaux d'abstraction se font de manière automatique à partir d'analyses statiques des arbres de syntaxe abstraite des implantations des composants et des spécifications de l'architecture de l'application.
Les objectifs de ce processus d'extraction sont les suivants :

- D'un point de vue modélisation, il s'agit de construire automatiquement des représentations de haut niveau des implantations des composants et de leurs comportements à ces différents niveaux d'abstraction. À l'étape de conception, elles permettent de représenter l'application selon d'autres points de vue [IEEE, 2000] que celui exclusivement orienté architecture.
  De plus, des annotations d'ordre non fonctionnel sur ces vues comportementales permettent d'enrichir les spécifications du concepteur pour par exemple préciser des contraintes temporelles au niveau applicatif.
- Il permet également de vérifier certaines propriétés pour assurer la cohérence des spécifications du concepteur comme nous le détaillons dans cette section.

La figure 4.33 présente l'articulation globale des différentes étapes d'extraction du comportement. La première transition est fonction des implantations des composants (le `Content` de la figure 4.3 page 39 spécifié par le concepteur), et il ne peut bien entendu s'effectuer qu'à partir d'une spécification structurelle **valide et instanciable**.
La deuxième transition se fait directement à partir des informations sérialisées sous forme de modèles.

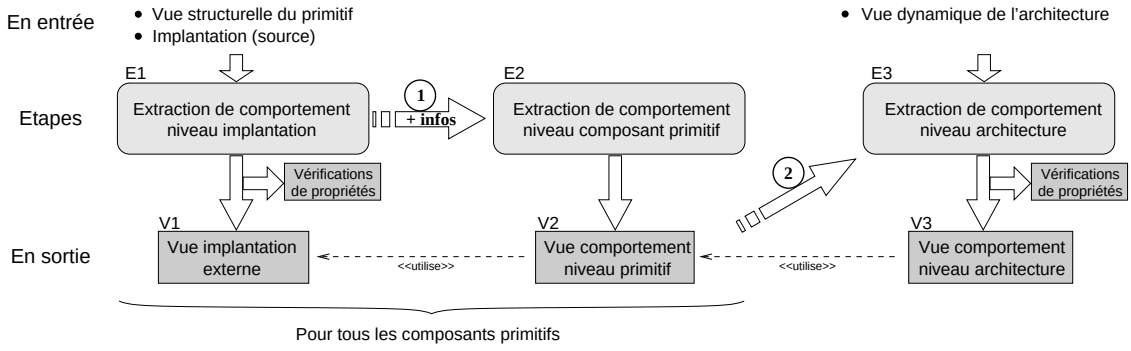


Figure 4.33 – Étapes d'extraction du comportement des descriptions TINAP.

La prochaine section présente succintement un extrait d'application sur lequel nous nous basons pour présenter les aspects comportementaux de notre approche, détaillés dans les sections qui suivent.

4.5.1 Exemple d'architecture TINAP

Pour illustrer nos propos relativement aux aspects comportementaux développés dans cette section, nous nous basons sur l'architecture schématisée figure 4.34. Il s'agit d'un extrait de l'étude de cas présentée chapitre 6 (page 108).

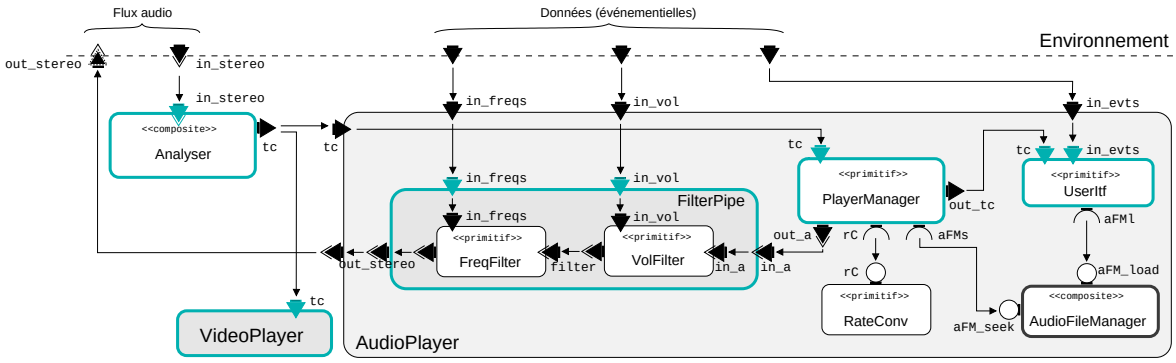


Figure 4.34 – Exemple d'architecture TINAP (vue dynamique présentant les noms des interfaces).

Cette application consiste à récupérer un flux d'entrée audio (dans lequel sont codées des informations de contrôle), en provenance d'une carte son. Ce flux est matérialisé par des interfaces de flot de données. Après une étape d'analyse de ce flux, il est possible de contrôler la lecture de différentes sources multimédias, comme un lecteur audio par exemple, schématisé figure 4.34 par le composite `AudioPlayer`, ou encore un lecteur vidéo (composant `VideoPlayer`). À son tour, le lecteur audio produit un flux de sortie destiné à la carte son (fonctionnant en *full-duplex*). D'autres événements en provenance de l'environnement, matérialisés par des interfaces de données, sont susceptibles d'interagir avec l'application (comme les liaisons établies le montrent).
Cette figure présente une vue dynamique simplifiée de l'application. En effet, l'architecture met en évidence les composants actifs et protégés mais nous ne présentons pas les protocoles spécifiques attachés à certaines liaisons. Tous les composants actifs de cette vue dynamique sont des composants mono-actifs.
Dans le contexte de cette section, l'objectif n'est pas de détailler précisément les fonctionnalités de l'application mais de disposer d'un exemple suffisamment complet pour caractériser les aspects comportementaux de TINAP.

4.5.2 Comportement de niveau implantation

À partir du parcours de l'AST (*Abstract Syntax Tree*) des fichiers sources (ou vues implantations internes des composants), le processus d'analyse se charge d'extraire les chemins d'exécution susceptibles de traverser l'implantation des composants primitifs sous forme d'automates. Les transitions sont alors étiquetées par les nœuds de l'AST nécessaires à la construction des vues comportementales (les ***`actions internes atomiques`***[38] de l'implantation des composants : déclarations de méthodes, de structures de données, lecture/écriture du contenu d'une variable, etc) et représentent l'ensemble des chemins d'exécutions possibles en fonction des structures de contrôle (boucles, tests, *switchs*, *goto*).
À ce niveau d'abstraction, nous sommes alors en mesure de vérifier certaines propriétés relativement à la spécification de la vue structurelle du composant, notamment :

– Que le code respecte les contrats spécifiés par la vue structurelle, notamment en implantant l'ensemble des points d'entrée de séquence et en vérifiant que les occurrences de flot sont

[38] atomiques au niveau du langage de programmation.

potentiellement consommées et produites, respectivement pour les interfaces d'entrée de flot d'interaction interne et les interfaces de sortie de flot.

- Que les types de données spécifiés qui vont être produits et consommés sont conformes.
- Que les données passées en paramètres des appels à destination de l'infrastructure d'exécution ou de l'environnement respectent les conventions de projection de l'IDL.
- Que l'implantation ne contient pas de code mort (c'est-à-dire des blocs de code non accessibles pour tous les chemins d'exécution possibles).

De plus, cette analyse de l'implantation permet de détecter (et de mémoriser comme nous l'expliquerons dans la section suivante) les variables globales partagées par les différentes séquences implantées par le primitif[39]. Elle permet également de vérifier si une implantation est « avec état » ou « sans état ». Une implantation « avec état » définit des variables globales accédées en écriture lors de l'exécution de l'application.
Nous ne détaillerons pas davantage ce processus d'analyse, les explications relatives à son implantation sortent du contexte de ce document.

Cette première passe d'analyse s'effectue sur l'ensemble des composants primitifs de l'application. Les graphes de comportement ainsi extraits ne sont pas représentés à l'étape de conception. Il s'agit d'une étape intermédiaire qui vérifie les propriétés énumérées ci-dessus (levant potentiellement des erreurs lors d'une vérification à l'étape de conception ou en amont de l'étape de compilation) et permet notamment de construire la vue implantation externe présentée figure 4.31 et les vues comportementales de niveau composant et architecture présentées ci-dessous.

4.5.3 Comportement de niveau composant primitif

À ce niveau d'abstraction, le comportement est représenté par :

1. les séquences d'occurrence d'événements susceptibles d'être échangés entre l'implantation d'un composant et son environnement par l'intermédiaire de ses interfaces. Il s'agit donc d'extraire l'ensemble des traces d'exécution des actions observables des composants.
2. les états entre ces séquences qui correspondent à l'exécution de blocs séquentiels de code internes à l'implantation du composant.
3. Les accès aux variables partagées.

4.5.3.1 Illustration

L'automate produit en sortie du processus d'extraction du comportement se construit à partir de celui obtenu suite à l'analyse de niveau implantation décrit ci-dessus mais :

1. il se limite à ne présenter que les interactions avec l'environnement du composant (ou actions observables).
2. il caractérise les états qui correspondent à des blocs séquentiels de code internes mais observés de manière atomique à ce niveau d'abstraction. Un tel bloc correspond à une concaténation de l'ensemble des actions internes atomiques définies ci-dessus entre deux actions observables et nous le qualifions de ***`pas d'exécution local`*** au composant.

[39] Au sein des implantations THINK, les variables globales au primitif sont stockées dans une structure de données spécifique. Le développeur doit suivre cette convention d'implantation, ce qui facilite notamment l'analyse des accès à ces variables.

Indéterminisme et itération. Une trace d'exécution ainsi représentée n'est pas exclusivement séquentielle. En effet, les actions observables d'un composant peuvent être fonction des structures de contrôle internes à l'implantation. Deux possibilités sont à considérer :

- L'indéterminisme, qui correspond à une alternative au sein d'une trace d'exécution, typiquement si une action observable est fonction d'un test. Parmi un ensemble de chemins d'exécution, un seul sera alors choisi : un état de l'automate sera associé à plusieurs transitions de sortie.
- L'itération, qui correspond à une trace d'exécution pouvant se répéter une ou `n` fois, si une action observable est définie au sein d'une boucle. Cette situation est alors caractérisée par un cycle au sein d'une séquence de l'automate.

Une transition entre deux états peut être également sans étiquette (notée « τ ») pour caractériser une « transition silencieuse » (c'est-à-dire sans répercussion sur l'environnement du primitif). Par exemple, la figure 4.35 représente l'automate obtenu à la suite de l'analyse de l'implantation donnée figure 4.32 (ici limité à la méthode `mA`).

```
1  void SRVACT_entry__mA (...) {
2    /* -- Bloc de code métier A -- */
3    for (...) {
4      CLT_outReq__mCa (...);
5    }
6    CLTSEND_outS__EventA();
7    /* -- Bloc de code métier B -- */
8    PRV_privateM ();
9  }
10 void PRV_privateM() {
11   if (ATT_b == TRUE)
12     CLTSEND_outS__EventB();
13   else
14     /* -- Bloc de code métier C -- */
15 }
```

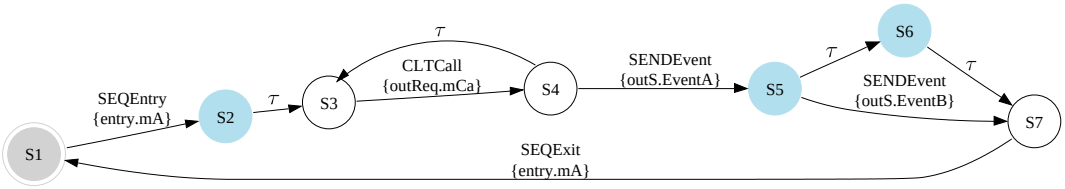


Figure 4.35 – Extrait de l'implantation de `CompA` et comportement de niveau composant primitif associé.

La représentation graphique des automates comportementaux de ce niveau d'abstraction est basée sur les conventions suivantes :

- Les transitions sont simplement étiquetées par l'entrée et la sortie d'une séquence (respectivement `SEQEntry` et `SEQExit`), par les actions d'interactions avec l'environnement du composant en fonction de la nature des interface (`CLTCall`, `WRITEData`, `CONSUMEEvent`, etc) ou par « τ ».
- Les états « remplis » correspondent à un pas d'exécution local au composant entre deux actions observables. À l'inverse, un état « vide » matérialise un ensemble d'actions observables sans entrelacement de pas d'exécution local. L'état « cerclé » correspond à l'état initial et final.

Sur cette figure, les pas d'exécution locaux donnés pour les états `S2`, `S5` et `S6` correspondent respectivement aux blocs de code `A`, `B` et `C` de l'implantation de `CompA` (l'appel à la méthode privée `PRV_privateM` est masqué, ou encore « fusionné » au comportement de la séquence).

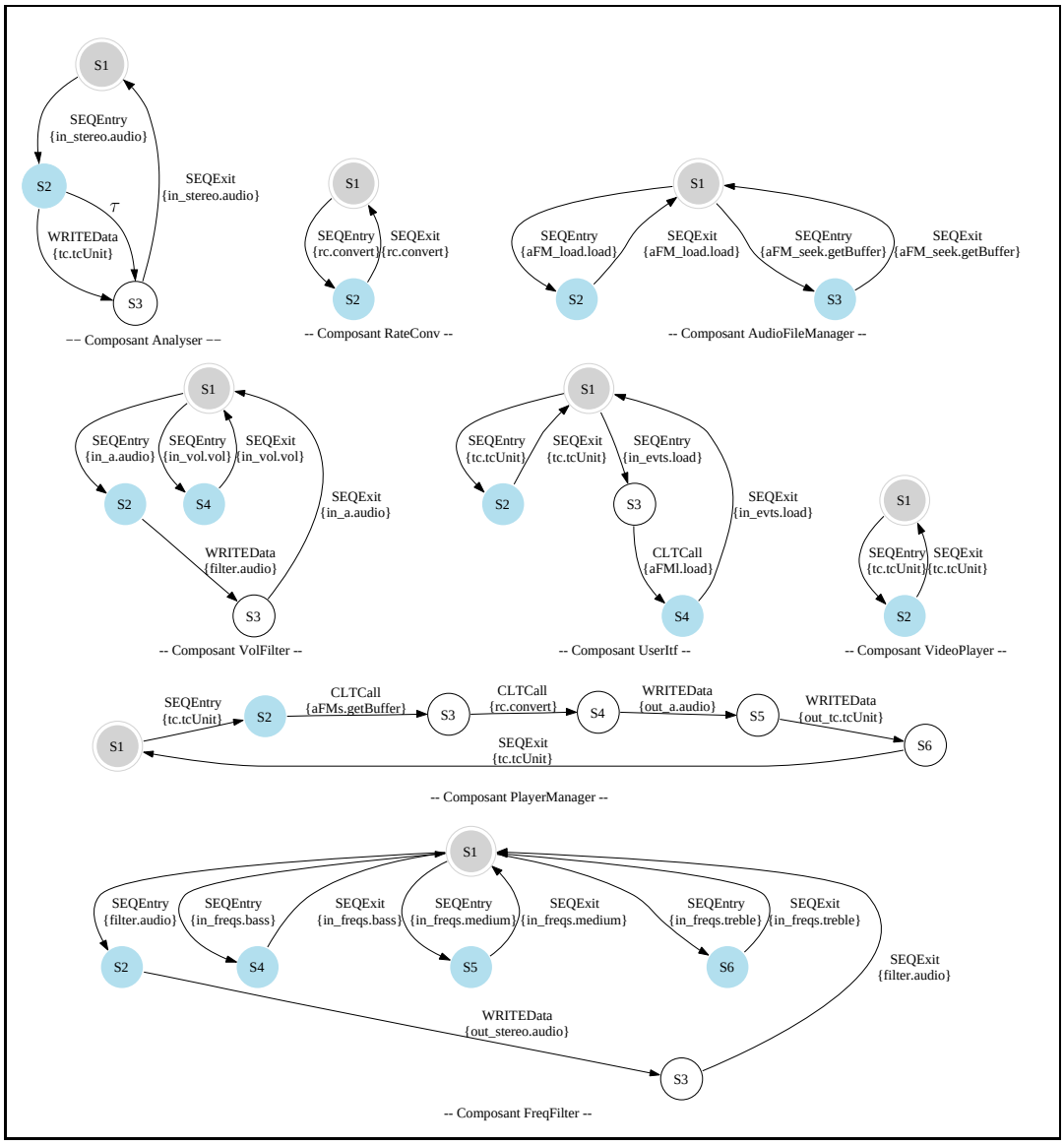


Figure 4.36 – Comportements des primitifs de l'application présentée figure 4.34.

4.5.3.2 Automates de comportement de l'exemple

Les comportements extraits des implantations des primitifs de l'architecture présentés figure 4.34 sont schématisés figure 4.36.

À chaque automate est donc associé l'ensemble des points d'entrée de séquence que le composant primitif implante.
À ce stade, nous retrouvons certaines caractéristiques pour les implantations que nous avons énoncées section 4.4.1.1, par exemple :

- Les composants `RateConv` et `AudioFileManager` sont des *`composants autistes`* (les séquences qu'ils implantent n'interagissent en aucune façon avec l'environnement).
- Les composants `VolFilter` et `AudioFileManager` sont des composants multi-implantés, le composant `PlayerManager` est mono-implanté.

4.5.3.3 Méta-modélisation

La figure 4.37 présente l'extrait de méta-modèle défini pour représenter les automates de comportement attachés aux composants. Il s'agit d'une simple machine à états dont les transitions sont étiquetées tel que nous l'avons explicité ci-dessus. Les références vers les méta-classes qui correspondent aux étiquettes sont celles définies dans le paquetage `implementation` de la figure 4.31.

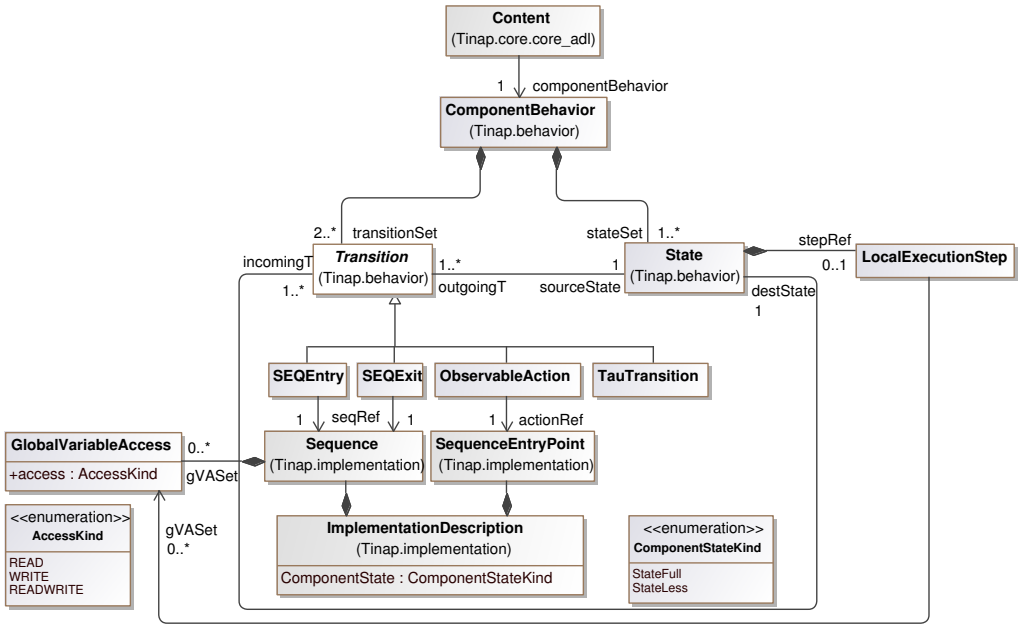


Figure 4.37 – Extrait du méta-modèle relatif à la vue comportement de niveau composant.

Ce méta-modèle permet également de représenter les accès potentiels d'un pas d'exécution local à des variables globales partagées par les différentes séquences implantées par le composant et donc de spécifier si l'implantation est « avec état » ou « sans état » (nous détaillons cet aspect page 82).
Les informations ainsi modélisées sont « ré-utilisables » : elles sont attachées aux composants et peuvent être exploitées par différentes applications.

4.5.4 Comportement de niveau architecture

Le comportement de niveau architecture consiste à donner une vue globale du comportement de l'application en exploitant l'ensemble des informations spécifiées par le concepteur (vue architecture et dynamique) et les automates de comportement de niveau composant.

Les descripteurs d'activité définissent les composants actifs de l'application, il s'agit donc de composants dont le code sera exécuté dans le contexte d'une tâche du système d'exploitation. Cependant, les chemins d'exécution susceptibles d'être parcourus par une tâche attachée à un composant actif ne se limitent pas au code encapsulé par ce composant, le fil d'exécution est en effet susceptible de traverser d'autres composants de l'architecture. Pour cela, nous définissons

la notion de ***séquence d'activité globale**** : il s'agit de construire un automate de comportement d'une séquence complète dont l'ensemble des actions observables s'exécutent dans le contexte d'une même activité. Une séquence globale peut donc être transverse à l'architecture de l'application.

Cette transversalité entre architecture et séquence d'activité globale est un point essentiel de TINAP. Le modèle de composant permet d'assembler de manière arbitraire les composants actifs et passifs pour concevoir la sémantique globale de l'application. Les entités de première classe manipulées par le concepteur sont donc les blocs de code fonctionnels implantés par les composants et non exclusivement les activités concurrentes de l'application.

Ainsi, à chaque point d'entrée de séquence d'une interface active d'un composant (actif) sera associée une *séquence d'activité globale*. Une telle séquence se construit donc automatiquement à partir des interfaces d'activation d'un composant actif, puis par composition séquentielle des comportements associés aux interfaces passives des composants liés de bout-en-bout aux interfaces de sorties de ce composant actif. Le « rayonnement » d'une telle séquence d'activité globale s'arrête lorsqu'une interface destination est une interface d'activation ou une interface d'entrée de flot interne. En effet, dans ce dernier cas, le fil d'exécution se contente de déposer une donnée ou de lever un événement qui sera consommé dans le contexte d'une autre activité (comme nous l'avons détaillé figure 4.29, page 63).

4.5.4.1 Illustration

Une *séquence d'activité globale* se spécifie par un automate à entrée/sortie. Toutes les interactions d'un composant avec son environnement au sein d'une telle séquence se font par simples appels de méthode dans le contexte d'une unique tâche (donc purement synchrones). Les transitions de l'automate représentent alors les actions observables sur les interfaces passives d'entrée et de sortie des composants.

Pour caractériser ces actions, nous avons repris un simple formalisme proposé dans SAFARCHIE [Barais, 2005], lui même inspiré de FSP (*Finite State Processes*, [Magee, 1999]). Quatre actions sont proposées :

- les actions émises notées `!action`.
- les retours de réponses à une action émise notés `?action$`.
- les actions reçues notées `?action`.
- les retours d'une action notés `!action$`.

Dans TINAP, au sein d'une séquence d'activité globale, ces actions se font par simples appels de méthodes mais peuvent représenter une interaction entre interfaces de service aussi bien qu'une interaction entre interfaces de flot. Ainsi, « `action` » est définie par le nom du composant, le nom de l'interface et le nom du point d'interaction concerné par l'interaction.

Par exemple, l'entrée d'un fil d'exécution sur une interface d'entrée passive se traduit par une action de réception (`?action_entrée`) suivie d'un retour de l'action (`!action_entrée$`). Réciproquement, la sortie d'un fil d'exécution sur une interface de sortie se traduit par une action émise (`!action_sortie`) suivie du retour de la réponse de cette action (`?action_sortie$`). Le suffixe `$` permet de distinguer le retour d'une action d'une émission d'action.

Les transitions d'indéterminisme et d'itération présentées section 4.5.3.1 (page 75) sont également spécifiées pour construire ces automates d'entrée/sortie. Une séquence d'activité globale ainsi définie permet donc de représenter complètement le rayonnement d'un fil d'exécution susceptible de traverser les différents composants de l'architecture.

Exemple de séquence globale. L'automate à entrée/sortie correspondant à l'unique séquence d'activité globale du composant `PlayerManager` est donné figure 4.38 [40].

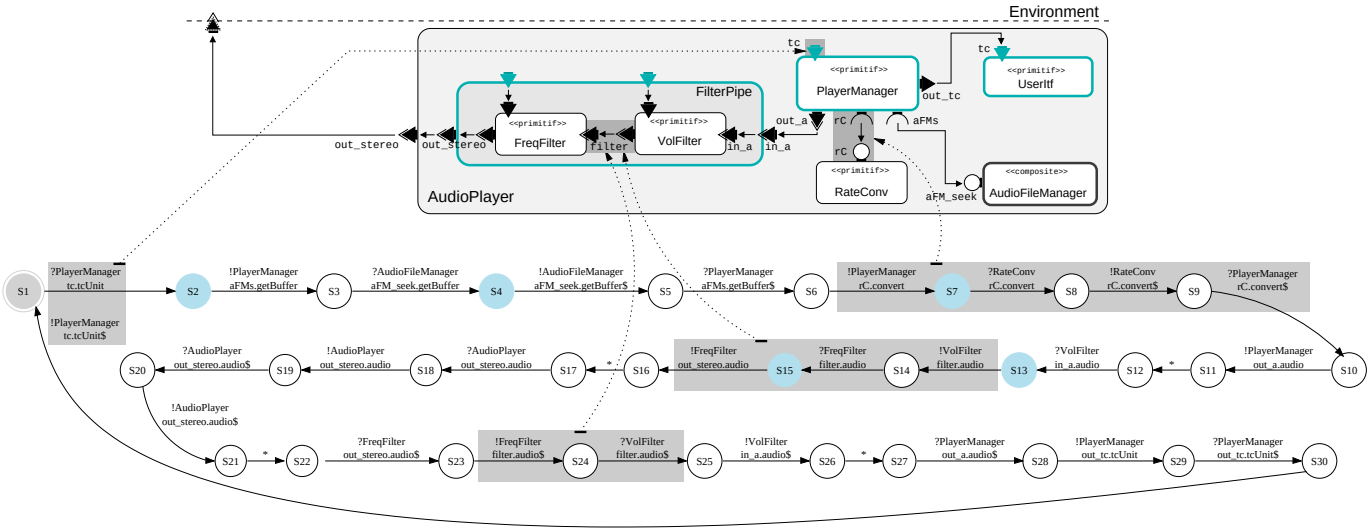


Figure 4.38 – Automate de comportement de l'unique séquence d'activité globale du composant actif `PlayerManager`.

Cet automate se construit par composition séquentielle des automates des composants `PlayerManager`, `AudioFileManager`, `RateConv`, `VolFilter` et `FreqFilter` schématisés figure 4.36 (après une étape de transformation des actions observables de niveau composant primitif vers les actions d'entrée/sortie de niveau architecture). En effet, l'architecture spécifie que tous ces composants sont liés par l'intermédiaire d'interfaces passives. L'automate résultant est totalement séquentiel puisqu'aucun indéterminisme ni aucune itération sur les actions observables n'est caractérisée sur ces automates de niveau composant primitif.

Encapsulation hiérarchique. Il est important de noter qu'à ce niveau d'abstraction, le comportement se spécifie à partir d'une « architecture plate », c'est-à-dire sans encapsulation hiérarchique : il n'est pas possible d'abstraire ce comportement dans une approche totalement « boîte noire » au sein de TINAP. En effet, un composite peut définir des sous-composants et des liaisons auxquels n'importe quels types de descripteurs de contrôle peuvent être attachés, et la connaissance des aspects dynamiques de ces sous-composants (par exemple s'il s'agit de sous-composants actifs ou protégés) ne peut s'abstraire au niveau du composite. Ce point rejoint les explications formulées à ce sujet page 63.

Processus global. Ce processus d'extraction des séquences d'activité globales peut alors se répéter pour tous les composants actifs de l'application. L'information alors obtenue est schématisée par la figure 4.39[41].

Sur cette figure sont représentées les cinq séquences d'activité globales qui correspondent aux cinq composants actifs définis au sein de la vue dynamique de l'application. Ces séquences

[40] L'automate présenté sur cette figure a été simplifié : les transitions associées aux interfaces de délégation `in_a` et `out_stereo` de `FilterPipe` ont été supprimées pour faciliter la lisibilité (une séquence de transitions supprimées est

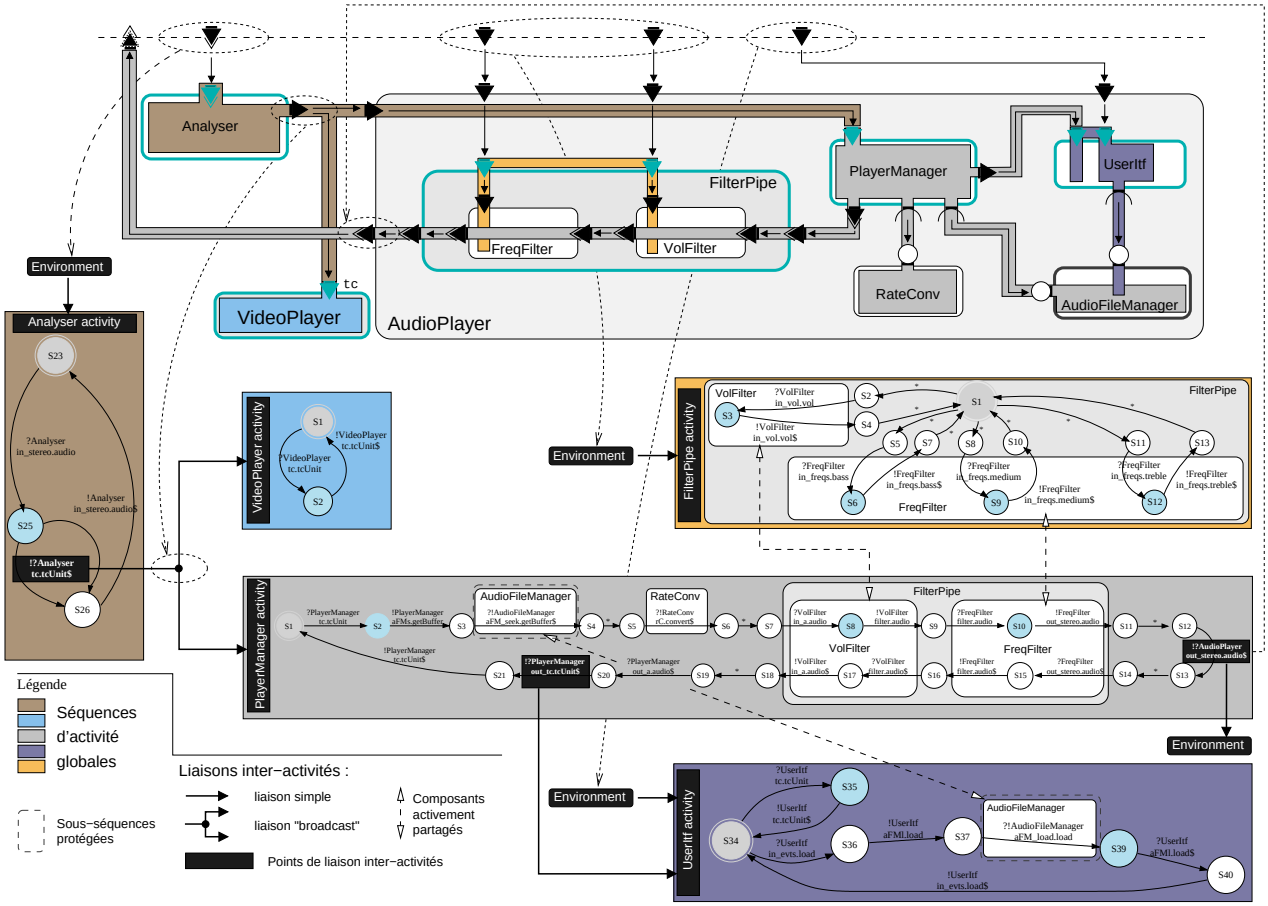


Figure 4.39 – Séquences d'activité globales de l'architecture.

seront exécutées dans le contexte d'une tâche spécifique du système d'exploitation. Il faut noter qu'à ce niveau de description, il n'est pas précisé s'il s'agit de séquence globale mono-active ou multi-active [42].

Les activités ainsi représentées interagissent de différentes manières :

- Avec l'environnement : une occurrence en provenance de celui-ci peut activer une séquence (comme c'est le cas pour la séquence de l'`Analyser`) ou écrire une donnée ou lever un événement sur une interface d'interaction interne de l'application, et réciproquement entre l'application et l'environnement.
- Entre activités de l'application : dans ce cas, on parle de liens de causalité entre activités (liens d'activations ou entre interfaces d'interactions internes).
  Lorsqu'un ensemble d'activités sont ainsi liées causalement, on parle de ***`transaction`***.

Nous avons précisé que la sémantique d'une interface multi-liée de sortie de flot TINAP était celle d'un *broadcast*. C'est par exemple le cas de l'interface de sortie du composant `Analyser`.

alors notée « * »).

[41] Comme sur la figure précédente, les automates des séquences globales de `PlayerManager` et de `FilterPipe` ont été réduits pour plus de lisibilité : certains états ont été supprimés (transitions sans étiquette), d'autres concaténées (une transition `?action` suivie d'une transition `!action$` sur une interface d'entrée concaténée en `?!action$` ou une transition `!action` suivie de `?action$` sur une interface de sortie concaténée en `!?action$`.

[42] En effet, dans ce dernier cas, différentes tâches de l'OS sont susceptibles d'exécuter une même séquence de manière concurrente.

Ainsi, une telle interface de sortie duplique les occurrences sortantes vers toutes les interfaces d'entrée consommatrices (les interfaces d'entrée du `PlayerManager` et du `VideoPlayer` dans le contexte de l'exemple).
Pour simplifier le schéma de la figure 4.39, notons que la séquence de l'`Analyser` ne présente pas l'automate de cette interaction. Cependant, il faut bien comprendre que ce *broadcast* se fait dans le contexte du fil d'exécution de l'appelant par du code mis en œuvre au sein de l'infrastructure d'exécution (présentée chapitre 5). Concrètement, cette interaction est implantée par un *push* des occurrences de sortie vers les interfaces d'entrée.

Ré-entrance. Les fils d'exécution des composants actifs sont donc susceptibles de « parcourir » les composants de manière transversale à l'architecture, nous parlerons de ***composants activement partagés***[*43]. Par définition, un composant mono-actif n'est pas activement partagé (l'infrastructure d'exécution le garantit), alors qu'un composant multi-actif l'est. Cependant, tous les autres composants (c'est-à-dire les composants passifs) de l'architecture peuvent potentiellement l'être. À partir de l'ensemble des informations obtenues à ce stade du processus, nous sommes en mesure de détecter les composants ainsi partagés, si celui-ci définit, par exemple :

- une séquence locale d'une interface d'entrée passive invoquée au sein de plusieurs séquences d'activité globales distinctes,
- une séquence locale d'une interface d'entrée passive invoquée au sein d'une séquence d'activité globale d'un composant multi-actif.
- plusieurs séquences locales d'interfaces d'entrée passives invoquées au sein de séquences d'activités globales distinctes.

Par exemple, les primitifs `FreqFilter`, `VolFilter` et `AudioFileManager` de la figure 4.39 sont activement partagés, leurs implantations respectives peuvent être appelées dans le contexte de fils d'exécution distincts.

Un composant activement partagé ne pose aucun problème de ré-entrance, si :

- le composant est « sans état ». En effet, dans ce cas, l'implantation ne repose sur aucune variable globale potentiellement accédée en parallèle.
- le composant est « avec état » mais qu'aucune séquence d'activité globale n'accède aux mêmes variables globales. En d'autres termes, cette situation se produit lorsqu'aucune séquence locale au primitif n'est entrelacée autour des mêmes variables globales.
  La figure 4.40 schématise un composant activement partagé dont deux séquences locales potentiellement appelées par des activités distinctes sont entrelacées (a) ou non-entrelacées (b) autour de variables globales accédées communément[44].

Dans le cas de notre exemple, le composant `AudioFileManager` est un composant protégé (sur la figure 4.39, cette ressource partagée protégée est schématisée autour des sous-séquences concernées). Les accès concurrents seront donc sérialisés par l'infrastructure d'exécution, garantissant ainsi l'intégrité de l'état interne du composant. Cependant, les primitifs `VolFilter` et `FreqFilter` ne le sont pas. De plus, le fil d'exécution issu de `FilterPipe` accède en écriture à une variable globale dont la valeur est lue au sein du fil d'exécution issu de `PlayerManager`. Une telle situation est détectable automatiquement par ce processus d'analyse de niveau architecture et peut lever un *warning* dans l'outil de conception[45].

[43] « Activement » partagé, pour faire la différence avec un « composant partagé » FRACTAL.
[44] Les accès aux variables partagées sont renseignés pour tous les états par l'intermédiaire de la méta-classe `LocalExecutionStep` représentée figure 4.31.
[45] En réalité, dans le cas de notre exemple, cette situation est normale d'un point de vue fonctionnel.

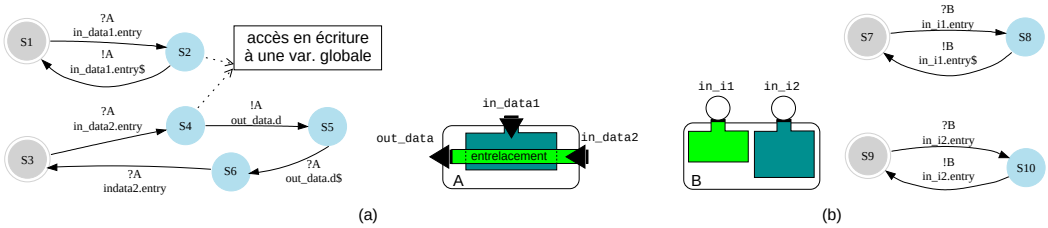


Figure 4.40 – Exemples de composants passifs activement partagés.

4.5.4.2 Méta-modélisation

Le méta-modèle correspondant à la vue comportement de niveau architecture consiste en une sérialisation de l'ensemble des informations schématisées sur la figure 4.39, mais d'autres méta-classes ont été définies pour permettre de structurer ces informations sous une forme plus condensée (le méta-modèle est donné en annexe A.1). L'important est de considérer les informations ainsi rendues disponibles aux outils de modélisation (et d'analyse).

Un exemple de représentation graphique qui pourrait être construit à partir des informations structurées par celui-ci est donné figure 4.41. Il schématise les six transactions de l'application, une issue du composant `Analyser`, quatre issues du composant `FilterPipe` et une issue du composant `UserItf`[46].

Ce méta-modèle permet donc de représenter le comportement de l'application sous forme d'un ensemble de ***`transactions`***. Une transaction est composée d'un ensemble d'activités liées causalement et dont l'activation initiale est issue de l'environnement (comme c'est le cas de notre exemple pour les activités des composants `Analyser`, `VideoPlayer`, `PlayerManager` et `UserItf`) ou d'une activation périodique.
À son tour, une activité est composée d'un ensemble de séquences d'activités globales représentées par un ensemble de « **pas** ». Nous faisons la distinction entre un ***`pas d'activation`***, un ***`pas d'exécution séquentiel`*** et un ***`pas de liaison entre activités`*** :

- Un pas d'activation correspond à un état d'activation d'une séquence d'activité globale. Il s'agit alors d'une activation périodique ou sporadique.
- Un pas d'exécution séquentiel concrétise sous forme atomique un ensemble d'actions s'exécutant dans le contexte d'une même activité mais susceptibles de traverser différents composants de l'architecture. Un tel pas est susceptible de s'exécuter de manière protégée (`ProtectedExecutionStep` comme schématisé sur la figure).
- Un pas de liaison entre activités concrétise les actions qui permettent aux activités d'interagir entre elles.

Ces pas sont alors reliés par des transitions qui représentent leurs séquencements au sein de la séquence d'activité globale.
Les liaisons entre activités concrétisent un lien d'activation ou un lien d'interaction interne entre deux composants ou entre l'environnement et l'application.
Les pas d'exécution séquentiels sont simplement représentés par les automates à entrée/sortie tel que nous les avons présentés dans la section précédente (mais non illustrés sur la représentation de la figure 4.41), de telle sorte à assurer une traçabilité entre cette représentation du comportement de l'application et la vue structurelle spécifiée par le concepteur.

Nous pouvons insister sur le fait que la vue (structurelle) dynamique (représentée figure 4.38, page 79) et la vue représentée figure 4.41 sont bijectives. Elles représentent l'ensemble de l'application selon deux points de vue différents, mais complémentaires.

[46] On peut noter qu'une même activité est susceptible de prendre part à différentes transactions, comme c'est la cas pour l'activité des composants `UserItf` ou `FilterPipe`.

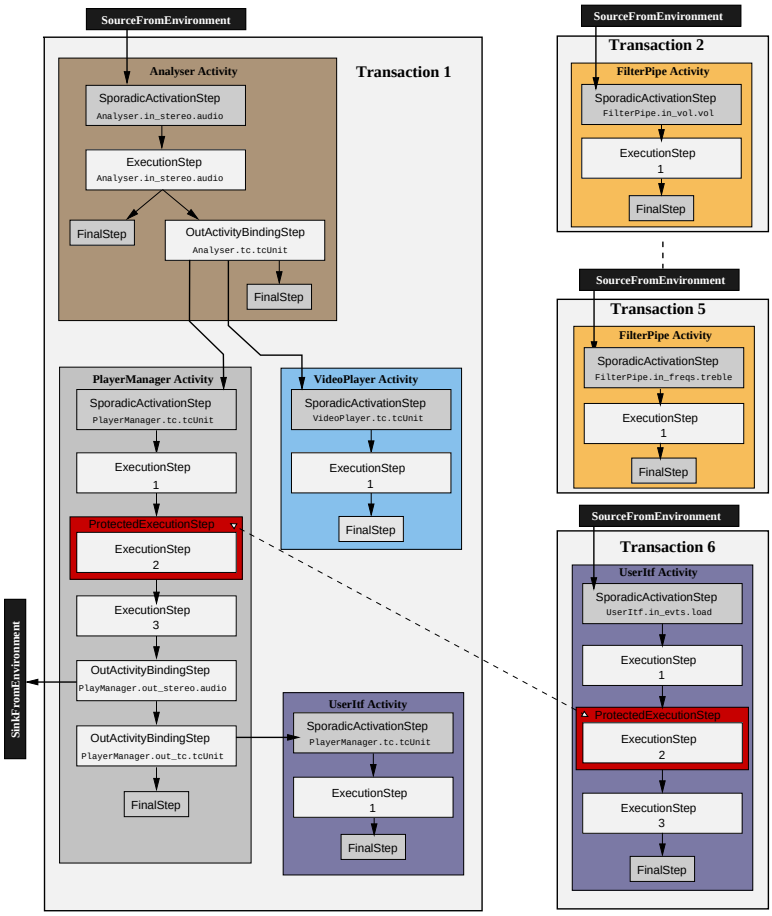


Figure 4.41 – Une représentation graphique possible de la vue comportement de niveau architecture (à partir des entités définies au sein du méta-modèle).

Annotations sur les vues comportementales. Des annotations ont été ajoutées sur les différents méta-modèles des vues comportementales[47]. Elles permettent simplement de renseigner les temps d'exécution au pire cas (WCET, *Worst Case Execution Time*) de blocs d'exécution, et donc en fonction de la plate-forme matérielle sous-jacente.
Ces annotations permettent de caractériser le WCET à différentes granularités en fonction des méta-classes auxquelles elles s'attachent.

Pour le comportement de niveau composant primitif (cf. figure 4.37), il s'agit alors d'attacher ces annotations à la méta-classe `LocalExecutionStep`, qui caractérise précisément un bloc de code entre deux actions observables du composant.

Pour le comportement de niveau architecture, ces annotations s'attachent aux entités suivantes :

- à un pas d'exécution global (nommé `SequentialExecutionStep` sur la figure 4.41). Dans ce cas, le WCET correspond à la somme de tous les WCET des pas d'exécution locaux qui composent ce pas d'exécution séquenciel global.
- à un pas d'exécution protégé (`ProtectedExecutionStep`). Le WCET attaché à cette entité permet notamment de caractériser le temps d'exécution maximal passé dans une ressource protégée.

[47] Il s'agit de simples annotations – telles que nous les avons définies page 49 – qui renseignent les temps d'exécution en fonction des paramètres de la plate-forme sous-jacente.

Le WCET est une grandeur qui permet notamment de mettre en œuvre un test d'ordonnançabilité (évoqué section suivante) permettant de statuer sur l'aptitude de l'architecture matérielle à assurer les contraintes temporelles de l'application. Cependant, dans notre cas, il s'agit simplement de proposer un mécanisme pour caractériser cette grandeur à partir des abstractions définies par TINAP et d'être en mesure d'exploiter ces informations dans une démarche de « réutilisation de composants sur étagère » – l'exploitation concrète des techniques existantes [Puaut, 2005] au sein de notre démarche sort du contexte de cette thèse.

4.5.4.3 Perspectives d'exploitation

L'extraction du « comportement de niveau architecture » permet de construire automatiquement – à partir d'une spécification d'une vue dynamique d'une application TINAP– une « vue bas-niveau » du comportement de cette application. En effet, il s'agit de représenter les séquences de blocs de code fonctionnels (notamment sous forme agglomérée) s'exécutant dans le contexte d'activités concurrentes.
L'objectif de notre démarche est avant tout d'assurer un pont entre l'espace de conception proposé par TINAP, qui promeut une spécification de haut niveau à base d'assemblages de composants et pour lesquels les aspects dynamiques sont directement spécifiés au niveau de l'architecture, vers une représentation plus adaptée à des outils d'analyse formelle de propriétés non fonctionnelles, c'est-à-dire basée sur un nombre restreint d'abstractions de base. Il est important de noter que ces expérimentations n'ont pas été menées concrètement dans le contexte de cette thèse au moment de l'écriture de ce document (il s'agit donc de simples perspectives mais qui néanmoins justifient en partie notre volonté de proposer une vue du comportement à ce niveau d'abstraction).

Couplé à la possibilité d'annoter la vue dynamique d'une application TINAP avec des contraintes d'ordonnancement (par priorités ou échéances, tel que nous le présentons dans la prochaine section), l'idée est d'être en mesure d'exploiter des outils d'analyse de propriétés temporelles tels que la vivacité (« *quelque chose de bien arrivera un jour* »), la sûreté (« *quelque chose de mauvais n'arrivera jamais* ») ou encore le respect d'échéances spécifiées entre deux événements observables sur une interface d'un composant. Ces analyses se basent généralement sur une description du comportement à l'aide d'automates temporisés [Bouyer, 2005] et exploitent des outils de model checking [Merz, 2001] tels que KRONOS [Bozga *et al.*, 1998] ou UPPAAL [Bengtsson *et al.*, 1995].

Notre démarche est également motivée par l'utilisation d'outils d'analyse d'ordonnançabilité, dont le but est de déterminer pour un modèle et une configuration de tâches donnée si le système est valide d'un point de vue temporel, c'est-à-dire si toutes les contraintes de temps spécifiées par le concepteur sont respectées par ce système. Il s'agit d'une analyse *hors-ligne* nécessitant notamment la connaissance des pires temps d'exécution des traitements, des modes d'activation des tâches[48] et de leur politique d'ordonnancement, des liens de causalité entre tâches au sein des transactions et des contraintes de synchronisation entre les entités. La vue comportement de niveau architecture telle que nous sommes capable de la construire automatiquement réduit donc le fossé sémantique entre l'espace de conception TINAP de haut niveau et celui des modèles d'analyse correspondant aux modèles de tâches utilisés dans la théorie de l'ordonnancement temps réel [Espinoza, 2007]. De telles analyses peuvent être conduites par des outils tels que MAST [Harbour *et al.*, 2001], TIMES [Amnell *et al.*, 2002], ou encore SYMPTA/S [Henia *et al.*, 2005] issu de l'industrie. Une perspective intéressante serait d'utiliser le canevas d'analyse spécifié dans MARTE [OMG, 2007].

4.5.5 Bilan sur les vues comportementales

Dans le contexte de cette dernière section, nous avons succintement présenté un outil intégré au cycle de conception d'une application TINAP. Il s'agit, à partir d'une analyse statique du

[48] Une caractérisation temporelle des stimuli de l'environnement est également nécessaire pour ce type d'analyse.

code source des composants, de vérifier la cohérence de certaines propriétés et ainsi d'aider le concepteur dans la démarche de spécification de son architecture logicielle. L'objectif est également d'automatiser l'extraction d'une représentation abstraite du comportement de ces composants.

Cette abstraction du comportement interne des implantations de composants peut être qualifiée de « contrat comportemental » alors attaché aux spécifications des primitifs (et réutilisable dans différents contextes applicatifs). Il s'agit de représenter les enchaînements des actions observables du composant, les blocs de code séquentiels s'exécutant entre ces actions et les potentiels accès aux variables d'état partagées au sein du composant.
Il s'agit alors de proposer des outils pour être en mesure de raisonner sur le comportement résultant de l'assemblage final des composants, et donc sur la sémantique globale de l'architecture logicielle. Cet aspect est assuré par un ensemble de transformations entre la vue architecturale dynamique de haut niveau manipulée par le concepteur vers une représentation orientée « flot d'exécution » de pas de traitements séquentiels. Elle permet (1) de fournir automatiquement au concepteur une vue transactionnelle (sur laquelle il peut notamment attacher des échéances temporelles de « bout en bout », comme nous le présentons dans la prochaine section) ou « orientée tâche » de son application, et (2) d'exploiter cette représentation bas-niveau au sein d'outils d'analyse non fonctionnelle dont le formalisme d'entrée est proche de ce niveau d'abstraction.

4.6 TINAP à différents niveaux d'abstraction

Dans les sections précédentes ont été présentées les différentes vues TINAP. Elles correspondent à ce que nous avons qualifié d'« espace de conception » TINAP (cf. section 3.1.1). Comme nous l'avons formulé dans le chapitre 3, notre volonté est également d'utiliser ces concepts à différents niveaux d'abstraction : celui de l'applicatif, de l'infrastructure d'exécution et du système d'exploitation. Sans chercher à entrer dans les détails, on peut affirmer que les concepts partagés par ces trois niveaux sont ceux spécifiés dans les paquetages `core` et `behavior` (cf. figure 4.2 page 37). Les concepts d'un niveau d'abstraction donné peuvent alors être enrichis pour prendre en compte des sémantiques plus spécifiques, comme nous l'avons présenté dans ce chapitre, notamment par des exemples de modèles d'exécution pour des « composants actifs » et de communications pour caractériser leurs interactions. À ce titre, l'ensemble des concepts présentés dans ce chapitre caractérisent l'espace de conception TINAP de niveau applicatif.

Les deux figures ci-dessous schématisent respectivement les différents niveaux d'abstraction que nous avons considérés et présentent les annotations utilisées pour caractériser ces niveaux à partir de la définition d'une interface[49].

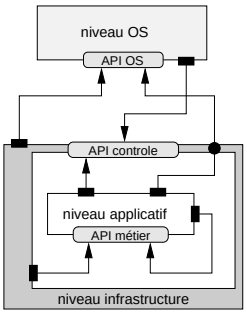


Figure 4.42 – Interactions entre les différents niveaux d'abstraction.

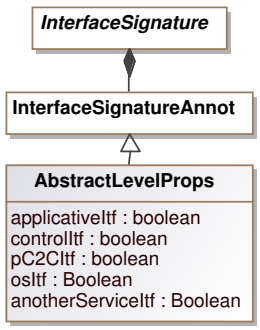


Figure 4.43 – Annotations des signatures des interfaces TINAP.

[49] Certains paramètres seront détaillés par la suite, dans la section 5.

Il est possible que l'implantation d'un composant de niveau applicatif nécessite des services de l'OS, par exemple pour protéger finement une variable partagée ou pour accéder au système de fichier. De telles dépendances doivent être explicitées au niveau architectural par les interfaces requises correspondantes. D'un point de vue méthodologique, nous considérons que l'API de référence pour les services de l'OS est celle de KORTEX (la bibliothèque de composant fournie avec THINK), mais qu'à l'étape de compilation, il est possible – par indirection ou par transformation de code – d'assurer la portabilité des composants applicatifs au-dessus de n'importe quel OS.

4.7 Spécification de contraintes d'ordonnancement

Différentes possibilités sont offertes au concepteur pour spécifier des contraintes sur l'ordonnancement des activités en concurrence au sein d'une application TINAP. Nous en décrivons succintement certaines dans cette section.
Ces informations s'attachent aux descripteurs de contrôle de la « vue dynamique » (elles s'apparentent alors à des annotations placées sur la description de l'architecture) ou aux entités spécifiées au sein de la « vue comportement de niveau architecture ».

4.7.1 Par priorité

Le mécanisme élémentaire pour contraindre l'exécution d'une application consiste à assigner de simples priorités statiques attachées aux activités TINAP. Cette possibilité est offerte au concepteur par l'intermédiaire des descripteurs d'activités que nous avons proposés[50].
Dans ce cas, les tâches du système d'exploitation attachées aux composants actifs métier seront simplement ordonnancées en fonction de ces priorités[51].

4.7.2 Par échéances temporelles

À partir des différentes vues proposées par TINAP, il est possible d'envisager de nombreuses manières pour spécifier des contraintes temporelles au niveau applicatif.
Il s'agit de contraintes d'échéance dites « relatives », c'est-à-dire qu'elles s'appliquent implicitement à une action observable : une telle échéance permet donc d'exprimer un intervalle de temps maximum à ne pas dépasser entre une action observable et le traitement qui en résulte – ou de contraintes « absolues », spécifiées en fonction de la date d'initialisation du système.

Exemples à partir de la vue dynamique :

- il est possible d'attacher une échéance à une interface de sortie d'un composant. Ainsi, le temps s'écoulant entre l'observation d'un événement de sortie sur cette interface (la production d'une donnée, un appel de méthode, etc) et le traitement qu'il en résulte doit être borné par cette échéance.
- sur une interface d'activation d'un composant actif, le concepteur peut alors spécifier une échéance à ne pas dépasser entre une demande d'activation en provenance de l'environnement et la fin du traitement associé au sein du composant actif.

Exemples à partir de la vue comportementale de niveau architecture :

[50] Il s'agit d'une annotation attachée à la méta-classe `ActivityDescriptor` (illustrée page 56).
[51] Dans le cas d'un composant multi-actif, une telle priorité s'applique alors à l'ensemble des tâches attachées au composant.

– il est possible d'attacher une échéance sur une transaction. Il s'agit dans ce cas d'une échéance sur l'ensemble des flots d'exécution qui prennent part à cette transaction (ou échéance dite de « bout en bout »).
– une échéance sur un « pas d'exécution » (`executionStep` représenté sur la figure 4.41) qui correspond alors à un bloc de code séquentiel s'exécutant dans le contexte d'une tâche donnée et susceptible de traverser différents composants de l'architecture.

4.7.3 Support des contraintes d'ordonnancement à l'exécution

Lorsque le concepteur spécifie les contraintes d'ordonnancement de son applicatif sous forme de priorités statiques, le respect à l'exécution de ses spécifications est simple : seul un ordonnanceur à priorité est nécessaire au sein de l'infrastructure d'exécution sous-jacente. L'étude de cas présentée chapitre 6 est basée sur de telles priorités.

Dans le contexte de contraintes spécifiées sous forme d'échéances temporelles[52], deux stratégies sont envisageables au sein du cycle de conception pour les faire respecter à l'exécution :

1. Une analyse formelle est réalisée « hors-ligne » à partir des spécifications, comme par exemple une analyse d'ordonnançabilité présentée ci-dessus : dans ce cas, en sortie de ce processus, si le système s'avère ordonnançable, des priorités statiques sont attribuées aux différentes activités applicatives assurant le respect de ces contraintes à l'exécution.
2. Sans chercher à valider en amont le système, il est possible de gérer dynamiquement ces échéances – donc « en-ligne » – par un ordonnancement de type EDF (*Earliest Deadline First*) supporté par l'infrastructure d'exécution. Dans ce cas, le test du respect des contraintes peut alors se faire par simulation. L'étude de cas présentée section 6.3 (page 117) est basée sur ce principe.

4.8 Conclusion du chapitre

Dans ce chapitre, nous avons présenté l'**espace de conception de TINAP** , c'est-à-dire les concepts de haut niveau qui sont proposés par notre approche et la manière dont le concepteur peut les manipuler pour construire son système.

Cet espace de conception est découpé en plusieurs vues. La « **vue structurelle** » constitue le socle de la démarche à partir de laquelle les informations des autres vues (spécifiées par le concepteur ou construites à partir d'un processus automatique) découlent. Au sein de TINAP, les entités de première classe sont des blocs de code fonctionnels statiques, les architectures ainsi spécifiées illustrent la sémantique globale du système par assemblage, réifiant les liens de dépendances fonctionnelles et de flot de contrôle et de données susceptibles d'être échangés.
La « **vue dynamique et de contrôle** » personnalise la vue structurelle en l'annotant. Ce mécanisme permet de préciser la sémantique des entités de première classe du langage. Nous nous sommes servi de ce mécanisme pour caractériser certains aspects dynamiques de la logique applicative. Ils caractérisent notamment un modèle de concurrence : il s'agit alors de raisonner en termes de tâches concurrentes, qui communiquent et se synchronisent selon différentes politiques ou en terme de composants à états partagés et protégés. Ces dispositifs extra-fonctionnels (gestion des tâches et de leurs protocoles d'activation, modèles de communication, etc) proposés au sein de cette vue deviennent en quelque sorte partie intégrante du langage TINAP. En effet, il s'agit d'aspects que le concepteur n'a pas à prendre en compte dans son code métier mais

[52] Deux manières sont offertes au concepteur pour spécifier les échéances temporelles de son application : par un « descripteur de contrôle » spécifique à partir de la vue dynamique ou, au même titre que pour les priorités, par de simples annotations au niveau de la vue comportementale de niveau architecture.

dont il devient alors utilisateur. Cette possibilité peut tout à fait s'étendre à d'autres sémantiques spécifiques.
Les implantations des composants se spécifient en C et doivent alors respecter de simples conventions, il s'agit d'assurer le lien entre ce niveau d'abstraction et celui de l'architecture. Ce que nous avons qualifié de « **vue implantation** » permet également de représenter sous forme abstraite la structure de l'implantation : construite automatiquement, elle permet de représenter comment chaque bloc de code implanté par le composant interagit avec son environnement et accède aux variables globales partagées internes à celui-ci. Il faut noter que cette dernière information illustre la nécessité d'externaliser des caractéristiques internes de l'implantation sous forme d'interfaces enrichies, pour être en mesure de vérifier le bon respect des accès aux variables partagées lors de la construction de la vue dynamique.
Enfin, les « **vues comportementales** » permettent de représenter le comportement à deux échelles, celle du primitif (entité de première classe qui encapsule un comportement) et celle de l'architecture. Elles sont aussi construites automatiquement à partir des implantations des composants. La vue de « niveau composant primitif » fournit une abstraction du comportement interne du composant sous forme d'automate et permet de caractériser les blocs de code séquentiels (avec leurs WCET) qui s'exécuteront en fonction des chemins d'exécution. Il faut noter que cette vue, couplée à celle de la « vue implantation » rassemble les informations nécessaires et suffisantes pour réutiliser un primitif dans la chaîne de conception TINAP même sous sa forme compilée. Au « niveau architecture » la vue permet de concrétiser le comportement global issu de l'assemblage des composants et des spécifications de la vue dynamique. L'information qu'elle représente n'est plus fonction des abstractions de haut niveau de l'architecture, mais demeure totalement aplatie. Il s'agit alors de raisonner à partir d'abstractions plus basiques pour être en mesure d'exploiter des techniques de validation (par exemple, analyse d'interbloquage, de performance, d'ordonnançabilité).

On peut insister sur le fait que lorsqu'il s'agit de proposer un nouveau composant TINAP à intégrer dans une bibliothèque, la démarche que nous promouvons est celle conduite par l'implantation. En ce sens, le comportement est spécifié par le concepteur en C, et il s'agit d'automatiser les informations qu'il est pertinent d'abstraire, plutôt qu'une démarche inverse (qui est généralement de mise dans les approches orientées modèles). En effet, il s'agit de s'assurer que les abstractions sur lesquels les outils valident les contraintes de fiabilité soient conformes à l'implantation réelle.

L'espace de conception TINAP que nous avons présenté dans ce chapitre constitue le « **noyau de concepts canoniques** » qui est utilisé à tous niveaux d'abstraction d'une structure TINAP (applicatif, infrastructure d'exécution, système d'exploitation[53]).

Ce « noyau multi-vues » est spécifié sous forme de méta-modèles. Ils permettent de représenter clairement la structure des concepts qui y sont définis et de spécifier les contraintes d'ordre syntaxique mais aussi sémantique qu'ils engendrent. En outre, ces méta-modèles permettent d'implanter des outils d'aide à la conception de haut niveau (notamment au sein de la plate-forme ECLIPSE, dont certains aspects sont présentés succintement en annexe B). Couplé à la prise en charge des contraintes, ils permettent d'assister sensiblement le concepteur à spécifier des assemblages TINAP cohérents et valides.

Le chapitre suivant présente comment l'infrastructure d'exécution TINAP est implantée avec ces concepts du noyau canonique.

[53] Seul le modèle de concurrence que nous avons défini n'est pas caractérisé au niveau du système d'exploitation.

Chapitre 5

Infrastructure d'exécution TINAP

Sommaire

5.1 Introduction 89
5.2 Concepts de niveau infrastructure 90
5.2.1 Membranes 90
5.2.2 Réification du lien entre le niveau applicatif et le niveau infrastructure . . 90
5.2.3 Intercepteurs 91
5.2.4 Contrôleurs 92
5.2.5 Vers un langage de niveau infrastructure 93
5.3 Implantation des membranes TINAP 93
5.3.1 Etape 1 : Projection des interfaces 93
5.3.2 Etape 2 : Insertion des intercepteurs 95
5.3.3 Etape 3 : Insertion des contrôleurs 97
5.3.4 Etape 4 : Liaison entre composants de la membrane 98
5.3.5 Etape 5 : Implantations des composants de contrôle 99
5.3.6 Autres implantations et caractéristiques de l'infrastructure 101
5.4 Méta-modélisation d'une « vue infrastructure » 103
5.5 Bilan sur le niveau infrastructure 103
5.6 Niveau système d'exploitation 104
5.7 Conclusion du chapitre 105

5.1 Introduction

Dans le chapitre 4, nous avons présenté ce que nous avons qualifié d'« espace de conception TINAP ». Il s'agit d'un langage de haut niveau car l'expressivité offerte au concepteur repose sur un ensemble de (non-)fonctionnalités qui doivent être implantées au sein de l'infrastructure d'exécution. En ce sens, une architecture TINAP n'est pas directement instanciable telle qu'elle est définie par le concepteur.

Comme nous l'avons exprimé au sein du chapitre 3, notre objectif est d'exploiter le paradigme composant à différents niveaux d'abstraction pour concevoir un système dans sa globalité. Dans ce chapitre, nous nous intéressons à la mise en œuvre des concepts TINAP au niveau de l'infrastructure d'exécution (sections 5.2 et 5.4) et des services élémentaires du système d'exploitation (section 5.6).

5.2 Concepts de niveau infrastructure

L'infrastructure d'exécution TINAP s'implante sous forme d'assemblages de composants. Cette section présente les concepts fondamentaux de ce niveau d'abstraction.

5.2.1 Membranes

L'infrastructure d'exécution est dite « distribuée ». En ce sens, il ne s'agit pas d'implanter de manière monolithique cette couche de fonctionnalités nécessaire à l'exécution des composants métier mais de la « disposer » autour de ces derniers en fonction de leurs besoins spécifiques. L'infrastructure propre à un composant donné est appelée « **membrane** »[54], issu de la terminologie FRACTAL. Elle est définie par un ensemble de « **composants de contrôle** » : les **intercepteurs** et les **contrôleurs**.

Les spécificités d'une membrane sont les suivantes :

- en considérant les différents niveaux d'abstraction présentés dans la section 3.1.3 (page 32), une membrane implante les concepts nécessaires à l'exécution d'un langage de plus « haut niveau ». C'est par exemple le cas des membranes TINAP qui implantent les concepts du niveau applicatif. Dans ce cas, on peut parler de « **membrane spécifique à un domaine** » (ou « membrane de domaine »).
- De manière plus générale, son rôle est de mettre en œuvre une superposition de fonctionnalités arbitraires, propres à un besoin particulier (non obligatoirement prévu initialement).
- Elle peut s'apparenter à une **couche de virtualisation**, par exemple pour faire le lien entre le « fonctionnel métier » et les services élementaires implantés par le système d'exploitation sous-jacent.
- Elle est générée automatiquement et de manière transparente pour le concepteur.
- Elle est programmable et extensible[55].

Dans notre démarche, une membrane est rendue concrète à l'exécution par un composite. Ainsi, un primitif de niveau applicatif sera encapsulé au niveau infrastructure par un composite auquel seront ajoutés les composants de contrôle qui implantent la membrane.
Un composite de niveau applicatif sera préservé au niveau infrastructure mais des composants de contrôle y seront ajoutés.

5.2.2 Réification du lien entre le niveau applicatif et le niveau infrastructure

Comme nous l'avons mentionné, le niveau applicatif de TINAP repose sur un ensemble de concepts qui doivent être supportés par l'infrastructure d'exécution. À chaque composant applicatif sera donc attachée une membrane spécifique qui met en œuvre ces fonctionnalités et mécanismes de contrôle nécessaires à son exécution. On peut alors parler de projection d'un tel composant vers un « composant exécutable » au sein de son propre support d'exécution.
L'implantation d'un composant applicatif et celle de son infrastructure sont donc fortement couplées. Dans notre approche, nous nous sommes attachés à rendre concret cette relation en définissant les ***`interfaces pC2C`*** (pour *Primitive Control To Content* ou *Primitive Content To Control*). **Il s'agit d'interfaces de service** qui traduisent une relation de dépendance (et nous parlerons de ***`liaisons pC2C`***) entre une implantation métier et la membrane qui lui est attachée (dans une direction comme dans l'autre).
Dans la section 5.3.1 nous expliquons concrètement comment ces interfaces sont générées et manipulées à ce niveau d'abstraction.

[54] Dans ce document, nous employons donc souvent le terme « d'infrastructure d'exécution » au lieu du terme « membrane » pour un composant donné.
[55] Ces deux aspects sont à attribuer aux outils qui implantent le processus de génération de l'infrastructure.

5.2.3 Intercepteurs

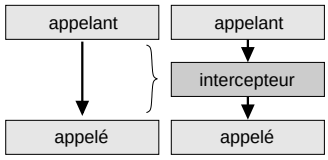


Figure 5.1 – Interposition.

L'*interposition* est une technique fondamentale en informatique. Elle s'applique à une multitude de mécanismes [Fassino, 2001], par exemple pour mettre en œuvre les talons (*stub* et *skeleton*) dans les systèmes répartis pour abstraire les communications, les liens de réification dans les systèmes réflexifs, les sous-contrats pour la spécialisation d'un comportement, les conteneurs des modèles de composants ou encore l'injection de code *advice* dans les approches aspects. Typiquement, elle consiste à ajouter une indirection sous forme d'une couche logicielle – un intercepteur – supplémentaire entre un bloc de code appelant et un bloc de code appelé (figure 5.1).

L'interposition est à la base de la mise en œuvre des membranes FRACTAL. Elle permet de superposer à la couche métier une couche de contrôle de complexité arbitraire en interceptant les communications entrantes et sortantes respectivement en destination et en provenance des composants métier.
Les intercepteurs peuvent être générés automatiquement, et ce processus est notamment rendu possible grâce à la notion d'interface, dont la spécification est indépendante de leur implantation. Ainsi, l'injection d'intercepteur peut se faire de manière transparente relativement aux implantations du code appelant et du code appelé (c'est-à-dire sans transformation de code métier).

Les intercepteurs sont des types particuliers de composants de contrôle. Nous formulons ci-dessous quelques explications et définitions pour caractériser plus précisément leur sémantique dans le cas général :

- Nous dirons qu'un composant d'interception présente une ***interface interne**** liée à l'interface qu'il intercepte (par exemple, les interfaces `pC2C` que nous avons évoquées) et une ***interface externe**** qu'il présente à l'environnement.
- Nous faisons la distinction entre les ***intercepteurs d'entrée**** et les ***intercepteurs de sortie****, respectivement pour une interception d'un fil d'exécution entrant vers le contenu intercepté (à savoir l'implantation d'un primitif ou les sous-composants d'un composite) ou pour une interception du fil d'exécution sortant du contenu. Cette notion s'apparente à celle que l'on peut trouver dans la terminologie AOP[56] pour « encadrer » une méthode métier par exemple (*before* et *after*) mais ici appliquée à un groupe de méthodes (rassemblées au sein d'une même interface).
- Dans le cas général, nous considérons donc qu'un composant d'interception s'applique à une unique interface interceptée.
- Un intercepteur, en plus des interfaces interne et externe auxquelles il est attaché est susceptible de requérir (et même parfois de fournir) des interfaces vers d'autres composants (généralement des contrôleurs, présentés ci-dessous), il s'agit alors d'un **déroutement** du flot de contrôle. Ce déroutement illustre alors un lien de dépendance entre le métier et les composants de son infrastructure). Nous faisons donc la distinction entre ***intercepteur avec déroutement*** et ***intercepteur sans déroutement***. Dans le premier cas, nous qualifions d'***interface de déroutement**** une interface qui concrétise le déroutement du fil d'exécution.
- Nous qualifions d'***intercepteur de type symétrique**** un composant dont l'interface externe est du même type que celle qu'il intercepte (donc du même type que son interface interne). Au contraire, nous parlerons d'***intercepteur de type asymétrique****

[56] *Aspect Oriented Programming*.

lorsque son interface externe est de type différent. Ce dernier mécanisme est par exemple couramment employé pour assurer la « glue » entre composants ne présentant pas les mêmes interfaces (aspect retrouvé dans les approches « basées connecteur »).

- L'interposition est un mécanisme qui par définition permet de « superposer des *non fonctionnalités* à celles qu'il intercepte ». Généralement, un intercepteur n'est pas attaché à une implantation donnée, il permet seulement de dérouter les appels vers des composants externes, appelés contrôleurs. Cependant, dans notre approche, certains intercepteurs implantent des fonctionnalités qui leur sont propres. Nous faisons donc la distinction entre ***intercepteur sans fonctionnalité*** et ***intercepteur avec fonctionnalité***. Dans ce dernier cas, la granularité d'un intercepteur peut être arbitraire, il est ainsi possible d'implanter ses fonctionnalités au sein d'un composite. Les fonctionnalités implantées par un intercepteur sont liées à la mise en œuvre de liaisons spécifiques établies entre composants de niveau applicatif.
- Enfin, nous parlerons d'***intercepteur mono-directionnel*** et d'***intercepteur bi-directionnel*** en fonction de la polarité de ses interfaces interne et externe. Un intercepteur mono-directionnel présente une interface interne et une interface externe de polarité opposée : une interface externe fournie et une interface interne requise pour un intercepteur d'entrée (et la réciproque pour un intercepteur de sortie). Un intercepteur bi-directionnel spécifie toujours une interface externe et interne fournie.
- Il est possible que l'interface interne d'un intercepteur d'entrée (et réciproquement l'interface externe d'un intercepteur de sortie) ne soit pas définie. Ce cas particulier utilise le déroutement vers un composant de contrôle externe (un contrôleur) et est essentiellement utilisée lorsque l'interface interceptée est appelée par un fil d'exécution différent de celui de l'appelant (pour mettre en œuvre les interactions entre deux composants actifs par exemple).
- Les intercepteurs sont utilisés aussi bien pour intercepter des communications de (ou vers) un primitif que d'un composite de niveau applicatif.

5.2.4 Contrôleurs

Un contrôleur est aussi un composant de contrôle. À la différence d'un intercepteur dont l'objectif principal est d'intercepter les invocations à destination ou en provenance du contenu puis de les dérouter, le contrôleur implante des « fonctionnalités de contrôle » qui se superposent à celles implantées par le contenu auquel il est attaché.

Nous pouvons préciser certains aspects propres à ce type de composants de contrôle :

- Un contrôleur encapsule des fonctionnalités de granularité arbitraire, éventuellement au sein d'un composite.
- Il définit des ***interfaces de contrôle externes*** et des ***interfaces de contrôle internes***. Dans le premier cas, il s'agit d'interfaces particulières rendues visibles à l'environnement que celui-ci est susceptible d'invoquer (une interface définie pour introspecter le composant auquel est attaché le contrôleur par exemple) ; il s'agit toujours d'une interface d'entrée. Dans le deuxième cas (interfaces de contrôle internes), elles sont utilisées pour assembler les composants de contrôle appartenant à une même membrane.
- Dans le cas courant, un contrôleur est propre à la membrane du contenu ainsi encapsulé. En effet, il s'agit généralement d'un contrôleur « avec état », dont l'état est spécifique au contenu contrôlé.
  Cependant, un contrôleur peut être aussi partagé par différentes membranes.
- Un contrôleur peut être un simple composant de virtualisation, par exemple pour virtualiser une API du système d'exploitation sous-jacent auprès de composants de niveau applicatif.

5.2.5 Vers un langage de niveau infrastructure

Les deux sections précédentes illustrent notre volonté de caractériser – dans le cas général – les spécificités des composants d'interception et de contrôle dans une démarche de « componentisation des (non-)fonctionnalités de l'infrastructure d'exécution ». L'objectif de notre démarche est donc de manipuler ces « personnalités » d'interfaces et de composants de contrôle en tant qu'entités de première classe. Il s'agit alors de proposer un langage dédié à la description des architectures des membranes, sous forme d'une spécialisation des concepts du « noyau TINAP ». Comme nous le présentons par la suite (section 5.4) ces spécificités sont également rendues concrètes au sein d'un méta-modèle, et permettent ainsi d'ajouter de la sémantique aux entités de contrôle modélisées. La prochaine section s'attache à décrire l'implantation des membranes TINAP.

5.3 Implantation des membranes TINAP

Dans cette section, nous présentons les principales étapes du processus de génération d'une infrastructure TINAP. Pour chaque composant, il s'agit de générer automatiquement les membranes qui mettent en œuvre les aspects non fonctionnels du profil de composant. Elles se caractérisent principalement de la manière suivante :

1. De simples projections d'interfaces applicatives vers des interfaces de niveau infrastructure sont effectuées,
2. les intercepteurs,
3. puis les contrôleurs sont ajoutés au sein de la membrane,
4. les liaisons entre les différents composants de la membrane sont établies,
5. enfin, les implantations de ces composants de contrôle sont générées.

Toutes ces étapes sont fonction des spécifications de l'application définie par le concepteur, c'est-à-dire de la vue structurelle (composants, interfaces, liaisons, etc) et de la vue dynamique (descripteurs de contrôle).

Dans les sous-sections qui suivent, pour chacune de ces étapes, nous présentons de manière concise les points principaux que le processus de génération doit prendre en compte dans le cas général, sans toutefois être exhaustif.
Nous nous basons ensuite sur le cas particulier du composant primitif `CompA` (donné figure 4.28, page 62) pour illustrer plus précisément certains de ces propos (nous considérons qu'il s'agit d'un composant mono-actif, dont l'unique interface fournie est spécifiée comme interface d'activation).

5.3.1 Etape 1 : Projection des interfaces

5.3.1.1 Cas général

La première étape de génération de l'infrastructure d'exécution d'un composant métier consiste en une projection des interfaces de flot TINAP vers des interfaces de service. En effet, ces modèles de communication sont gérés par l'infrastructure et implantés à ce niveau d'abstraction par de simples appels de méthodes. Ainsi, pour un composant primitif :

– Une interface de sortie de flot se projette à ce niveau d'abstraction en une simple interface de service requise,

– Une interface implantée de flot en une interface fournie.
– Une interface d'interaction interne est projetée en une interface requise. En effet, une telle interface est une interface d'entrée au niveau applicatif, mais la mémorisation des occurrences est assurée par l'infrastructure (au sein d'un intercepteur comme nous l'expliquons ci-dessous) et leur consommation est explicitement demandée par l'implantation du code métier (il s'agit donc d'un service requis à ce niveau d'abstraction).

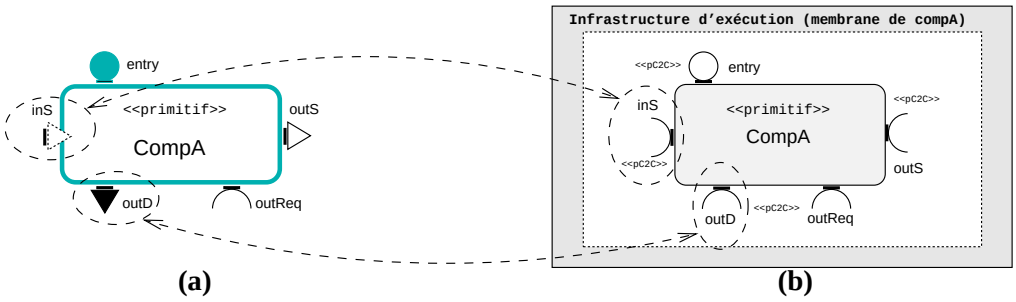
Les interfaces ainsi projetées sont les interfaces dites `pC2C` que nous avons caractérisées. Leurs signatures (de niveau infrastructure) sont générées en fonction de celles définies par le concepteur.
Pour les interfaces de flot, les signatures générées sont basées sur les mêmes conventions que celles explicitées page 68.

Les interfaces d'activation définies au niveau applicatif sont aussi des interfaces `pC2C` car elles fournissent des points d'entrée qui seront invoqués par l'infrastructure dans le contexte d'une activité. Deux cas particuliers de signatures sont générés pour ces interfaces : (1) pour un composant auquel est attaché un « descripteur périodique ». Dans ce cas, une signature d'interface atomique sera générée, avec une unique opération portant le nom du descripteur de contrôle[57]. (2) Pour un composant « `actif et logique` ». Dans ce cas également, une signature d'interface atomique sera générée, avec une opération dont les paramètres seront la concaténation de l'ensemble des points d'interaction des interfaces d'activations concernées. En effet, dans ce cas, un unique point d'entrée de séquence doit être fourni par l'implantation du composant.

À ce niveau d'abstraction, la vue implantation externe d'un primitif (définies page 67) permet de vérifier que le primitif implante les points d'entrée conformément à ces conventions (ou lever une exception le cas échéant en amont de l'étape de compilation).

5.3.1.2 Exemple du composant `CompA`

Par exemple, toujours en nous basant sur le composant `CompA` (donné sur la figure 4.28, page 62) : ce primitif est attaché à des interfaces de flots de données et d'événements et à un descripteur d'activité qui définit l'interface `entry` comme interface d'activation. La projection de ses interfaces et des signatures associées s'établit tel que schématisé par la figure 5.2.



Signatures de niveau applicatif :
1. Interface d'entrée d'événement **inS** ⇒
   EventInterface **inSignItf**
   {inNotifyEvt;}
2. Interface de sortie de donnée **outD** ⇒
   DataInterface **outDataItf**
   {MyStructType s;}

Signatures de niveau infrastructure :
1. Interface de service requise **inS**
   OperationInterface **inSignItf_gen**
   {inNotifyEvt ();}
2. Interface de service requise **outD**
   OperationInterface **outDataItf_gen**
   {s (MyStructType s);}

Figure 5.2 – Projection des interfaces (et signatures) de niveau applicatif (a) de `CompA` vers les interfaces de niveau infrastructure (b).

[57] En effet, les descripteurs de contrôle sont nommés (cf. figure 4.17, page 53).

5.3.2 Etape 2 : Insertion des intercepteurs

5.3.2.1 Cas général

Les intercepteurs sont des types particuliers de composants. Comme nous l'avons présenté, ils peuvent être « avec fonctionnalité ». Cette propriété des intercepteurs est utilisée dans notre approche pour mettre en œuvre les liaisons TINAP de niveau applicatif (gestion des protocoles de communication, des interfaces applicatives multi-liées, etc) et prendre en charge certains aspects d'ordre non fonctionnels.

Ces liaisons de niveau applicatif sont donc implantées au niveau infrastructure par différents intercepteurs liés entre eux. On peut ainsi parler de « liaisons composites » qui concrétisent des chemins de communication arbitrairement complexes entre deux interfaces de composants (de niveau applicatif) et implantées par un ensemble de composants de liaison (les intercepteurs dont la sémantique est ici proche de celle de *stub* et de *skeleton*). Ces intercepteurs sont alors reliés par des « liaisons primitives » (c'est-à-dire sans indirection)[58]. La figure 5.3 schématise ces aspects.

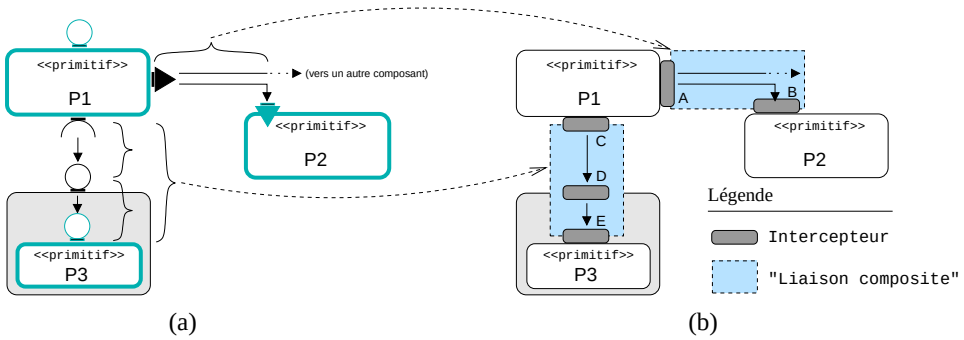


Figure 5.3 – Intercepteurs et notion de « liaisons composites » au niveau infrastructure.

La partie gauche de la figure présente un simple exemple de composants liés par leurs interfaces tel que le concepteur le spécifie au niveau applicatif. La partie droite schématise la disposition des intercepteurs telle qu'elle est prise en charge par le processus de génération de l'infrastructure d'exécution.

La génération du code de ces intercepteurs (ou « liaisons composites ») dépend des descripteurs de contrôle spécifiés par le concepteur pour les composants et les liaisons de niveau applicatif. Sans être exhaustif, nous donnons quelques précisions relativement à l'exemple schématisé ci-dessus :

- L'intercepteur d'entrée `B` de `P2` implante la logique du protocole de la liaison de données issue de `P1` (boîte aux lettres, rendez-vous, etc).
- Il en est de même pour l'intercepteur d'entrée `E` de `P3` qui implante la logique du protocole de la liaison issue de `P1` (synchrone, asynchrone, etc).
- L'intercepteur de sortie `A` de `P1` prend en charge la sémantique de *broadcast* associée à l'interface multi-liée.

[58] La notion de « liaison composite » s'apparente aussi à celle de « connecteur ».

Dans la section 5.2.3, nous avons caractérisé les « intercepteurs de type asymétrique ». Ils sont utilisés dans certains cas de liaisons composites pour assurer le lien entre interfaces métier et interfaces définies spécifiquement entre intercepteurs de l'infrastructure d'exécution. Ces interfaces spécifiques permettent de faire transiter des informations d'ordre non fonctionnel nécessaires à la gestion des communications (comme par exemple une échéance temporelle dans le cas de l'implantation du modèle d'exécution ACCORD présenté section 6.3 (page 117). Par exemple, une interface spécifique peut-être utilisée entre les intercepteurs `C` et `D` et les intercepteurs `D` et `E` présentés sur la figure.

Sans entrer dans les détails, le code des intercepteurs est également généré pour prendre en charge le passage des informations (données, paramètres) soit par valeur, soit par référence en fonction des liaisons, ainsi que la copie de ces informations entre les différents composants.

5.3.2.2 Exemple du composant `CompA`

De manière plus concrète, nous nous intéressons maintenant aux intercepteurs générés pour le composant `CompA` de notre exemple. La figure 5.4 les schématise.

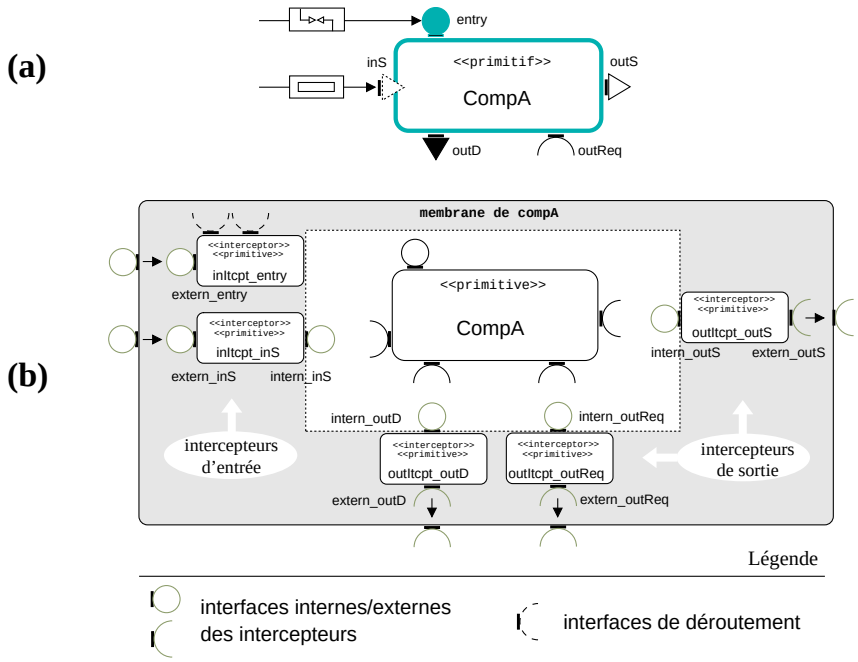


Figure 5.4 – Vue dynamique de `CompA` de niveau applicatif (a) et intercepteurs qui lui sont associés au niveau infrastructure (b).

Nous pouvons ici préciser certaines propriétés des intercepteurs données dans la section 5.2.3 :

– Dans cet exemple, tous les intercepteurs de sortie sont des *`intercepteurs mono-directionnels`*. L'intercepteur associé à l'interface métier d'entrée de flot d'interaction interne `inS` est un *`intercepteur bi-directionnel`* : les composants de l'environnement lèvent des événements par l'intermédiaire de l'interface externe de l'intercepteur, le métier les consomment par l'interface interne.

– À part l'intercepteur `inItcpt_entry`, tous les intercepteurs sont *`sans déroutement`* sur cet exemple.
– L'*`intercepteur avec déroutement`* est ici un cas particulier d'intercepteur sans interface interne. Il faut noter qu'une interface de déroutement est par définition une interface de contrôle interne (présentée page 92) puisqu'elle concrétise une dépendance fonctionnelle entre composants de contrôle.
– Dans le contexte de cet exemple, les deux intercepteurs d'entrée seront *`avec fonctionnalité`* (dont les implantations sont présentées à l'étape 5), ils implantent les protocoles de communication définis par les descripteurs de contrôle attachés aux liaisons métier (représentés graphiquement sur la figure 5.4 (a), et qui correspondent aux protocoles « rendez-vous » et « incrémenté » caractérisés figure 4.28, page 62).

5.3.3 Etape 3 : Insertion des contrôleurs

5.3.3.1 Cas général

Au même titre que les intercepteurs, les contrôleurs sont aussi des types particuliers de composants.
Les contrôleurs TINAP prennent principalement en charge :

– Les implantations des protocoles d'activation définis par les descripteurs d'activité de niveau applicatif (« mono-actif », « multi-actif », etc).
– l'implantation des composants protégés.
– la mémorisation des informations transmises par les occurrences entrantes associées aux interfaces d'activation (données, paramètres des opérations et éventuellement d'autres informations d'ordre non fonctionnel nécessaires à la couche d'infrastructure). Cette fonctionnalité est assurée par une boîte aux lettres générique permettant d'ajouter et de retirer des éléments. Il s'agit également d'un espace mémoire propre au composant actif auquel il s'attache : les données qui y sont copiées lui appartiennent et ne peuvent être modifiées par d'autres composants de l'environnement.

5.3.3.2 Exemple du composant `CompA`

Les deux contrôleurs associés à la membrane du composant métier `CompA` sont schématisés sur la figure 5.5.

Le composant `MailBox` est la boîte aux lettres permettant donc de mémoriser les informations transmises par un composant appelant de l'environnement et à destination des méthodes implantées par le composant métier. Elle fournit deux interfaces de contrôle internes, l'une pour socker des informations (`mbox_add`), l'autre pour les récupérer (avec ou sans suppression) (`mbox_get`). Dans le contexte de cet exemple, elle permet de stocker les paramètres d'entrée définis par les méthodes de l'interface active `entry` de `CompA`).

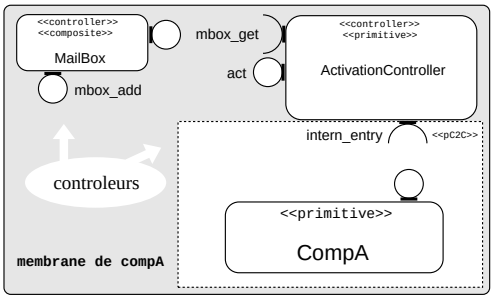


Figure 5.5 – Contrôleurs de `CompA`.

L'`ActivationController` contrôle l'exécution des méthodes implantées par le concepteur en fonction du descripteur d'activation spécifié par le composant métier, dans le cas de

cet exemple, il s'agira de la mise en œuvre du « protocole d'activation mono-actif ». Il requiert donc une interface `pC2C` lui permettant d'invoquer les méthodes implantées par le métier (`intern_entry`).
Un tel contrôleur définit des interfaces de contrôle internes : il requiert l'interface pour récupérer des informations de la boîte aux lettres et fournit une interface `act` permettant aux autres composants de contrôle de la membrane de demander explicitement une activation du composant métier contrôlé.

Sur cet exemple, la membrane de `CompA` n'exporte pas d'interface de contrôle externe à son environnement.

5.3.4 Etape 4 : Liaison entre composants de la membrane

5.3.4.1 Cas général

Le lien entre les composants de contrôle et le contenu métier est assuré par les liaisons `pC2C`. Typiquement, les interfaces du contenu sont liées aux interfaces internes des intercepteurs et des contrôleurs.

Les composants de contrôle sont liés par l'intermédiaire des interfaces de contrôle internes qui concrétisent les dépendances fonctionnelles entre contrôleurs de la membrane et entre intercepteurs et contrôleurs.

Dans le cas des membranes générées pour les composants actifs, d'autres liaisons entre composants de contrôle sont nécessaires pour implanter les aspects non fonctionnels de TINAP. Elles correspondent à un besoin de synchronisation entre contrôleur d'activation et intercepteur d'interface d'activation. Plutôt que d'énumérer tous les cas d'utilisation, nous donnons un exemple concret de ce besoin de synchronisation ci-dessous.

5.3.4.2 Exemple du composant `CompA`

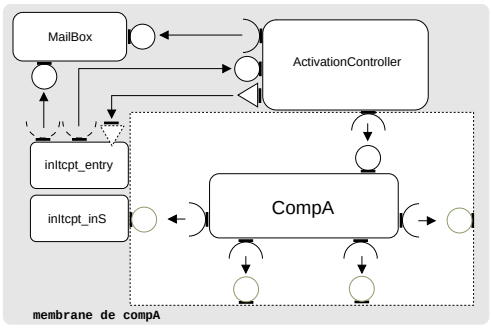


Figure 5.6 – Liaisons de la membrane.

L'ensemble des liaisons établies au sein de la membrane de `CompA` est donnée figure 5.6. Dans le cas de cet exemple, l'intercepteur `inItcpt_entry` implante le protocole attaché à la liaison à destination de l'interface `entry` de niveau applicatif, il s'agit du protocole « rendez-vous » (cf. figure 5.4a) pour une liaison client-serveur. Pour ce type de protocole, le fil d'exécution du client sera donc bloqué jusqu'au commencement effectif de l'exécution de la requête correspondante. L'interface d'entrée d'événement qui lui est attachée permet de concrétiser ce besoin de synchronisation et ces événements sont alors levés par le contrôleur d'activation.

L'intercepteur d'entrée et les contrôleurs sont donc liés par des liaisons de contrôle internes qui illustrent les dépendances entre ces composants :

– Les deux interfaces fournies par le composant `MailBox` permettent l'ajout ou le retrait des éléments de la boîte. Dans ce cas, ils sont ajoutés par l'intercepteur et retirés par le contrôleur d'activation.

– La liaison entre l'intercepteur et le contrôleur d'activation est utilisée pour notifier explicitement ce dernier qu'une requête d'activation en provenance de l'environnement a été demandée.
– La liaison d'événement entre le contrôleur d'activation et l'intercepteur permet de débloquer la tâche associée au client en attente du commencement de l'exécution de la requête.

Les liaisons `pC2C` sont également établies entre le composant métier et les composants de contrôle.

5.3.5 Etape 5 : Implantations des composants de contrôle

5.3.5.1 Cas général

La dernière étape consiste à générer les implantations des composants de contrôle. Comme nous l'avons déjà remarqué, le code de ces composants est fonction des descripteurs attachés aux composants et aux liaisons de niveau applicatif.

En plus d'implanter les aspects non fonctionnels propres aux descripteurs TINAP, certains composants de contrôle permettent également d'assurer le lien entre composants métier et services du système d'exploitation. C'est par exemple le cas des contrôleurs d'activation : ils fournissent les points d'entrée des tâches de l'OS attachées au composant actif et prennent en charge leurs demandes d'ordonnancement.

Généralement, les composants de contrôle sont des composants « avec état » et activement partagés. Pour les protéger d'éventuels accès concurrents :

– Il est possible de les définir en tant que « composants protégés ».
– Pour une protection plus fine des variables globales accédées, un tel composant peut requérir une ressource de l'OS de type *mutex*.

5.3.5.2 Exemple du composant `CompA`

La figure 5.7 illustre les points spécifiés précédemment. Sur cette figure, les interfaces `scheduler`, `runner` et `mutex` concrétisent les liaisons entre la membrane et les services du système d'exploitation.

Pour cet exemple, l'intercepteur `inItcpt_inS` met en œuvre le protocole « incrémenté » pour une liaison d'événement (cf. page 58). Il est implanté par un simple compteur accédé en écriture par (potentiellement) plusieurs fils d'exécution d'un producteur ou d'un consommateur (dans ce cas, `CompA`). Pour sérialiser les accès à cette variable globale du composant, un simple *mutex* est utilisé.
Le contrôleur `MailBox` est ici spécifié en tant que « ***`composant protégé`*** ». Il s'agit d'un composant autiste dont tous les appels entrants sur ses interfaces passives sont sérialisés, assurant ainsi la cohérence des données stockées dans la boîte aux lettres.
L'`ActivationController` implante dans ce cas le protocole d'activation pour un composant mono-actif : il gère les demandes d'activation en provenance de l'environnement en maintenant un compteur pour mémoriser les requêtes en attente. Son implantation gère également les

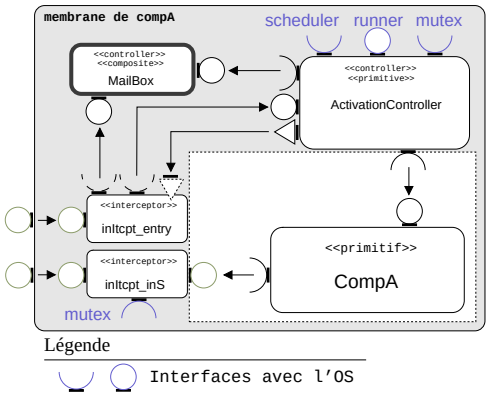


Figure 5.7 – Extrait de la membrane de `CompA`.

activations auprès de l'OS de la tâche attachée à `CompA` pour exécuter les méthodes métier définies par le concepteur : l'interface `scheduler` permet la demande explicite de l'activation de la tâche, l'interface `runner` est une interface atomique qui définit une simple méthode `run()` correspondant au point d'entrée de celle-ci.

À titre d'illustration, l'étape de génération des implantations des composants de contrôle donne les « comportements de niveau composant » (cf. page 74) pour l'intercepteur `inItcpt_entry` et pour le contrôleur `ActivationController` schématisés figure 5.8[59].

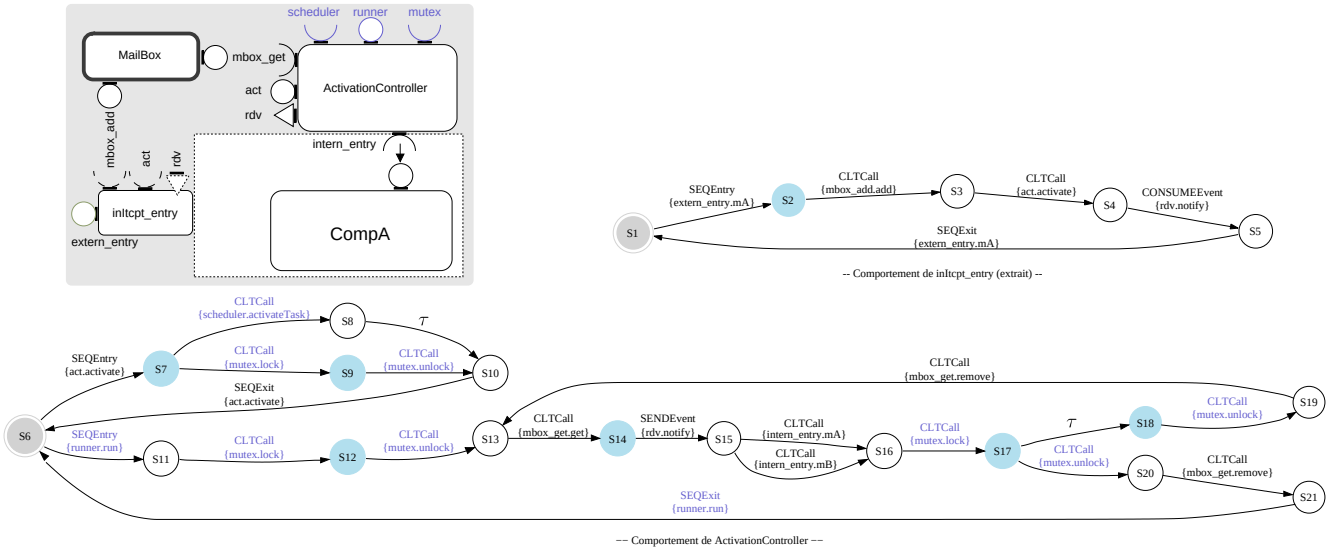


Figure 5.8 – Comportements d'un intercepteur et d'un contrôleur de la membrane de `CompA`.

La séquence `extern_entry.mA` de l'intercepteur d'entrée est invoquée dans le contexte du fil d'exécution de l'appelant. Le code associé au pas d'exécution local de l'état `S2` assure la sérialisation des paramètres d'entrée[60] qui vont être stockés dans la boîte aux lettres (action `in_Mbox.add`). L'activation est ensuite explicitement demandée auprès de l'`ActivationController`. Ce même fil d'exécution entre donc dans la séquence `act.activate` de ce dernier. Si le composant `compA` n'est pas déjà en cours d'activation (il s'agit ici d'un mono-actif), l'activation de la tâche de l'OS qui lui est associée est explicitement demandée, sinon, la demande d'activation est mémorisée en accès protégé (état `S9`). Enfin, le fil d'exécution se met en attente (demande de consommation de l'événement `rdv.notify`, transition `S4 -> S5`).
La séquence `runner.run` du contrôleur d'activation est invoquée par la tâche du système d'exploitation attachée à `CompA` lorsque celle-ci a été activée (le contrôleur est un composant activement partagé). Le compteur des demandes d'activation est éventuellement décrémenté (état `S12`), les informations récupérées de la boîte aux lettres[61], puis l'événement `notify` sur l'interface `rdv` est levé (c'est la sémantique du protocole « rendez-vous » pour les interfaces de service, l'appelant est débloqué lorsque l'appelé commence effectivement son exécution). Un simple

[59] Cette figure illustre notamment la « vue comportement » spécifiée dans le chapitre 4. Dans ce cas, c'est la vue comportement de niveau infrastructure qui est représentée, un raffinement de l'automate de comportement de niveau applicatif du composant `CompA`.

[60] Pour cet exemple, la méthode `mA` ne peut définir de paramètre de sortie puisque le protocole associé à la liaison est de type « rendez-vous ».

[61] Il faut noter que la politique par défaut pour récupérer un « message » de la boîte aux lettres est FIFO. Cependant, il est possible d'en paramétrer d'autres, comme par exemple par priorité.

switch permet de *router* le fil d'exécution vers la méthode métier donnée par les informations récupérées de la boîte aux lettres (dans le cas de notre exemple, il s'agit d'un appel à `mA` ou `mB`). Pour cet exemple d'exécution, la méthode `mA` du composant `CompA` sera donc appelée. Lorsque cette méthode métier retourne, soit une nouvelle demande d'activation est directement traitée, soit le fil d'exécution de la tâche se termine dans l'attente d'une nouvelle activation explicite (dans les deux cas, les informations associées à la requête métier qui vient de terminer sont supprimées de la boîte aux lettres).

5.3.6 Autres implantations et caractéristiques de l'infrastructure

Dans le contexte de ce document, nous ne donnerons pas tous les détails techniques relatifs au processus de génération de l'infrastructure d'exécution et aux implantations de l'ensemble des descripteurs de contrôle TINAP pour les composants et les liaisons.
L'objectif de cette section est de décrire brièvement certains de ces aspects qui n'ont pas été traités jusqu'à présent.

5.3.6.1 Autres descripteurs de contrôle TINAP

A. Descripteurs d'activité

Les autres descripteurs d'activité TINAP s'implantent de manière assez similaire à celui présenté dans la section précédente (il s'agissait de la prise en charge du « protocole d'activation » pour composant mono-actif correspondant au descripteur `BasicMonoActiveDescriptor` de la figure 4.20, page 56). Les trois descripteurs présentés sur la figure 4.20 sont basés sur les deux contrôleurs élémentaires que nous avons présentés (la boîte aux lettres pour mémoriser les données nécessaires à l'exécution des requêtes et le contrôleur d'activation) :

- Le contrôleur d'activation qui implante la gestion d'un composant multi-actif fournit autant d'interfaces `runner` (cf. figure 5.7) qu'il y a de tâches du système d'exploitation attachées à l'exécution du composant métier. À chaque demande d'activation en provenance de l'environnement, l'une des tâches du « *pool* » disponible exécute alors la requête, dans le cas contraire, les demandes sont mémorisées.
- Le protocole d'activation « et logique » est pris en charge par une boîte aux lettres spécifique qui se charge alors de demander l'activation du composant métier auprès de l'`ActivationController` lorsque la condition d'activation est vérifiée (cf. page 57).

Le descripteur pour composant mono-actif périodique est implanté par un contrôleur d'activation qui fournit donc une unique interface `runner`. Le code de la méthode `run()` utilise alors un service d'alarme fourni par le système d'exploitation pour gérer l'activation périodique du composant métier.

B. Descripteur pour composant protégé

Les composants de contrôle générés pour protéger les accès d'un composant métier sont schématisés sur la figure 5.9.

L'implantation d'un composant protégé est simple : les interfaces métier sont interceptées et les appels déroutés vers un contrôleur disposant d'une référence vers un sémaphore de l'OS (dont la capacité maximale est spécifiée par le concepteur, cf. page 61). Un « jeton » du sémaphore est demandé lorsqu'un fil d'exécution cherche à entrer dans l'implantation métier (lors d'une invocation entrante ou lors du retour d'une invocation sortante), et est libéré lorsqu'il en sort (lors d'une invocation sortante ou lors du retour d'une invocation entrante).

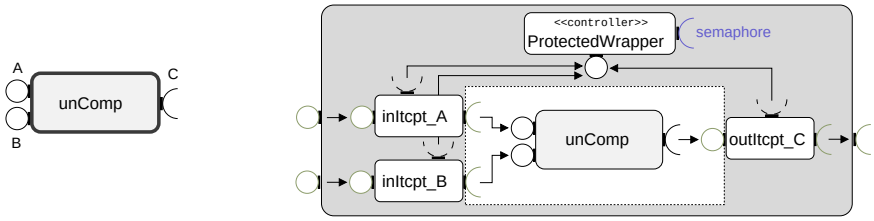


Figure 5.9 – Un composant protégé spécifié au niveau applicatif (a) et implantation de sa membrane au niveau infrastructure (b).

5.3.6.2 Non-préemption pour les liens d'activation

Un point important à prendre en compte au niveau de la génération de l'infrastructure d'exécution est le caractère prioritaire des liaisons d'activations entre composants. En effet, une requête d'activation, d'un composant émetteur, jusqu'à l'interface d'activation d'un composant actif se fait par appels de méthodes dans le contexte de cet émetteur (et dont le chemin d'exécution est susceptible de traverser des intercepteurs placés au sein des membranes des composites). Une telle demande d'activation peut aboutir à l'exécution d'une activité très prioritaire, il ne faut donc pas que la requête puisse être préemptée par d'autres activités de l'application.

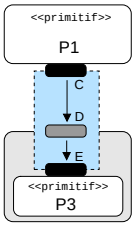


Figure 5.10 – Laison composite non préemptable.

Cette particularité est prise en charge par l'infrastructure d'exécution telle que représentée par la figure 5.10. Elle schématise la liaison composite donnée figure 5.3 (page 95).
Au niveau applicatif, il s'agit d'une liaison d'activation initiée par le composant `P1` à destination de l'interface d'activation du composant `P3`. Dans ce cas, la non-préemption de cette liaison est mise en œuvre au sein des intercepteurs de sortie `C` de l'émetteur et d'entrée `E` du destinataire. Ils sont alors attachés à une interface requise du système d'exploitation et la génération de leurs implantations assure les appels nécessaires pour mettre en œuvre la non-préemption du chemin d'exécution de cette liaison composite (dans notre cas, par l'intermédiaire des priorités attachées aux tâches de l'OS).

5.4 Méta-modélisation d'une « vue infrastructure »

Le processus de génération de l'infrastructure d'exécution TINAP est transparent pour le concepteur, c'est-à-dire qu'aucune intervention de sa part n'est nécessaire pour rendre l'application TINAP exécutable. Cependant, il est possible de « rendre visible » l'architecture générée à ce niveau d'abstraction au sein de l'outil de modélisation.

Ainsi, le méta-modèle présenté figure 5.11 permet de caractériser précisément les concepts employés à ce niveau d'abstraction.

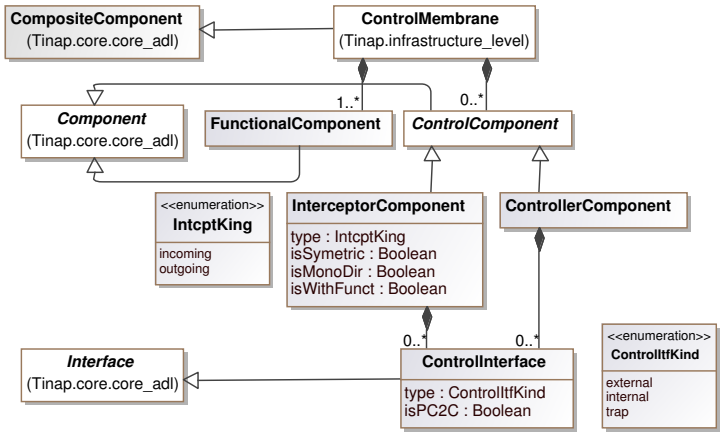


Figure 5.11 – Méta-modèle pour caractériser une architecture TINAP de niveau infrastructure.

5.5 Bilan sur le niveau infrastructure

L'infrastructure d'exécution TINAP implante les concepts de haut niveau proposés au concepteur pour spécifier son applicatif, notamment les mécanismes définis au sein du « modèle de concurrence » de la « vue dynamique et de contrôle », mais aussi pour implanter la perspective architecturale applicative (notamment par l'intermédiaire des intercepteurs, par exemple disposés autour des composites) à l'exécution.
Dans notre démarche, cette infrastructure est elle même implantée sous forme d'assemblages de composants – des intercepteurs et des contrôleurs – qui « se disposent » alors autour des composants applicatifs, au sein des membranes.

Au sein de l'implantation de l'infrastructure d'exécution, nous avons cherché à découpler la mise en œuvre des modèles de communication (implantés par des intercepteurs spécifiques) de celle liée aux autres (non-)fonctionnalités (implantées par des contrôleurs). Il s'agit de séparer la préoccupation d'interaction entre composants des autres aspects (dans ce chapitre, il s'agit essentiellement des contrôleurs d'activations des composants applicatifs actifs).

Les composants utilisés pour implanter cette couche d'infrastructure d'exécution sont totalement conformes aux concepts du « noyau » présentés dans le chapitre précédent. Notamment la vue comportementale de niveau composant qu'il est possible d'exploiter à ce niveau d'abstraction comme illustré figure 5.8. Par extension, la vue comportementale de niveau architecture se construit également à ce niveau d'abstraction, correspondant à la représentation du comportement métier « mergé » à celui de l'infrastructure (il s'agit d'une démarche de raffinement). Il en est de même pour les mécanismes caractérisant le modèle de concurrence de TINAP, qui ont

un sens à être utilisé également par les composants de contrôle, par exemple avec le contrôleur `Mailbox` en tant que composant protégé. Sur ces aspects, un exemple d'implantation plus conséquent est proposé comme étude de cas du chapitre 6.
Enfin, la spécialisation des composants en « intercepteurs » et en « contrôleurs » permettant de préciser un rôle (ou une sémantique) spécifique ayant du sens à ce niveau d'abstraction.

Cette section concrétise l'utilisation du « patron de conception/programmation orientée membrane » introduit chapitre 3. Dans le cadre de cette thèse, nous proposons de l'utiliser comme un moyen pour implanter les concepts de haut niveau proposés au niveau applicatif, et non pas seulement pour superposer des fonctionnalités optionnelles comme c'est le cas généralement dans les implantations FRACTAL.

Il est tout à fait envisageable, après l'étape de génération de l'infrastructure d'exécution, d'appliquer un algorithme de fusion pour transformer l'architecture des membranes en un code monolithique. Cet aspect constitue une perspective du travail présenté dans ce chapitre et pourrait s'appliquer à différentes échelles (code métier fusionné au « code de contrôle » d'un primitif donné, fusion complète d'un composite, c'est-à-dire de sa membrane et de ses sous-composants, etc) et pourquoi pas de manière transverse à l'architecture (par exemple en centralisant l'ensemble des contrôleurs d'activité TINAP en un unique, partagé par tous les compsants actifs de l'application).

Enfin, la prochaine section dresse quelques caractéristiques de l'implantation sous forme de composant pour la couche systèmes d'exploitation avec THINK.

5.6 Niveau système d'exploitation

L'objectif de notre démarche est d'être en mesure d'exploiter le paradigme composant pour la conception d'un système dans son intégralité (cf. section 3.1.3). À ce niveau d'abstraction, il s'agit donc de fournir par assemblage de composants les services élémentaires du système d'exploitation nécessaires à l'exécution d'une application TINAP.
La construction de système d'exploitation à base d'une spécification architecturale est l'objectif initial de THINK . Comme nous l'avons présenté (page 11), THINK s'accompagne d'une bibliothèque de composants – KORTEX – implantant les composants de la couche d'abstraction matérielle (HAL, *Hardware Abstraction Layer*) pour différentes plates-formes (ces composants réifient les exceptions, la MMU, le cache ou encore les pilotes de périphériques (écran, disque dur, clavier, etc), ainsi que les services classiques de système d'exploitation minimaliste (gestion de la mémoire, des systèmes de fichiers, de l'ordonnancement, des protocoles de communications, etc). Les assemblages de ces briques de base se spécifient par l'intermédiaire d'une extension de FRACTAL-ADL et de sa chaîne de compilation associée.

L'ensemble des concepts architecturaux nécessaires à ce niveau d'abstraction (c'est-à-dire relatifs à la « vue structurelle » de niveau OS) sont ceux spécifiés dans le paquetage `core_adl` de TINAP (cf. page 37). Les aspects dynamiques qu'il est possible d'attacher à ces composants se limitent au support de la reconfiguration. En effet, l'unique modèle de communication nécessaire est l'appel de méthode synchrone et les personnalités de composants proposées (composants actifs ou protégés) nécessaires à des niveaux d'abstraction plus élevés n'ont pas de sens à être utilisées à ce niveau de fonctionnalités.

La possibilité de concevoir le système d'exploitation par construction, et ce, à un fin niveau de granularité, permet de n'embarquer au sein de l'exécutif les seuls services élémentaires nécessaires à l'infrastructure d'exécution TINAP, permettant ainsi de réduire l'empreinte mémoire du binaire à déployer.

Sans entrer dans les détails, les composants actuellement disponibles au sein de KORTEX ne répondent pas à l'ensemble des critères (énumérés dans [Stankovic and Rajkumar, 2004]) nécessaires à la conception de « système d'exploitation temps-réel ». Ils assurent cependant la mise en œuvre d'un support multi-tâche, d'un ordonnanceur préemptif à priorité (et « *round-robin* »), de mécanismes de synchronisation (sémaphores) et d'implantation de primitives de gestion du temps, constituant un fondement suffisant pour les expérimentations menées dans le contexte de cette thèse.
Pour ces expérimentations, nous avons considéré que l'ensemble des services propres au système d'exploitation étaient encapsulés au sein d'un composite. Celui-ci fournit alors un ensemble d'interfaces de « type OS » (caractérisées figure 4.43, page 85). Les liaisons entre les « interfaces OS » requises par les composants de l'infrastructure d'exécution (ou par les composants applicatifs) et celles fournies par le système d'exploitation sont assurées automatiquement à l'étape de compilation. Nous avons également implanté des composants spécifiques, notamment pour le support d'un modèle de tâches proche de celui proposé par les spécifications OSEK [OSEK, 2003], ou la mise en œuvre d'un prototype d'ordonnanceur EDF (évoqué chapitre 6, section 6.3.2.3).

Vers une prise en charge de la portabilité. Il est intéressant de considérer la possibilité d'exécuter une application TINAP au-dessus de différents systèmes d'exploitation. Pour cela, deux possibilités sont envisageables : (1) générer l'infrastructure d'exécution en fonction des API (et des fonctionnalités offertes) de l'OS cible sous-jacent, ou (2) se baser sur une architecture de composants fournissant les interfaces nécessaires à l'infrastructure TINAP mais dont les implantations sont basées sur l'OS cible. Dans ce deuxième cas, les composants font office de couche de virtualisation entre l'infrastructure TINAP et l'OS cible. C'est sur ce principe que nous nous sommes basés pour notamment implanter des prototypes au-dessus de LINUX/POSIX[62].

5.7 Conclusion du chapitre

Ce chapitre met en évidence un élément fondamental de TINAP : celui de promouvoir un modèle de composant canonique pouvant être utilisé à différents niveaux d'abstraction, aussi bien pour celui de l'applicatif dont des éléments de haut niveau sont proposés au concepteur qu'au niveau de l'infrastructure d'exécution et du système d'exploitation. L'ensemble de concepts communs caractérisé au sein de TINAP définit la manière de structurer le logiciel (ses fonctionnalités, au sein de la « vue structurelle »), de spécifier les aspects dynamique et de contrôle à partir de ces entités structurelles de première classe, et enfin, d'abstraire la structure et le comportement interne des composants (au sein des « vues implantation et comportementales »).
De plus, nous promouvons une approche constructiviste pour rendre concret les assemblages des composants applicatifs avec ceux de l'infrastructure d'exécution et ceux de l'OS (fournissant les services élémentaires nécessaires). Cela correspond alors à une « représentation aplatie » de l'ensemble des composants fonctionnels qui s'exécutent au-dessus de la plate-forme matérielle. Il est notamment possible de raisonner sur le comportement global résultant de l'assemblage des composants du système, au même titre qu'au niveau applicatif (présenté dans le chapitre 4). Cela permet d'assurer la traçabilité entre les différents niveaux d'abstraction. De plus, l'homogénéisation des concepts permet l'utilisation d'outils dédiés (alors basés sur ces concepts communs) à ces différents niveaux.
La capacité de pouvoir raisonner en terme d'architecture pour ces trois niveaux permet notamment de contrôler à fine granularité l'ensemble des fonctionnalités du système, et de n'embarquer que celles strictement nécessaires.

[62] Cette possibilité peut s'avérer également intéressante dans le but de tester certaines parties d'une architecture applicative avant le déploiement final sur une cible matérielle.

Le travail présenté dans ce chapitre présente cependant des limitations. En effet, si nous considérons par exemple les concepts énoncés pour le niveau applicatif : nous proposons de mettre en œuvre les aspects extra-fonctionnels au sein de la membrane en utilisant le patron énoncé chapitre 3. Cependant, nous n'avons pas considéré comment les implantations des concepts TINAP (membranes pour composants actifs, implantation des modèles de communication et de synchronisation, etc) pouvaient cohabiter avec les implantations des contrôleurs spécifiés par FRACTAL pour assurer l'introspection et la reconfiguration dynamique. Et il s'agit bien entendu d'un problème complexe à résoudre dans le cas général. En outre, cette reflexion conduit également à considérer quelle doit être la « perception » des entités TINAP à l'exécution. En ce sens, faut-il définir et implanter une API qui permet de réifier à l'exécution les notions de composants actifs, de protocole de synchronisation attaché à une liaison d'événements, d'intercepteurs ou de contrôleurs, etc ? Ces aspects constituent pour le moment des perspectives de notre proposition.

Chapitre 6

Expérimentations et évaluations

Sommaire

6.1 Introduction 107
6.2 DeckX : un « contrôleur » de flux multimédias 108
6.2.1 Présentation de l'application 108
6.2.2 Interface TINAP de flux audio 112
6.2.3 « TINAPisation » de l'application 113
6.2.4 Discussions et évaluations 115
6.3 Implantation du modèle d'exécution ACCORD 117
6.3.1 Présentation de l'approche ACCORD 117
6.3.2 « TINAPisation » du modèle d'exécution ACCORD 121
6.3.3 Discussions et évaluations 125
6.4 Conclusion sur les expérimentations 127

6.1 Introduction

Deux études de cas ont été menées dans le but d'expérimenter notre approche. En premier lieu, nous avons cherché à expérimenter l'espace de conception TINAP de niveau applicatif pour implanter une application de contrôle et de filtrage de flux multimédias (section suivante). Une extension de TINAP est à ce titre proposée pour prendre en compte au niveau applicatif la spécification d'interfaces de flot de donnée audio, ainsi que les liaisons qui leur correspondent. En second lieu (section 6.3, page 117), nous nous sommes intéressé à implanter un « langage de conception » de haut niveau (ACCORD, développé par le LLSP) au sein de TINAP. Il s'agit d'une ingénierie des mécanismes extra-fonctionnels nécessaires aux spécifications ACCORD basée sur les concepts et patrons proposés au sein de l'infrastructure d'exécution TINAP. Il s'agit d'une extension du niveau applicatif de notre approche : un « descripteur de contrôle » spécifique est proposé au concepteur pour caractériser ses composants conformément au modèle de concurrence ACCORD, ces aspects ont alors été implantés dans les membranes correspondantes.

6.2 DECKX : un « contrôleur » de flux multimédias

6.2.1 Présentation de l'application

L'objectif de l'application est de pouvoir contrôler des flux multimédias numériques (musique ou vidéo par exemple) par l'intermédiaire de disques vinyles.

6.2.1.1 Disque vinyle et techniques de « scratch », de l'analogique au numérique

Le disque vinyle. L'ancêtre de ce support analogique est le « 78 tours » (pour 78 rotations par minute) et connut son âge d'or entre 1920 et 1950. C'est donc au milieu du siècle dernier que le disque vinyle (ou plus simplement « vinyle ») tel que nous le connaissons aujourd'hui fait son apparition. Il s'agit d'un disque phonographique, support d'enregistrement sonore, disposant de deux faces sur lesquelles est gravé un sillon microscopique. Il s'agit d'un support analogique, c'est-à-dire que la quantité d'information qu'il contient n'a pas de limite définie, à la différence du disque compact (« CD »), source numérique pour laquelle le signal sonore est obtenue par échantillonnage. Le disque se lit avec une *platine vinyle* sur laquelle est montée une *cellule* avec un diamant qui parcours le microsillon par contact direct. Lors de la lecture, ce sont donc les vibrations qu'entrainent ce dernier sur la cellule qui sont directement amplifiées et audibles.

A partir de 1983, les vinyles ont fait place aux disques compacts mais ils sont loin d'avoir complètement disparu de la circulation ! En effet, dans les années 90, ils ont été remis à l'honneur massivement avec l'avènement de la « culture DJ », essentiellement issue de l'émergence des *musiques électroniques* et du *hip-hop*. En effet, seules les platines vinyles disposaient à l'époque d'une vitesse réglable, condition sine qua non pour enchaîner des morceaux dans un *mix* et offraient le support physique idéal pour l'apparition d'un nouvel art de création musicale, le *turntablism*[63], et notamment les techniques de *scratching*, un procédé consistant à faire tourner à la main le disque sous la tête de lecture de telle sorte à créer un effet sonore spécial. L'exemple le plus démonstratif dans ce domaine est sans doute le groupe français « Birdy Nam Nam »[64] fondé en 2002, fruit d'une évolution qui a conduit ces instrumentistes de la technique à la musique.

Vers le numérique. La fin des années 90 est marquée par le début d'une nouvelle ère, celle de l'industrialisation de divers équipements pour *DJ's* basés sur des supports d'enregistrements numériques. Les premières machines à entrer sur le marché utilisent le support CD. Ainsi, au lecteur « de salon » est ajouté un *pitch* qui permet de faire varier la vitesse de rotation du disque et donc de modifier en temps réel la vitesse du morceau en cours de lecture. D'autres dispositifs, équipés d'un plateau mécanique en rotation ont aussi tentés de « reproduire » l'interface physique de la platine vinyle. Cependant, toutes ces tentatives, aussi abouties soient elles techniquement, ne se sont jamais véritablement approchées du « touché » tangible ni des effets sonores obtenus en *scratchant* à partir d'une platine réelle.

Ce n'est que très récemment qu'une nouvelle technologie est apparue : au lieu de tenter d'imiter le support physique, l'idée est plutôt de l'utiliser tel quel. Elle consiste alors à graver un signal spécifiquement modulé sur un disque vinyle pouvant être ainsi lu par n'importe quel type de platine. Ce signal encode un certain nombre d'informations, que l'on qualifiera de *timecode*. Couplé à un ordinateur et à une carte son qui échantillonne le signal transmis par la platine, et à un processus d'analyse de ce signal qui permet de récupérer ces informations, il est alors possible de contrôler (par exemple) la lecture d'un fichier audio numérique. Dans ce cas, tous les

[63] Mot américain désignant l'art de créer de la musique grâce aux platines et aux disques vinyles. « *Un turntablist est une personne qui utilise les platines non pour jouer de la musique mais qui isole et manipule les sons pour créer de la musique* » (*DJ Babu, Dilated Peoples, 1996*)

[64] `http ://www.birdynamnam.com`

mouvements infligés par le DJ sur le vinyle sont immédiatement répercutés sur la source numérique comme si cette dernière était directement gravée sur le disque. Mais cette technologie ouvre également d'autres horizons, comme par exemple la possibilité de scratcher une vidéo, de faire varier la fréquence de coupure d'un filtre qui s'applique parallèlement à la musique en cours de lecture, ou pourquoi pas, comme une expérience qui semble avoir été menée, de contrôler une table de massage électronique !
Il faut noter que plusieurs industriels commercialisent déjà ce type de produits qui ont fait leur preuve dans le « milieu du DJ'ing » et qui se démocratisent significativement depuis peu. On peut notamment citer FINAL SCRATCH commercialisé par la société STANTON MAGNETICS[65], SCRATCH LIVE de la société SERATO[66] ou encore MSPINKY proposé par CYCLING74 [67].

6.2.1.2 Exploitation du signal modulé

Dans cette sous-section, nous nous attachons à décrire comment cette technologie est exploitée dans le contexte de l'application.

Paramètres essentiels. Le signal modulé, gravé sur le disque vinyle, doit permettre d'encoder les paramètres suivants :

- **La direction.** Mécaniquement, les techniques de scratch reposent sur un va-et-vient de la cellule de la platine autour d'une courte portion sonore, le disque tourne donc dans les deux directions.
- **La vitesse de lecture.** Le disque est susceptible de tourner à une vitesse quelconque, pendant un scratch par exemple pour lequel des accélérations et décélérations très fortes peuvent être infligées, ou simplement lorsque le DJ modifie le pitch de la platine ou arrête la rotation.
- **La position absolue.** Il correspond à l'emplacement du bras de la platine vinyle sur le disque. Typiquement, lorsque l'on utilise un disque classique, si l'on place la cellule au début du disque, on entendra le début du morceau, et réciproquement pour la fin. Il s'agit donc d'encoder dans le signal ce positionnement absolu permettant de connaître l'endroit exact de l'emplacement de la cellule.

MSPINKY. Comme nous l'avons fait remarquer, différents industriels commercialisent cette technologie de contrôleur analogique/numérique. Bien que ces implantations proposent toutes (à quelques fonctionnalités près) le même type d'application pour l'utilisateur final, les techniques de modulation pour obtenir le signal *timecodé* est propre à chacune d'elles. Notre cas d'étude repose sur celle proposée par MSPINKY [Wardle, 2006].

La figure 6.1 représente un extrait en stéréo de 8 millisecondes (échantillonné à une fréquence de 44100 Hz) du signal utilisé par MSPINKY. Ce signal a été créé par le développeur initial de l'application (par une transformée de Fourier inverse) par une combinaison linéaire, dans le domaine fréquentiel, d'une forte tonalité constante et d'un ensemble de sous-tonalités modulées sur des bandes de fréquence séparées.
À titre d'illustration, la figure 6.2 présente le spectre fréquentiel obtenu par transformée de Fourier d'une portion du signal généré.

[65] `http ://www.stantondj.com`
[66] `http ://www.serato.com/`
[67] `http ://www.cycling74.com/`

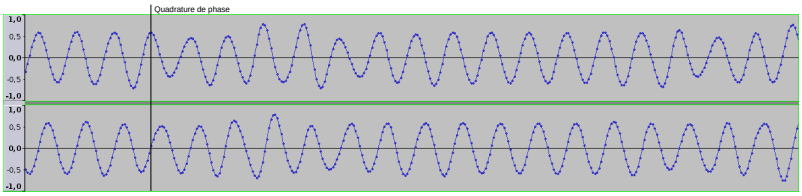


Figure 6.1 – Extrait du signal MSPINKY en stéréo.

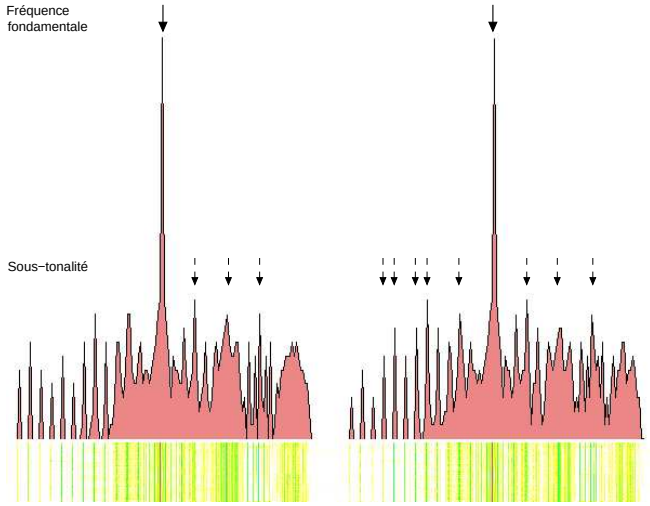


Figure 6.2 – Spectre fréquentiel du signal.

Processus d'analyse. Il consiste donc à récupérer les informations de *timecode* citées ci-dessus. Comme nous pouvons le voir sur la figure 6.1, les sinusoïdes des deux canaux sont en quadrature de phase. Selon le sens de lecture du disque, ce décalage de phase sera donc positif ou négatif, permettant alors de connaître la **direction**.
Sur la figure 6.2, la fréquence fondamentale du signal est clairement mise en évidence. A vitesse normale, cette fréquence est fixée à 1200 Hz. Lors du processus d'analyse, la détermination de cette fréquence fondamentale permet donc de connaître la **vitesse de lecture** du vinyle (pour estimer précisément cette fréquence fondamentale en temps réel, le processus d'analyse implanté par MSPINKY se base sur des travaux établis par [Quinn and Hannan, 2001]).
La **position absolue** s'obtient par analyse des sous-tonalités du signal. En effet, leur combinaison permet de coder « en dur » la valeur de ce positionnement. Cette valeur est codée sur un flottant qui commence donc à 0 et et qui augmente environ 155 fois par rotation du vinyle. Elle est mise à jour approximativement 86 fois par seconde (la durée maximale du signal ainsi codé est de quinze minutes).
Nous appelons une *unité de timecode* une structure de données stockant à un instant `t` les valeurs de ces trois paramètres.

6.2.1.3 Fonctionnement global de l'application

Un exemple de configuration logicielle et matérielle d'utilisation concrète de l'application est donné figure 6.3.

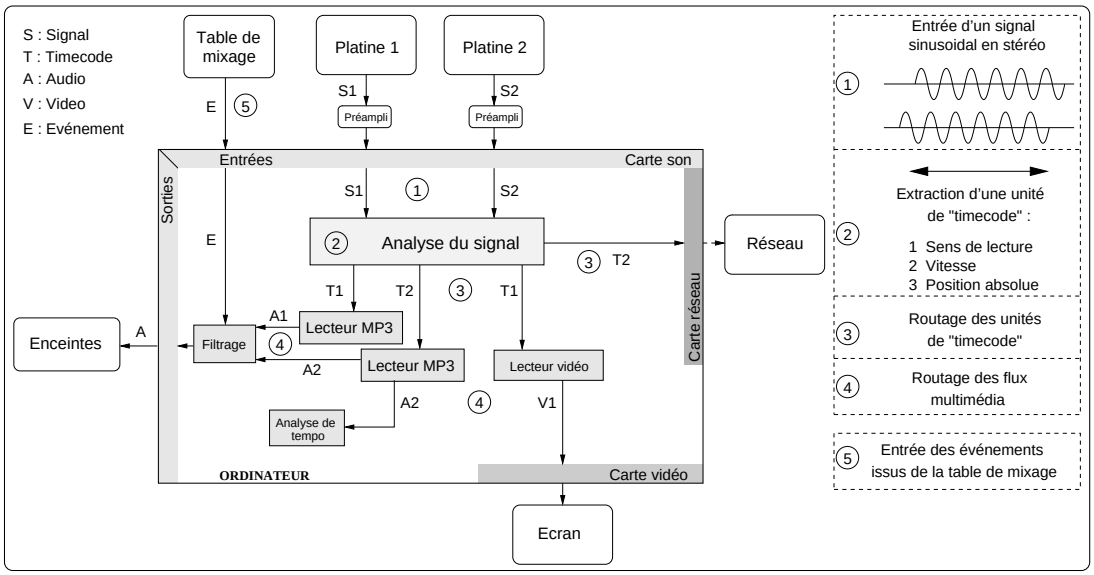


Figure 6.3 – Architecture logicielle « gros grain » et connectique externe.

Sur cet exemple, deux platines vinyles (chacune d'elle utilise un disque sur lequel est gravé le signal MSPINKY précédemment décrit) sont connectées aux entrées de la carte son de la machine. Il s'agit donc de deux flux audio stéréo échantillonnnés en entrée de la carte son.

A titre d'exemple, les différentes étapes de traitements logiciels peuvent être les suivants (les numéros des étapes correspondent à ceux donnés sur la figure) :

1. Les flux audio échantillonnnés par la carte son sont transmis périodiquement au processus d'analyse.
2. Pour chaque flux entrant, l'application se charge d'extraire des « unités de timecode » tel que nous l'avons décrit dans le paragraphe précédant.
3. Ces informations de timecode constituent alors un flux de contrôle qu'il est envisageable de rediriger vers n'importe quelles entités logicielles. Dans notre exemple, le timecode issu de chaque platine est routé vers deux lecteurs MP3 distincts. Mais on peut imaginer rediriger l'un des flux vers un lecteur vidéo pour en contrôler son déroulement, ou encore de rediriger un flux vers le réseau de telle sorte à ce qu'il soit consommé par d'autres clients répartis.
4. La dernière étape schématisée sur la figure concerne alors le routage des flux multimédias. Ceux issus des lecteurs MP3 sont dirigés vers la sorties de la carte son dont le pilote consomme les buffers périodiquement. On peut aussi imaginer vouloir rediriger le flux audio d'un lecteur MP3 vers un analyseur de tempo pour en extraire en temps réel le BPM (*Beat Per Minute*) du morceau en cours de lecture.

5. La table de mixage est un élément essentiel pour les techniques de mix et de scratch. Il s'agit d'un contrôleur matériel dont l'interface permet de modifier le volume ou les fréquences du son avant que celui-ci ne soit transmis aux enceintes pour être diffusé. Les flux audio peuvent ainsi être mélangés ou coupés volontairement par le DJ et/ou filtrés.

Dans notre expérimentation, nous utilisons une interface matérielle spécifique dont la manipulation par le DJ génère des événements interprétables par l'application.

On peut faire remarquer que les différentes connexions entre blocs logiciels établies dans cet exemple sont susceptibles d'évoluer au cours du cycle de vie de l'application. En effet, on peut imaginer un routage des flux de timecode de manière assez arbitraire : par exemple un même timecode pour contrôler de multiples lecteurs MP3 ou vidéo simultanément, et il est intéressant de proposer cette fonctionnalité à l'utilisateur. Nous reviendrons par la suite sur cet aspect dynamique de l'architecture.

6.2.2 Interface TINAP de flux audio

Les données audio-numériques se caractérisent de manière spécifique. Au sein de notre approche, nous avons ainsi défini une interface particulière pour caractériser ce type de flot de données (schématisée figure 6.4). Elles se caractérisent comme une extension des interfaces de flot définies chapitre 4 (page 41).

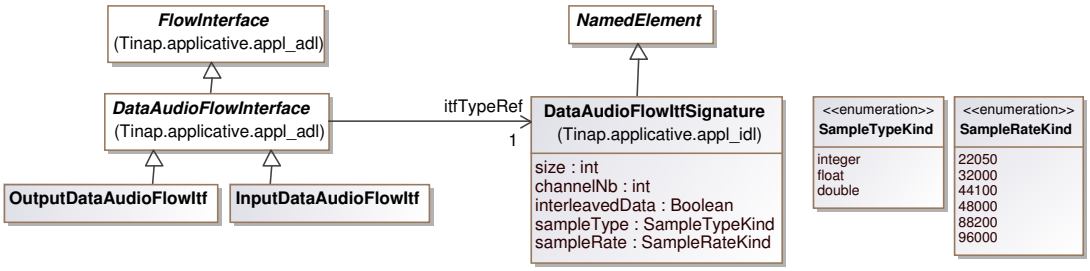


Figure 6.4 – Définition des interfaces de flux audio.

Ces interfaces sont typées par une signature qui caractérise précisément les flux audio qui s'échangent entre composants :

- Les données audio sont propagées au sein d'une structure scalaire (un tableau) dont le nombre d'éléments est caractérisé par l'attribut `size`.
- Il est possible au sein d'une même structure de propager plusieurs canaux audio (attribut `channelNb`). Typiquement, un flux en stéréo est composé de deux canaux.
- Pour un flux composé de plusieurs canaux, les échantillons peuvent être mémorisés de manière entrelacés ou non-entrelacés (attribut `interleavedData`).

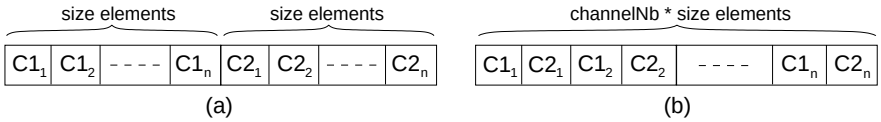


Figure 6.5 – Exemple d'un tableau d'échantillons (a) non-entrelacés et (b) entrelacés pour un flux audio en stéréo.

- Les attributs `sampleType` et `sampleRate` permettent respectivement de renseigner le type de donnée utilisé pour coder les échantillons et le taux d'échantillonnage du flux audio.

La spécification de ces informations à l'étape de conception permet de s'assurer que les composants applicatifs communiquent par l'intermédiaire de « types de flux audio compatibles » et permet de prendre en charge de manière automatique le processus de copie des buffers entre composants lors de l'étape de génération de l'infrastructure d'exécution.
Une simple convention de projection est définie entre la signature d'une telle interface et une structure de données utilisée par les implantations de composants.

La sémantique d'une interface d'entrée de flux audio multi-liée est particulière. Dans ce cas, la politique associée à la consommation est celle d'une « addition ». Les occurrences de données audio en provenance de différents producteurs sont alors additionnés en une unique occurrence. D'un point de vue fonctionnel, cet aspect permet de « mélanger » différentes sources audio au sein d'un unique signal.

6.2.3 « TINAPisation » de l'application

Cette section est dédiée à la présentation de l'architecture TINAP de l'application. Il s'agit de détailler la manière dont les préoccupations fonctionnelles ont été séparées ainsi que les aspects dynamiques liés à son fonctionnement.

6.2.3.1 Partie analyse et lecteur audio

Un extrait de l'architecture TINAP (vue dynamique) de l'application est donnée figure 6.6 (un symbole graphique est ajouté pour caractériser les interfaces de flux audio).

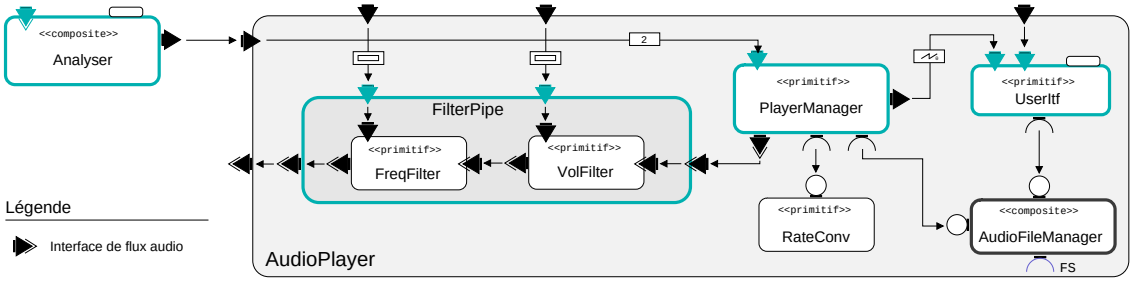


Figure 6.6 – Vue dynamique d'un extrait de l'architecture de l'application DECKX.

Les fonctionnalités attachées aux composants sont les suivantes :

- Le composant `Analyser` est le composant chargé de l'analyse du signal d'entrée issue d'une platine vinyle. En sortie, il produit des unités de timecode qui peuvent être dirigées en destination de multiples consommateurs comme nous l'avons évoqué précédemment.
- Le composite `AudioPlayer` encapsule l'ensemble des fonctionnalités d'un lecteur/contrôleur de fichier audio-numérique.
- Le composant `PlayerManager` est l'entité centrale du lecteur. Il s'agit d'un composant actif qui s'active sur la réception d'unité de timecode.
  À la réception d'une telle unité, il se charge principalement de récupérer une fenêtre d'échantillons audio auprès de l'`AudioFileManager` (qui requiert une interface « OS » `FS` pour utiliser le système de fichers sous-jacent), puis d'en modifier le taux d'échantillonnage (`RateConv`), d'en changer éventuellement le sens de lecture[68] et enfin de transmettre cette fenêtre au composite `FilterPipe`.

[68] C'est donc à ce stade que sont exploitées les informations de timecode : la fenêtre audio s'obtient en fonction de la position absolue, le taux de ré-échantillonnage est fonction de la vitesse et une direction négative impose d'inverser les échantillons du buffer.

– Le contenu du composite `FilterPipe` implante les fonctionnalités de filtrage (volume et fréquence). Il réceptionne les événements en provenance de la table de mixage. On peut remarquer que les sous-composants sont multi-implantés : une première interface d'entrée permet de mettre à jour les paramètres des filtres, une deuxième permet de faire transiter les buffers audios produits par le `PlayerManager` et sur lesquels seront alors appliqués ces filtres.
– Finalement, l'interface de sortie de flux audio de l'`AudioPlayer` produit les buffers audio qui seront envoyés à la carte son.
– Le composant `UserItf` est une interface (extrêmement basique), essentiellement destinée à permettre à l'utilisateur de choisir un fichier audio (un MP3 par exemple) donné. Nous considérons qu'il est à l'écoute d'événements en provenance du clavier.

6.2.3.2 Exemple d'architecture DECKX globale

Un exemple d'architecture globale de l'application DECKX est donnée figure 6.7. Elle permet d'analyser deux flux (donc en provenance de deux platines distinctes) en parallèle.

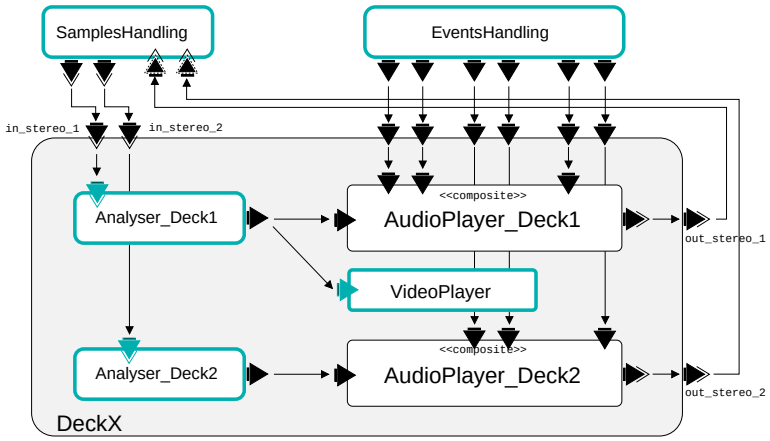


Figure 6.7 – Exemple d'architecture globale d'une application DECKX (avec deux entrées/sortie stéréo).

Dans le contexte de cette expérimentation, l'implantation a été particulièrement simplifiée. En effet, nous ne sommes pas allés jusqu'à implanter l'ensemble des services du système d'exploitation qui nous auraient permis d'exécuter l'application directement sur le matériel (pilote de la carte son, pilote USB de la table de mixage, etc). Nous nous sommes donc restreint à expérimenter le niveau applicatif et le niveau infrastructure de notre approche au-dessus de LINUX.
Sur la figure 6.7 sont représentés les deux composants `SamplesHandling` et `EventsHandling`. Ils implantent une couche de virtualisation entre LINUX et l'application, respectivement pour gérer l'interfaçage avec la carte son et les événements en provenance de la table de mixage et du clavier. Le Composant `SamplesHandling` est un composant périodique. À chaque cycle d'activation, il se charge de récupérer les échantillons audio en entrée de la carte son pour les fournir à l'application et recupère ceux en provenance de l'application pour les restituer à la sortie de la carte son.
Le composant `VideoPlayer` est une simple expérimentation d'un contrôleur vidéo. Nous avons simplement adapté une application existante en guise de prototype.

6.2.3.3 Autres caractéristiques

Contraintes temporelles. Elles se spécifient relativement à la période fixée par le composant actif `SamplesHandling`. En effet, considérant le composite `DeckX` de la figure 6.7, le temps de calcul lié aux traitements effectués entre l'occurrence d'une donnée en entrée (par exemple sur l'interface `in_stereo_1`) et sa conséquence (l'occurrence d'une donnée sur l'interface de sortie `out_stereo_1`) ne doivent pas dépasser la période fixée par le composant `SamplesHandling` sous peine d'*underflow* sur le buffer de sortie de la carte son. Il s'agit donc de contraintes de « bout en bout » qui se spécifient à partir de la vue comportementale de niveau architecture définie dans le chapitre 4. Plus précisément, si l'on se réfère à la figure 4.41 (page 83), qui fournit une « perspective orientée transaction » de l'application, il s'agit d'attacher cette échéance à la `Transaction 1`, qui correspond aux liaisons établies par la vue dynamique représentée figure 6.7 (pour la première platine permettant de contrôler un lecteur audio et vidéo, respectivement les composants `AudioPlayer_Deck1` et `VideoPlayer`).
Il faut noter que le choix de la période attachée au composant `SamplesHandling` est déterminant pour le bon fonctionnement de l'application. En réalité, plus elle est réduite, moins le délai entre les mouvements du DJ sur le disque et leurs conséquences en sortie de la carte est élevé. Cette latence doit être inférieure à 10 ms – il s'agit du seuil au delà duquel le retard est perceptible par l'être humain – pour que l'application soit réellement fonctionnelle.

Reconfiguration dynamique. Les aspects de reconfiguration qu'il est intéressant de considérer d'un point de vue fonctionnel concernent les liaisons spécifiées entre les composants d'analyse et les composants de lecture audio et vidéo. En effet, l'utilisateur de l'application peut être amené à les reconfigurer de telle sorte à choisir laquelle des platines contrôle le lecteur vidéo, ou encore pour être amené à contrôler les deux lecteurs audio à partir d'une unique platine.

6.2.4 Discussions et évaluations

Intérêts de TINAP. Du point de vue du concepteur, si nous comparons une « implantation courante » (c'est-à-dire monolithique, en « C élémentaire », dont certaines sont disponibles) de ce cas d'étude par rapport à une implantation basée sur TINAP, il en ressort en premier lieu les avantages établis du paradigme composant : la séparation des préoccupations et la représentation de l'architecture fonctionnelle résultante facilite grandement la compréhension de l'application. Il en est de même pour la représentation des entités actives (de la vue dynamique) à partir de l'architecture, comparée par exemple à la mise en œuvre des aspects multi-tâches généralement enfouis dans l'implantation. De plus, ces aspects sont clairement représentés par la « vue comportementale de niveau architecture » qui offre une perspective transverse à l'architecture des aspects dynamiques de l'application.

Relativement au domaine des applications audio-numériques, l'utilisation de TINAP présente différents avantages. Le paradigme composant pouvant s'employer pour spécifier des configurations de type « *pipes and filters* » convient particulièrement bien au domaine où le filtrage de flux audio prend une place prépondérante (comme c'est le cas pour le composite `FilterPipe` et ses sous-composants représentés figure 6.6).

C'est normalement à la charge du concepteur d'implanter les mécanismes de bufferisation des flux audio qui transitent le long de son application. Cette étape est généralement fastidieuse, source récurrente d'erreurs et son implantation ne devrait pas être associée au code métier. Au sein de TINAP, la prise en charge de ces aspects est transparente pour le concepteur. Ils sont gérés de manière automatique en fonction des descriptions architecturales de l'application et sont donc découplés du code purement fonctionnel.

En guise de perspective, il serait à terme intéressant d'enrichir TINAP pour le dériver en un atelier complet de conception d'applications audio[69].

Relativement à l'étude de cas. L'architecture de DECKX que nous avons étudiée est relativement simple et les aspects opératoires sont prépondérants par rapport aux aspects de contrôle. Il aurait été en effet intéressant de considérer davantage de synchronisations entre les différents modules logiciels (en effet, l'architecture résultante est essentiellement orientée « flot de données »). Une extension intéressante serait de rendre possible une synchronisation des différents lecteurs Mp3 en fonction d'une analyse de leurs tempos respectifs (il s'agit d'une caractéristique de mixage couramment employée dans les musiques électroniques). Cependant, l'étude de cas présentée dans la section suivante caractérise davantage de contraintes de synchronisation entre composants.
Toujours en guise de perspective, cette étude de cas s'avère intéressante pour mener des analyses hors-ligne d'ordonnançabilité ou de performances. Il s'agit alors de minimiser la latence globale du système jusqu'au point où la plate-forme matérielle supporte les traitements nécessaires sans *underflow*, et ce, en fonction du nombre de « platines virtuelles » voulu par l'utilisateur final.

Allocation. La vue dynamique du lecteur Mp3 (le composant `AudioPlayer`) représentée figure 6.6 définie trois sous-composants mono-actifs (les composants `FilterPipe`, `PlayerManager` et `UserItf`). À l'exécution, il en résulte que trois tâches du système d'exploitation seront allouées à ces composants pour un lecteur Mp3 donné. Lorsque l'on considère l'architecture globale de l'application présentée figure 6.7, et étant donné le caractère séquentiel et redondant des traitements effectués dans les deux lecteurs Mp3 ainsi qu'au sein des deux analyseurs en entrée, on peut considérer que cette allocation est loin d'être optimale. Considérant l'architecture actuelle, à l'opposé de ce choix d'allocation des entités actives, on pourrait considérer le composite `DeckX` comme unique composant mono-actif, l'ensemble des traitements associés aux lecteurs audio et vidéo seraient alors sérialisés au sein d'une unique tâche du système d'exploitation. Proposer un processus automatique, qui à partir d'une vue dynamique TINAP, serait susceptible de déterminer une allocation maximisant les performances tout en respectant un certain degré de parallélisme permettant de respecter les contraintes fonctionnelles de l'application sort bien entendu du contexte de cette thèse mais constitue néanmoins une perspective de recherche intéressante.

Couverture des concepts TINAP. Les concepts proposés dans notre approche (considérant également l'extension pour prendre en compte et implanter les liaisons de flot de données audio) correspondent tout à fait à ceux nécessaires pour implanter une application du type de DECKX. Les aspects extra-fonctionnels (notamment la gestion des activités, la mise en œuvre des interactions de flot de données audio) dont les implantations sont prises en charge de manière générative et automatique, et donc le découplage de ces aspects relativement à ceux du métier, constituent des points favorables quant à la conception de DECKX avec TINAP.
La souplesse offerte au concepteur pour spécifier des composants actifs et passifs, et implicitement des composants activements partagés, est absolument nécessaire pour concevoir ce genre d'applications. Et la possibilité de raisonner sur les aspects opérationnels de manière globale (par l'intermédiaire de la « vue comportementale de niveau architecture », qui exploite l'ensemble des informations disponibles) nous conforte dans ce choix de modèle de concurrence, plutôt que dans un modèle beaucoup plus restrictif comme il est généralement de mise avec les ADLs issus du domaine de l'embarqué.
Le comportement interne des composants TINAP est arbitraire : il s'agit de pouvoir implanter des composants fortement couplés avec leur environnement aussi bien que de simples filtres. Cette possibilité est nécessaire pour le domaine d'application que nous visons.

[69] Comme c'est le cas de certaines approches existantes telles PUREDATA, une implantation open-source de MAX/MSP (`http ://puredata.info`) ou CLAM (*C++ Library for Audio and Music*, `http ://clam.iua.upf.edu`).

6.3 Implantation du modèle d'exécution ACCORD

6.3.1 Présentation de l'approche ACCORD

ACCORD/UML [Tessier *et al.*, 2003, Gérard *et al.*, 2002] est une plate-forme de modélisation et d'exécution d'applications temps réel embarquées, orientées objets, et développée par l'équipe ACCORD au sein du CEA-LIST/LLSP.
L'étape de modélisation est basée sur une méthodologie UML dont l'objectif est d'assurer au développeur un continuum le long des différentes étapes du cycle de développement de l'application, de l'analyse des exigences jusqu'au prototypage. Un ensemble d'outils a donc été développé en ce sens (transformation de modèles, utilisation de patrons de conception spécialisés pour le temps réel et générateurs de code) ainsi qu'un framework opérationnel offrant une couche d'abstraction entre le modèle d'exécution des applications ACCORD et différents systèmes d'exploitation (actuellement au-dessus de VXWORKS et de LINUX).
Dans le contexte de ce document, nous ne détaillerons pas cet aspect méthodologique de l'approche. Pour plus de détails, le lecteur peut se reporter à [Gérard *et al.*, 2002, Gérard, 2002].

6.3.1.1 Modèle d'exécution ACCORD

La structure d'une application ACCORD est constituée d'un ensemble d'objets. L'approche fait la distinction entre objets passifs et objets actifs. Les objets passifs sont des objets « classiques » dans le sens où ils fournissent un ensemble de méthodes qui peuvent être partagées par différents objets actifs. Ils correspondent donc à des objets partagés et n'offrent pas de contrôle lié à la concurrence des accès. Le concept d'objet actif utilisé dans ACCORD définit des extensions temps réel au paradigme d'objet actif initialement proposé dans [Nierstrasz, 1987].

La notion d'objet temps réel. Un objet temps réel (RTO) peut être considéré comme une entité autonome gérant ses propres ressources de calcul.

Les invocations de méthodes fournies par les objets actifs se font par échange de messages (ou par réception de signaux). À un tel objet est donc associé un ensemble de tâches qui sont alors attachées aux traitements de ces invocations entrantes.
La structure d'un RTO au sein de l'approche ACCORD est constituée principalement de quatre composantes : un ensemble de méthodes fournies, un ensemble d'attributs, un contrôleur local et une boîte aux lettres, représentées figure 6.8.

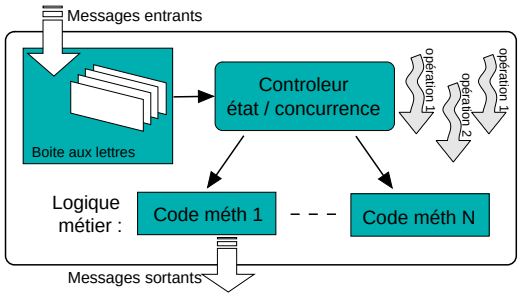


Figure 6.8 – Structure informelle d'un RTO.

Le RTO reçoit et stocke les messages entrants au sein de sa boîte aux lettres. Le contrôleur local est en charge de sa gestion, par exemple en attachant ou en détruisant des tâches associées au traitement d'un message et de la gestion des contraintes de concurrence (présentées ci-dessous). La boîte aux lettres contient tous les messages en attente d'exécution à un instant donné.

Le modèle de concurrence associé au RTO. Un objet temps réel peut être considéré comme un moniteur multi-tâches. Au sein d'un même objet, l'approche ACCORD propose deux mécanismes pour spécifier les contraintes de concurrence entre les invocations, assurant ainsi la cohérence de l'état de l'objet :

- **Contraintes d'états :** elles spécifient quelles sont les méthodes autorisées pour un état donné. L'utilisateur modélise ces contraintes par une machine à états. Elle définit les enchaînements valides d'opérations susceptibles de s'exécuter au sein de l'objet et pour chaque état, les messages qui seront rejetés ou ignorés.
- **Contraintes d'accès :** elles spécifient si les méthodes de l'objet peuvent s'exécuter concurremment ou non. À l'étape de la modélisation, les méthodes de l'objet sont annotées pour spécifier leurs accès en *lecture* ou en *écriture* relativement aux attributs de l'objet. Le protocole de gestion de ces accès est 1 écrivain ou N lecteurs exclusivement. Les méthodes indépendantes de l'état de l'objet peuvent être annotées *parallèles*, elles sont donc susceptibles de s'exécuter en concurrence avec n'importe quelles autres méthodes de l'objet.

Spécifications temps réel. Dans le modèle d'objet actif supporté par l'approche ACCORD, les contraintes temps réel sont attachées aux messages échangés entre objets. Les communications entre deux objets s'apparentent à des invocations d'opérations de type client/serveur, mais les requêtes de services entre un client et un serveur se font par échange de message. C'est donc le client qui spécifie les contraintes temps réel associées à son invocation. Le serveur tente alors d'assurer la compatibilité entre les contraintes demandées par le client et ses propres capacités, à savoir en terme de politiques d'accès aux ressources (la gestion de la concurrence intra-objet spécifiée ci-dessus) et en terme de politique d'ordonnancement global (concurrence inter-objets).

Les contraintes attachées aux messages peuvent être des contraintes de temps, comme par exemple une date à laquelle doit commencer l'invocation, une échéance ou encore une périodicité (dans ce dernier cas, il est également possible de spécifier une condition d'arrêt), ou encore des contraintes liées à l'importance de l'invocation pour préciser plus finement comment le traitement de l'invocation doit être géré lorsque le système est en surcharge. Pour plus d'informations sur ce sujet, on peut se reporter à [Terrier *et al.*, 1993].

Modèle de communication. Les objets actifs qui composent une application ACCORD communiquent par échanges de messages. Dans ce contexte, tous les traitements d'un objet actif fournissant un ensemble de méthodes sont déclenchés par la réception des messages (synchronisation inter-objets).
Les différents schémas de synchronisation qui sont supportés par le modèle de l'objet actif sont ceux que l'on retrouve classiquement (synchrone, asynchrone, rendez-vous, etc) et que nous avons déjà présentés page 60, relativement aux descripteurs de liaisons pour les interfaces de service TINAP.
C'est à l'étape de la modélisation que l'utilisateur spécifie quelles politiques de synchronisation sont utilisées pour les invocations entre RTO.

Les objets temps réel peuvent aussi communiquer par échanges de signaux (il s'agit alors d'un mécanisme de notification asynchrone). Un signal reçu en entrée d'un objet actif est toujours associé à une opération fournie par l'objet. Le signal est donc utilisé pour déclencher l'exécution d'une opération. Bien entendu, une telle opération attachée à un signal ne doit pas définir de paramètres de sortie ni de valeur de retour.

Modèle d'exécution ACCORD, résumé. L'aspect multi-tâche d'une application ACCORD intervient donc (conceptuellement) à deux niveaux de granularité : (1) au niveau objet temps réel, utilisé comme un moniteur multi-tâche orienté service (2) à un niveau global puisqu'une application ACCORD est structurée par un ensemble d'objet temps réel. Pour résumer, on peut donc considérer qu'essentiellement trois mécanismes permettent de contraindre l'ordre d'exécution des tâches d'une telle application :

1. les contraintes d'accès spécifiées sur les méthodes fournies de l'objet actif,
2. les contraintes d'états définies à l'aide d'une machine à états qui spécifie le comportement propre à un objet actif,
3. et les annotations temps réel associées aux messages utilisées principalement pour définir des échéances sur les exécutions.

La portée des contraintes d'accès et d'états est locale à un objet temps réel donné, celle des annotations temps réel est globale à l'application. En effet, lorsque les contraintes locales sont satisfaites pour une invocation de service entrante donnée, la tâche associée à ce service est alors placée dans un plan d'ordonnancement global à l'application. A ce stade, l'ensemble des tâches de l'application concourent pour obtenir le processeur, l'ordonnancement est alors assuré par le système d'exploitation selon l'algorithme EDF (*Earliest Deadline First*).

Il est important de préciser que la machine à états associée à un objet temps réel n'est pas un simple contrat (abstrait) de comportement. En effet, l'exécutif implantant le modèle d'exécution ACCORD doit s'assurer que les transistions spécifiées par l'utilisateur sont scrupuleusement respectées à l'exécution. Nous pouvons donc considérer que ce modèle repose sur deux niveaux hiérarchiques d'ordonnancement des tâches : en premier lieu un ordonnancement lié à la logique métier (puisque les contraintes d'états et d'accès correspondent à la « sémantique métier » spécifiée par l'utilisateur) propre à chaque objet temps réel. En second lieu, un ordonnancement global lié aux contraintes temps réel (les échéances).
Il faut noter que l'ensemble de ces concepts, mécanismes et capacités d'expression offerts à l'utilisateur pour spécifier une application conforme au modèle d'exécution ACCORD peuvent conduire à des incohérences entre les deux niveaux. De ce fait, une spécification ACCORD peut s'avérer invalide d'un point de vue sémantique et le modèle proposé n'est pas suffisamment détaillé pour prendre en compte certaines incompatibilités. Cependant, cet aspect n'est pas le sujet de ce document.

6.3.1.2 Implantation de référence du modèle d'exécution ACCORD

Une explication détaillée de la structure de l'exécutif ACCORD est au-delà du cadre de ce document. Nous nous contenterons donc d'expliciter seulement certains points qui nous semblent pertinents et donnerons un exemple simplifié d'une séquence de traitement associé à une invocation de service entre deux objets temps réel.

Structure de l'implantation de référence. L'implantation de référence du modèle ACCORD a été developpée par le LLSP. Elle est écrite en C++ et est composée principalement de deux bibliothèques : le *noyau Accord* et une couche de virtualisation.
Le noyau fournit l'implantation des concepts principaux du modèle : l'objet temps réel associé au traitement des messages, la gestion « multi-threadée » des invocations de services, la politique de gestion locale à l'objet des contraintes de concurrence ainsi que l'ordonnancement global utilisant l'algorithme EDF (ou par priorité). Une extension de ce noyau a été également développée

pour prendre en compte la répartition d'une application ACCORD sur différents nœuds avec la gestion des communications que cela implique.
La couche de virtualisation fournit simplement au noyau ACCORD une abstraction du système d'exploitation sous-jacent pour simplifier la portabilité. Actuellement, cette couche est portée sous VXWORKS et LINUX.

Au sein de cette implantation, à chaque message échangé par l'application est associé un thread de contrôle et un thread d'exécution. Le premier exécute les traitements liés à la gestion des contraintes de concurrence et d'ordonnancement global. Le thread d'exécution est attaché à la méthode implantée par l'utilisateur mais reste bloqué par le thread de contrôle tant que les contraintes liées à la requête n'ont pas été vérifiées. Une priorité plus importante est attachée aux threads de contrôle par rapport aux threads d'exécution de telle sorte que les traitements qui leurs sont attachés soient prioritaires.

Exemple de traitement de message. Considérons un cas d'invocation de service entre un objet client `O1` et un objet serveur `O2`. `O1` demande l'exécution de la méthode `m()` sur `O2` avec une échéance notée `e`. L'ensemble des traitements associés au déroulement de cette invocation seront alors les suivants (représentés figure 6.9) :

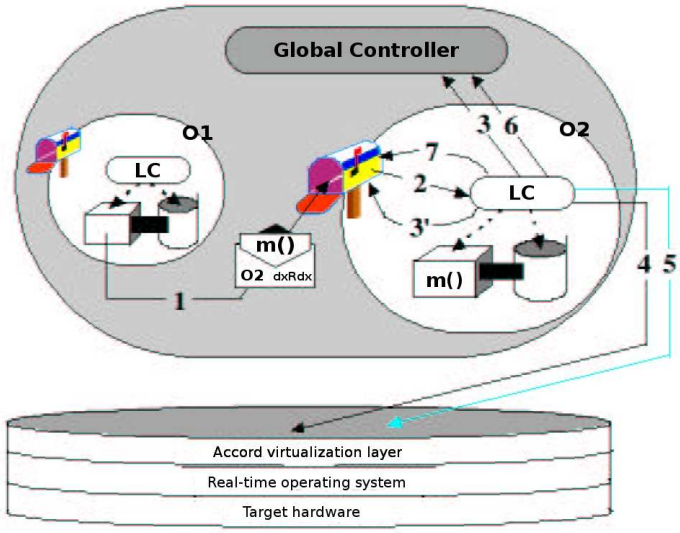


Figure 6.9 – Ensemble des traitements associés à un message, `O1` envoit `m()` à `O2`.

1. `O1` envoie le message `m()` à `O2`, celui-ci est alors estampillé avec l'échéance `e`. Un thread de contrôle est alors attaché à ce message et est lié à un descripteur contenant l'ensemble des caractéristiques du messages (les paramètres de la méthode, les contraintes temps réel, la référence du client, etc).
2. L'exécution de ce thread de contrôle préempte les exécutions éventuelles des threads d'exécution en cours (un plus haut niveau de priorité lui est associé) et lance les appels nécessaires pour vérifier les contraintes de concurrence et d'état de l'objet `O2`.
3. Deux situations peuvent alors se produire :
   - Les contraintes ne sont pas vérifiées, le message ne peut alors être traité. Celui-ci est alors stocké dans la boîte aux lettres de l'objet `O2` dans l'attente d'un traitement ulté-

rieur (le thread de contrôle se place alors en attente d'une notification qui sera levée lorsqu'un changement d'état de l'objet se produira).

– Les contraintes sont vérifiées et le traitement associé au message entrant est lancé. Le thread de contrôle ajoute alors le message dans le plan d'ordonnancement global (ordonné par les échéances).

4. Si l'échéance associée au message est la plus courte de la liste d'ordonnancement global, un thread d'exécution est attaché à la méthode `m()` qui peut démarrer son exécution.
5. Le thread de contrôle attend alors la fin de l'exécution du thread d'exécution. Si l'échéance associée au message est dépassée, une faute temporelle est levée.
6. Lorsque le traitement est terminé, le thread de contrôle valide le nouveau changement d'état interne de l'objet `O2` et débloque éventuellement un thread de contrôle en attente dans le plan d'ordonnancement global et/ou un thread de contrôle en attente dans la boîte aux lettres de l'objet.
7. Le thread de contrôle se met finalement en attente d'autres requêtes à traiter.

Nous pouvons noter que le modèle d'exécution proposé n'utilise pas de mécanisme de contrôle centralisé, c'est-à-dire qu'il n'y a pas de thread gérant le plan d'ordonnancement global, mais l'insertion/retrait d'une demande de service dans ce plan d'ordonnancement (implanté sous forme de liste partagée) provoque le déblocage du thread suivant de proche en proche. Pour une explication plus détaillée de cette implantation de l'exécutif ACCORD, et notamment au-dessus de LINUX, on peut se reporter à [Servat *et al.*, 2003].

6.3.2 « TINAPisation » du modèle d'exécution ACCORD

L'objectif de cette étude de cas consiste à utiliser les concepts énoncés par TINAP pour expérimenter une mise en œuvre du modèle d'exécution ACCORD.

Dans un premier temps, il s'agit de proposer un descripteur de contrôle simplifié permettant de prendre en compte les aspects de l'approche ACCORD qui ne sont pas nativement proposés dans notre démarche. C'est l'objectif de la prochaine sous-section.
Nous nous intéressons ensuite à décrire les aspects les plus importants d'une implantation des « membranes ACCORD ».

6.3.2.1 Niveau applicatif

Au niveau applicatif, le « *Real Time Object* » (RTO) du modèle ACCORD se concrétise par un simple composant TINAP : il fournit et requiert des interfaces et s'implante comme nous l'avons présenté dans le chapitre 4 (page 65).
Cependant, pour caractériser sa personnalité, il sera attaché à un « **descripteur de contrôle ACCORD** ». Il s'agit d'un simple descripteur[70] de type `multi-actif`, qui spécifie donc une borne maximale du nombre de tâches du système d'exploitation susceptibles d'exécuter ses méthodes métier. En outre, il permet de spécifier les contraintes de concurrences énoncées dans la section précédente :

– une simple machine à états pour spécifier les contraintes d'états du composant. Les transitions sont alors étiquetées par les points d'interaction atomique des interfaces fournies par le composant.

– des annotations sur ces points d'interaction pour spécifier les contraintes d'accès (*lecture*, *écriture* ou *parallèle*).

[70] Le méta-modèle correspondant à ce descripteur est donné en annexe A.2.

Conceptuellement, nous considérons, pour un tel composant, que toutes ses séquences implantées seront des *`interfaces actives`*, c'est-à-dire dont le code sera exécuté exclusivement dans le contexte d'une tâche gérée par son infrastructure.
Les spécifications des modes de communication et des contraintes temporelles se font telles qu'elles sont définies nativement par TINAP et que nous avons présentées respectivement dans le chapitre 4[71].

6.3.2.2 Niveau infrastructure

Par rapport aux implantations des descripteurs d'activité élémentaires TINAP que nous avons présentés dans le chapitre 5, une implantation d'un exécutif ACCORD se caractérise par les points suivants, qu'il est nécessaire de prendre en charge :

- L'ordonnancement des tâches se fait en-ligne, en fonction des échéances spécifiées par le concepteur, et selon l'algorithme EDF.
- Un mécanisme, en-ligne également, doit être mis en œuvre pour s'assurer du non dépassement des échéances, et déclencher un traitement le cas échéant (dans cette expérimentation, nous n'avons pas implanté cet aspect).
- Les exécutions des tâches d'un « composant ACCORD » sont fonction des contraintes d'états et d'accès spécifiées par le concepteur.

Au sein de l'infrastructure, ils s'implantent au niveau :

- des membranes générées autour des « composants ACCORD »,
- des liaisons à destination des « composants ACCORD », plus précisément au niveau des intercepteurs de sortie côté client,
- et du système d'exploitation au sein duquel doivent être implantés les services élémentaires nécessaires.

Pour faciliter la compréhension de l'infrastructure d'exécution qui implante la logique non fonctionnelle du modèle ACCORD, nous nous basons sur l'exemple donné par la figure 6.10.

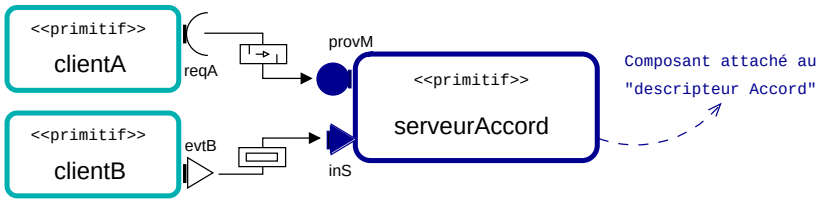


Figure 6.10 – Exemple d'architecture TINAP de niveau applicatif dont un composant est attaché à un « descripteur ACCORD ».

Deux simples composants actifs sont liés à un composant auquel est attaché un descripteur de contrôle ACCORD. Le concepteur spécifie les protocoles de communication pour ces liaisons, le protocole « synchrone » pour la liaison de service entre les interfaces `reqA` et `provM` et « l'asynchrone » pour la liaison d'événement entre les interfaces `evtB` et `inS`.
Les autres informations que le concepteur doit spécifier et qui ne sont pas représentées graphiquement :

- les échéances temporelles associées aux points d'interaction atomique de `reqA` et de `evtB`,

[71] Dans le contexte de notre expérimentation, nous n'avons pas considéré les possibilités de spécifier des dates de réveil et des périodes d'activation sur les « messages ACCORD ».

– les informations relatives au descripteur de contrôle du composant `serveurAccord` spécifiant :
  – les contraintes d'états et d'accès des points d'interaction atomique des interfaces d'entrée `provM` et `inS`.
  – le nombre de tâches du système d'exploitation qui seront attachées au composant à l'exécution (3 par exemple).

Dans les sections qui suivent, nous présentons succintement l'implantation du modèle d'exécution ACCORD au sein d'une infrastructure TINAP.

Implantation côté client

Du côté client (ou interfaces de sortie), la mise en œuvre de la logique d'une liaison à destination d'un composant ACCORD se caractérise par un simple intercepteur de sortie, schématisé par la figure 6.11 (dans le cas du composant `clientA`).

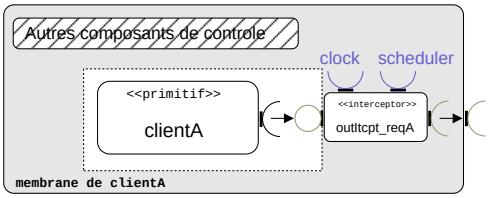


Figure 6.11 – Intercepteur côté client.

Il s'agit d'un intercepteur avec fonctionnalité qui prend en charge les aspects non fonctionnels de la liaison avec le composant destination. **Il est configuré avec les échéances spécifiées par le concepteur au niveau applicatif**. À chaque invocation en provenance du métier, il se charge d'initier la structure de données qui sera interprétée par le serveur : notamment en y ajoutant une estampille temporelle permettant d'associer une date machine absolue au démarrage de l'invocation (interface OS `clock`) et en renseignant l'identifiant de la tâche du système d'exploitation appelante (interface `scheduler`).
Il s'agit d'un *`intercepteur de type asymétrique`* : son interface externe est typée par une signature uniquement utilisée par les composants de contrôle générés, et utilisant des structures de données prenant en charge les aspects non fonctionnels de la liaison. Ces structures de données permettent de sérialiser l'ensemble des informations nécessaires à l'exécution de la requête par le « composant ACCORD » destinataire.
La liaison entre le composant `clientB` et le `serveurAccord` s'implante de manière similaire.

Implantation côté serveur : « membrane ACCORD »

La « membrane ACCORD » générée pour le composant métier `serveurAccord` est schématisée figure 6.12.

L'interception des interfaces d'entrée métier du composant `serveurAccord` s'implante tel que nous l'avons présenté dans le chapitre 5.

Le composant `RtcLocalController` est un composant mono-actif qui se charge de traiter les invocations de messages entrants et de lancer leur exécution si l'état courant du composant métier `serveurAccord` le permet (cet état est géré par le `RtcConstraints`). Si une invocation n'est pas autorisée à s'exécuter, le contrôleur se met en attente d'un changement d'état sur son interface d'entrée d'événement `unblock`.
Il s'agit d'un composant de contrôle générique, c'est-à-dire que sa structure et son implantation ne sont pas fonction des descripteurs de contrôle spécifiés par le concepteur au niveau applicatif.

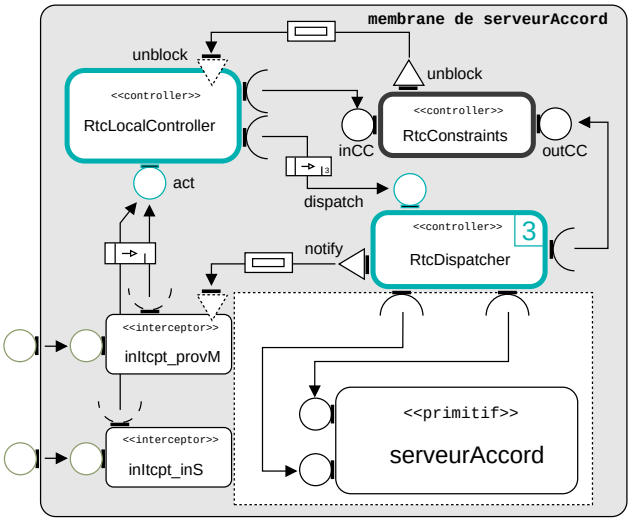


Figure 6.12 – Structure de la membrane de contrôle associée au composant métier `serveurAccord`.

Au contraire, le composant `RtcDispatcher` est généré en fonction des spécifications métier. Il s'agit d'un composant multi-actif, borné à `3` (correspondant à la valeur de la borne spécifiée par le concepteur au niveau applicatif) dont la fonction est d'exécuter le code implanté par le métier en fonction des demandes initiées par le `RtcLocalController` sur son interface d'activation. Lorsque ces exécutions sont terminées, il se charge de changer l'état courant du composant métier auprès du `RtcConstraints`, et éventuellement de débloquer la tâche cliente en attente de la terminaison du traitement (par l'interface événement `notify` qui implante la logique « synchrone » de la liaison métier entre `reqA` et `provM` spécifiée figure 6.10).

Le composant `RtcConstraints` est un composant généré en fonction des contraintes d'états et d'accès spécifiées par le concepteur (c'est au sein de ce composant que sont « intégrées » les informations fournies par le concepteur au niveau applicatif). Son interface `inCC` permet au `RtcLocalController` de tester si une requête est autorisée ou non à s'exécuter en fonction de l'état courant du composant métier. Son interface `outCC` permet de modifier cet état lorsqu'une invocation termine son exécution. Il s'agit d'un composant activement partagé et est donc protégé afin de sérialiser ses accès.

À l'exécution, une unique tâche du système d'exploitation sera attachée au `RtcLocalController`[72] et trois tâches au `RtcDispatcher`, cependant :

- l'exécution du contrôleur `RtcLocalController` doit être prioritaire par rapport aux exécutions des tâches qui exécutent le métier. La génération de l'infrastructure doit donc s'en assurer.
- comme nous l'avons précisé, les « tâches métier » sont ordonnancées en fonction des échéances attachées aux invocations.

[72] Il s'agit d'une « implantation centralisée » du contrôle, contrairement à l'implantation de référence du modèle d'exécution ACCORD.

La figure 6.13 schématise un extrait de la structure de la membrane du composant multi-actif `RtcDispatcher`. Ce composant est attaché à trois tâches du système d'exploitation, son `MultipleActivationController` fournit donc trois points d'entrée.
Contrairement aux contrôleurs d'activation que nous avons présentés dans le chapitre 5, qui requièrent une interface vers l'ordonnanceur à priorité de l'OS, celui-ci requiert une interface vers un ordonnanceur EDF. L'API définie permet de demander les activations des tâches en spécifiant une échéance absolue en paramètre.
Ainsi, l'ensemble des tâches destinées à exécuter le métier des différents « composants ACCORD » d'une même application seront ordonnancées globalement en fonction de leurs échéances.

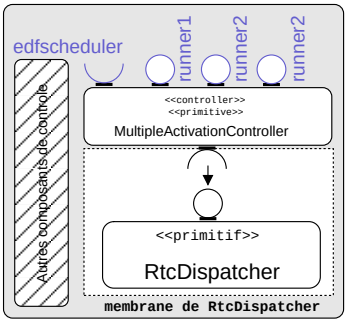


Figure 6.13 – Extrait de membrane de `RtcDispatcher`.

6.3.2.3 Niveau système d'exploitation

Pour nos expérimentations – comme nous l'avons énoncé dans la section 5.6 (page 104) – les services fournis par le système d'exploitation sont encapsulés au sein d'un composite. Pour le support du modèle d'exécution ACCORD, nous avons donc implanté un prototype d'ordonnanceur EDF, lui-même basé sur l'ordonnanceur à priorité implanté au sein de KORTEX (la bibliothèque de composant fournie avec THINK).
L'ensemble des composants `RtcDispatcher` de l'application sont alors liés à l'ordonnanceur EDF ; celui des composants `RtcLocalController` sont liés à l'ordonnanceur à priorité. Les priorités attachées à ces derniers composants actifs de contrôle sont supérieures à celles utilisées par l'ordonnanceur EDF de telle sorte à ce que leurs exécutions soient toujours prioritaires par rapport aux tâches qui exécutent le code applicatif.

6.3.3 Discussions et évaluations

Niveau applicatif. Considérant l'espace de conception de « niveau applicatif », un point intéressant à relever concerne la distinction possible entre une approche « purement » orientée modèle et celle de TINAP. En effet, dans notre approche, nous promouvons une « conception orientée programmation » pour spécifier le comportement des primitifs. Nous nous intéressons ensuite à construire de manière automatique une représentation abstraite de cette implantation et d'être en mesure de raisonner sur ces abstractions. Dans une telle démarche, que l'on peut qualifier de « *bottom-up* », les implantations et leurs abstractions sont de facto synchronisées. Cependant, l'approche proposée par la méthodologie ACCORD se veut « *top-down* » mais se fait alors en deux étapes : il est d'abord demandé au concepteur de modéliser son application avec les abstractions proposées puis d'implanter le comportement dans un langage de programmation sous-jacent. Il s'agit donc d'un effort supplémentaire demandé au concepteur, et sans être en mesure d'assurer l'isomorphisme entre l'implantation et les spécifications de haut niveau.

Toujours en considérant le niveau applicatif, cette expérimentation permet de spécifier des applications ACCORD non plus sous forme de configuration d'objets mais sous forme d'assemblages de composants. D'un point de vue conceptuel, il s'agit donc de bénéficier des apports du paradigme composants pour de telles applications, notamment en promouvant une approche constructiviste (et considérant les possibilités de la composition hiérarchique). À l'exécution, il s'agit également de bénéficier des capacités d'introspection et de reconfiguration dynamique qui ne sont pas considérées dans l'implantation de référence.

Séparation des préoccupations. En premier lieu, l'exercice mené par cette expérimentation consiste en une « componentisation » du modèle d'exécution ACCORD, il s'agit d'en proposer une séparation des préoccupations de fine granularité. Cette étape de conception n'est bien entendu pas triviale et dépend fortement des propriétés que le concepteur décide de favoriser (compréhension de l'architecture, réutilisation des composants, découplage par les données, minimisation des interactions par rapport aux traitements internes, etc).
Comme nous l'avons évoqué dans le chapitre 5, la séparation des préoccupations s'opère également entre les traitements du modèle d'exécution (au sein des contrôleurs) et les implantations des communications entre les composants métier (au sein des intercepteurs).
La séparation des préoccupations proposée dans la « membrane ACCORD » (présentée figure 6.12) tend à maximiser la modularisation des traitements et des données. Cependant, les composants de contrôle sont tout de même fortement couplés. Il s'agit d'un aspect inhérent aux algorithmes nécessaires à la mise œuvre du modèle d'exécution : la sémantique globale du contrôle se base sur un fort entrelacement des traitements et des données. Ce couplage fort est d'autant plus identifiable lorsque l'on considère le cas du dépassement d'une échéance (que nous n'avons pas implanté dans le contexte de cette expérimentation). En fonction de la manière dont une telle situation doit être traitée à l'exécution, sa mise en œuvre peut s'avérer complexe puisqu'il s'agit d'un état global partagé par l'ensemble des composants de contrôle côté client et côté serveur.
Il faut également constater que les spécifications de la vue dynamique de TINAP (notamment concernant le choix des entités actives ou protégées) influencent la séparation des préoccupations : les aspects fonctionnels et dynamiques ne peuvent effectivement être purement orthogonaux.

Compréhension de l'infrastructure. L'architecture de l'infrastructure d'exécution qui résulte de cette séparation des préoccupations facilite notablement la compréhension de la mise en œuvre du modèle d'exécution ACCORD. En effet, les informations compilées par la vue dynamique de niveau infrastructure, couplée à celles de la vue comportementale qu'il est possible d'obtenir, constituent une base d'informations pertinente et ce, à un niveau de granularité fin.

Membrane componentisée. Les algorithmes qui mettent en œuvre le modèle d'exécution ACCORD ne sont pas triviaux et l'architecture de granularité fine que nous avons proposée implique de nombreuses interactions et partages d'états entre les différents composants de contrôle. Cette étude de cas nous permet donc de penser que d'un point de vue conceptuel, l'utilisation du paradigme composant à ce niveau d'abstraction ne s'avère pas un obstacle et qu'il pourrait permettre l'implantation de modèle d'exécution ou de services d'ordre non fonctionnels a priori arbitrairement complexes.

Performances. Notre expérimentation manque malheureusement d'une comparaison de performance avec l'implantation ACCORD de référence. Elle n'a pu être menée en raison de la nature « trop prototypique » de nos implantations. Mais même de manière très informelle, et sans entrer dans les détails, il est vraisemblable qu'une version « componentisée » soit moins performante qu'une version monolithique (sans considérer les aspects d'optimisation qu'il est bien entendu possible d'envisager et que nous avons évoqué section 5.5).
En outre, l'implantation que nous avons expérimentée ayant été réalisée *from scratch*, une comparaison minutieuse devrait être effectuée car certains choix de conception demeurent assez éloignés entre les deux approches (notamment concernant la « gestion centralisée du contrôle » TINAP contrairement à celle de l'implantation de référence).

Couverture des concepts TINAP. Les concepts proposés au sein de notre approche, aussi bien au niveau applicatif qu'au niveau de l'infrastructure d'exécution, ont permis de mener à bien cette expérimentation de la mise en œuvre du modèle de conception et d'exécution ACCORD. Au

niveau applicatif, il s'agit alors de proposer un descripteur de contrôle spécifique basé sur les entités de première classe de notre langage structurel. Au niveau de l'infrastructure TINAP, les concepts proposés sont suffisamment riches pour implanter les mécanismes du modèle d'exécution ACCORD. De plus, la volonté d'architecturer ce niveau de fonctionnalités également sous forme d'assemblages de composants ne constitue pas un obstacle à sa mise en œuvre.

6.4 Conclusion sur les expérimentations

Dans ce chapitre, nous avons présenté deux études de cas de telle sorte à évaluer notre approche. Dans un premier temps, nous nous sommes intéressé à la conception d'une application de contrôle de flux multimédias. L'objectif était d'expérimenter les concepts TINAP de niveau applicatif.
Selon nous, cette étude renforce l'intérêt à proposer des mécanismes spécifiques de domaine audessus de l'architecture logicielle fonctionnelle (comme c'est le cas de TINAP où nous considérons la possibilité d'attacher des descripteurs de contrôle à partir des entités structurelles de première classe) et de les mettre en œuvre au sein de l'infrastructure d'exécution sans intervention du concepteur final. Celui-ci se concentre alors exclusivement sur ses aspects métier et la conception applicative s'en trouve facilitée.

Dans un deuxième temps, nous nous sommes intéressé à adapter l'espace de conception proposé par ACCORD au sein de notre démarche orientée composant. L'objectif était surtout d'expérimenter l'implantation des membranes nécessaires au support de son exécution. Cette étude montre que la volonté d'expérimenter le paradigme composant à différents niveaux d'abstraction d'un système ne semble pas un obstacle. D'autres expérimentations doivent cependant être menées pour s'en assurer, et notamment concernant une mesure précise de l'impact sur les performances.

Chapitre 7

Conclusion générale

Cette thèse propose un cadre et un canevas pour la conception d'applications embarquées basés sur le paradigme composant – nommé TINAP.
Il s'agit de proposer un espace de conception aux concepts canoniques que l'on peut séparer en deux aspects essentiels.
Premièrement, nous considérons que l'architecture fonctionnelle doit constituer l'« épicentre » du cycle de développement du logiciel. La sémantique globale d'un logiciel ainsi architecturé s'établit en terme de blocs de comportement et de données (les composants) qui sont interconnectés (les liaisons) par l'intermédiaire de leurs interfaces. Nous nous basons sur ces principales abstractions pour ensuite spécifier les aspects propres au domaine d'application considéré. Les sémantiques des composants sont alors personnalisées et différents mécanismes de communication de haut niveau sont proposés. Dans cette thèse, nous nous sommes essentiellement focalisé sur la définition d'un simple modèle de concurrence permettant de caractériser les aspects opérationnels de l'architecture fonctionnelle, et protocoles d'interactions entre les composants. Cette démarche est issue de la volonté de découpler les aspects d'ordre non fonctionnels de l'architecture métier et de proposer au concepteur de les exploiter directement au sein du langage de conception, et ce, sans se soucier de leur mise en œuvre. En ce sens, il s'agit de définir ce que nous pourrions qualifier de « langage de conception de haut niveau orienté architecture ».
Deuxièmement, nous nous sommes intéressé à représenter des abstractions de l'implantation interne des composants et de leur comportement relativement aux interfaces de communication qu'ils exportent. Il s'agit d'externaliser l'ensemble des informations internes nécessaires pour raisonner sur certaines propriétés du système, et ce, sans avoir à « descendre » au niveau du langage de programmation sous-jacent. Les plus fines abstractions sur lesquelles nous nous basons deviennent par exemple les blocs séquentiels d'instructions et les accès aux variables partagées. L'ensemble de ces informations constitue ce que nous pourrions qualifier d'« interface riche » pour le composant.

Cet « espace de conception canonique » a été défini dans le but d'expérimenter le paradigme composant pour développer l'ensemble du logiciel s'exécutant au-dessus de la plate-forme matérielle. Il ne s'agit pas de se limiter à l'espace de l'applicatif, mais d'être à même de concevoir l'infrastructure d'exécution implantant les mécanismes de haut niveau de celui-ci, ainsi que les services élémentaires fournis par le système d'exploitation, et ce également par assemblages de composants. Il s'agit donc de manipuler cet ensemble de concepts commun, et de l'enrichir en fonction des besoins, pour développer les fonctionnalités de ces différents niveaux d'abstraction.

Le langage TINAP pour concevoir l'applicatif propose différentes personnalisations des entités de première classe, notamment pour caractériser les modèles d'interaction attachés aux liaisons,

pour spécifier le caractère opérationnel attaché aux composants, ou pour caractériser certaines propriétés d'ordre non fonctionnel (telle que les échéances temporelles). De plus, il offre une certaine souplesse au concepteur, le modèle de programmation qui découle des personnalités qui lui sont proposées n'est que faiblement contraint. En contrepartie, l'objectif est de vérifier la cohérence globale de l'assemblage des composants de manière automatisée, et ainsi de l'assister durant la phase de développement.
Cet espace de conception a été expérimenté pour implanter une application de contrôle de flux multimédias, et nous a permis d'évaluer l'intérêt des concepts de haut niveau et des patrons de conception proposés par notre approche.

C'est au sein de l'infrastructure d'exécution que sont implantées les personnalités de haut niveau des liaisons et des composants applicatifs. Il s'agit d'exploiter le patron de « conception orientée membrane » (issu des implantations FRACTAL) pour mettre en œuvre ces aspects, par assemblages de composants dits « de contrôle ». Puis de générer un support d'exécution dédié à chaque entité métier, en fonction de leurs personnalisations et de leurs propriétés non fonctionnelles.
Ce mécanisme a été exploité pour implanter les abstractions applicatives de TINAP, et les résultats nous poussent à considérer la faisabilité et l'intérêt de « l'ingénierie orientée composant » au niveau de l'infrastructure d'exécution. De plus, il peut être étendu pour supporter des modèles de conception plus spécifiques. Nous en avons fait l'expérience pour implanter le modèle d'exécution de la méthodologie ACCORD, dont la sémantique globale de l'application se conçoit par un ensemble de moniteurs d'objets multi-tâches, interagissant par messages et contraints par échéances temporelles. Cette étude de cas laisse à penser qu'il s'agit d'une démarche satisfaisante pour la mise en œuvre d'infrastructures de complexité a priori arbitraire.

Enfin, la mise en œuvre des services fournis par le système d'exploitation se conçoit de manière similaire. À ce niveau d'abstraction, nous nous basons sur THINK et sa bibliothèque de composants, que nous avons enrichie pour prendre en compte les besoins spécifiques à notre proposition.

La possibilité offerte par TINAP d'exploiter le paradigme composant pour la conception d'un système, des plus hauts au plus bas niveaux d'abstraction, offre de nombreux intérêts. Tout d'abord, il s'agit de bénéficier à tous ces niveaux d'une structuration accrue du logiciel, qui classiquement, facilite sensiblement sa compréhension, sa maintenabilité et les opportunités de réutilisation. De plus, la jonction entre ces différents niveaux d'abstraction se concrétise également par l'intermédiaire d'interfaces spécifiques et de leurs liaisons correspondantes. Il s'agit alors de bénéficier d'une approche constructiviste, non seulement « horizontalement » (c'est-à-dire pour assembler des composants à un même niveau d'abstraction) mais aussi « verticalement » pour réifier au niveau de l'architecture des dépendances avec des fonctionnalités fournies par les couches sous-jacentes. Cette approche permet d'assurer une certaine « navigabilité » – ou traçabilité – entre ces niveaux, offrant ainsi un mécanisme pour assurer le raffinement de la perspective fonctionnelle du logiciel. La spécification d'un « noyau » illustre également la volonté d'utiliser ses concepts canoniques (notamment au travers de ce que nous avons qualifié d'« interfaces riches ») de manière homogène à différents niveaux d'abstraction.

De plus, comme nous l'avons évoqué, une particularité de ce modèle commun consiste en un découplage entre la spécification des aspects fonctionnels des aspects opérationnels et dynamiques, ce qui permet de raisonner selon deux échelles de préoccupation. D'un côté, l'architecture des préoccupations métier, dont les assemblages caractérisent la sémantique globale des fonctionnalités du logiciel, de l'autre, l'architecture des préoccupations liées aux aspects opérationnels. Ces derniers fournissent le caractère multi-tâche et concurrent de ce logiciel, la protection des ressources partagées, les modèles d'interaction et de synchronisation entre ces entités opérationnelles ainsi que les contraintes temporelles qui leurs sont attachées. TINAP offre ainsi la possibilité de raisonner sur ces deux aspects transverses.

D'abord en phase de conception amont, au sein de laquelle de nombreuses propriétés peuvent être vérifiées a priori, relativement à l'assemblage des fonctionnalités métier. Mais il en est de même par rapport aux aspects dynamiques pour lesquels certaines caractéristiques inhérentes à l'assemblage des entités opérationnelles peuvent être vérifiées. C'est notamment à ce stade que notre volonté d'encadrer le flot de conception de TINAP au sein d'une démarche outillée prend son sens. En effet, les supports méthodologiques, issus de l'ingénierie dirigée par les modèles, deviennent incontournables pour aider le concepteur de manière (semi) automatique lorsqu'il s'agit de proposer des langages de haut niveau, et donc dont l'architecturation des concepts et patrons proposés est soumise à de nombreuses contraintes, ceux-ci étant sémantiquement riches et spécifiques ou interdépendants. Une attention particulière a notamment été portée pour assurer une projection de la perspective dynamique d'une spécification TINAP – prenant en compte l'ensemble des informations de haut niveau ainsi caractérisées – vers une représentation fondée sur des abstractions plus basiques, proches de celles utilisées dans le cadre d'analyses formelles et automatisées.
Ensuite en phases de génération de l'infrastructure et de compilation pour laquelle la concrétisation des points sus-cités permet de contrôler précisément, à la granularité de ces abstractions près, des paramètres non fonctionnels qui les caractérisent et des besoins en terme de reconfiguration, le processus aboutissant à l'exécutif final – processus qu'il est capital de contrôler dans le domaine de l'embarqué.
Enfin, à l'étape de l'exécution. En effet, nous sommes convaincu qu'il est nécessaire de réduire l'écart entre les abstractions utilisées à l'étape de conception du logiciel et celles caractérisées à l'exécution, de telle sorte à établir un continuum clair entre ces deux niveaux d'abstraction. Dans un contexte où les applications deviennent de plus en plus complexes, il est important que les concepts manipulés en amont trouvent facilement leur réalisation à l'exécution, favorisant les étapes de mise au point, de dimensionnement, de simulation, de correction, de maintenance du logiciel en cours de développement et dans une moindre mesure en terme d'adaptabilité pour les applicatifs embarqués les moins figés. Cette volonté passe donc par la nécessité de définir de nouvelles infrastructures adaptées, ainsi que de nouvelles abstractions permettant d'automatiser les processus d'analyses et de production d'exécutifs performants.

Résumé de la contribution

Les principales contributions de cette thèse peuvent se résumer par les points suivants :
Nous avons spécifié un modèle de composant (que nous avons qualifié de « noyau ») qui définit la manière dont doit être structuré le logiciel et quelles abstractions du code interne il est nécessaire d'externaliser pour raisonner sur certaines de ses propriétés. À partir de ce modèle et d'extensions appropriées, il est possible de concevoir et de raisonner sur l'ensemble d'un système, à savoir au niveau de l'applicatif, de son infrastructure d'exécution basée sur le « paradigme de conception orienté membrane », jusqu'aux services élémentaires du système d'exploitation. Ces extensions concernent principalement la spécification d'un modèle de concurrence qui permet d'expliciter les aspects multi-tâches et leurs contraintes de synchronisation et d'ordonnancement, les protocoles de communication entre composants, et la manière dont les différents niveaux de fonctionnalités s'assemblent dans une démarche constructiviste.
Les prototypes d'implantation de TINAP et les études de cas présentés dans ce document ont été réalisés avec THINK (à partir de sa chaîne d'outils et de la bibliothèque de composants qui lui est associée). Dans ce cadre, différentes membranes de contrôle ont été implantées et certains composants de la bibliothèque KORTEX ont été enrichis pour évaluer notre démarche.
Une attention particulière a été portée pour ancrer notre approche au sein d'un cadre de conception « dirigé par les modèles ». TINAP a été ainsi défini par un ensemble de méta-modèles, qui ont permis d'implanter un prototype d'éditeur au sein d'ECLIPSE prenant en compte les spécificités et contraintes de ce langage de haut niveau.

Limites et perspectives à court terme

Bien entendu, le travail effectué dans le contexte de cette thèse n'est pas complet. Plusieurs travaux restent à mener pour couvrir l'ensemble de la démarche que nous cherchons à promouvoir.

Tout d'abord, la plupart des outils réalisés dans le contexte de ce travail, aussi bien à l'étape de conception qu'à l'étape de la compilation, ne sont qu'à un stade de prototypes, leur utilisation est bien entendu pour le moment très limitée.

Nos expérimentations menées avec TINAP manquent de *benchmarks* d'analyse de performances, et il s'agit d'une limitation significative quant à considérer la démarche dans son ensemble. Il est indéniable que l'approche que nous promouvons engendre un coût important sur les performances (et l'empreinte mémoire). À terme, le flot d'exécution de TINAP ne peut être découplé d'une prise en charge fine de ces aspects, notamment en supportant différents niveaux d'optimisations, d'abord pour minimiser les coûts de la réification de l'architecture à l'exécution, ensuite relativement aux concepts TINAP de haut niveau dont les implantations peuvent être considérablement optimisées, et enfin à un niveau plus global du système. Cependant, de nombreuses optimisations peuvent être considérées de manière assez générique, notamment grâce à la capacité de raisonner avec les mêmes abstractions à différents niveaux (par exemple, la mise en œuvre de fusion de composants en un code monolithique). C'est notamment un point capital qui nous a poussé à expérimenter cette voie, de telle sorte à être en mesure de raisonner finement sur l'ensemble des fonctionnalités du système.

Comme nous l'avons évoqué, nous nous sommes intéressé à la projection d'une spécification TINAP vers une « représentation bas niveau », proche des abstractions utilisées par les outils d'analyse formelle. Cependant, nous ne sommes pas allé jusqu'à une expérimentation concrète avec un tel outil, qui nous aurait permis de tester la pertinence de cette représentation, notamment dans sa transformation vers le formalisme d'entrée de celui-ci, mais aussi relativement à l'exhaustivité des paramètres nécessaires. Enfin, le point le plus important consiste à vérifier si certaines constructions qu'il est possible de spécifier avec TINAP n'aboutissent pas à une combinatoire impossible à analyser dans certaines conditions.

En dehors d'un sous-ensemble, il n'a pas été défini la manière dont les concepts de haut niveau de TINAP peuvent se superposer aux spécifications FRACTAL en terme de capacité d'introspection, de reconfiguration et d'instanciation.

Perspectives à plus long terme

De telle sorte à s'affranchir des limites décrites ci-dessus et relativement aux résultats obtenus dans le contexte de cette thèse, différentes perspectives sont envisageables :

À l'exécution

Configurabilité dynamiques des concepts de TINAP. Relativement à la limite exposée précédemment concernant le rapprochement entre les spécifications FRACTAL et les concepts de haut niveau proposés par TINAP, nous pouvons poser la question du niveau de configurabilité de ces concepts que nous serions susceptibles de proposer à l'exécution. La question se pose, non pas seulement pour les aspects structuraux de TINAP, mais également pour ses aspects dynamiques. Est-il possible par exemple de changer dynamiquement le protocole d'interaction d'une liaison ? Offre-t-on la capacité de changer dynamiquement la personnalité d'un composant, par exemple pour passer d'un composant passif à un composant actif ? La mise en œuvre de certaines de ces possibilités ne sont pas triviales, et peuvent avoir un coût élevé en terme de performance s'il s'agit de les proposer dans le cas général. Et bien entendu, la prise en charge de cette reconfigurabilité, relativement à ces aspects dynamiques, est fonction des besoins du métier ciblé.

Définition d'une API conforme aux concepts TINAP. Il serait intéressant de proposer une API permettant de réifier les concepts TINAP à l'exécution (au même titre que les spécifications FRACTAL définissent une API pour introspecter, reconfigurer et instancier les composants). Il s'agirait de considérer les aspects structurels (notamment les liaisons de flot de données et d'événements) mais aussi les aspects liés au modèle de concurrence proposé. Cette difficulté est issue du fait que ces aspects ne peuvent être réifiés par des objets de première classe à l'exécution. Par exemple, une liaison de flot de donnée audio caractérise un mode d'interaction de haut niveau faisant intervenir différents composants à l'exécution, si l'on se place selon le point de vue de l'infrastructure d'exécution. Cependant, il devrait être possible de proposer à l'application en cours d'exécution, et donc selon « son point de vue applicatif », de demander l'établissement d'une telle liaison en une opération atomique. La définition d'une telle API ne pose pas de problème en soi, mais sa mise en œuvre, étant donné que l'infrastructure d'exécution est elle-même implantée par des assemblages de composants, nécessite de faire cohabiter différents niveaux d'API selon la perspective selon laquelle on se place. Il s'agirait de donner différentes représentations de l'état du système.
Cette perspective s'accompagne notamment d'une réflexion à avoir sur les propriétés ou personnalités définies à l'étape de conception qui doivent être rendues concrètes (ou non) à l'exécution, sous forme de méta-données. Par exemple, doit-on réifier à l'exécution la différences entre une interface métier, de contrôle, d'OS, de déroutement ou interne/externe pour les intercepteurs, etc ?

Génération de l'infrastructure

Réduction des coûts induits par la démarche. Comme nous l'avons évoqué dans les perspectives à court terme, il est nécessaire d'appliquer certaines optimisations au processus que nous proposons pour l'obtention de l'exécutif final, et qui peuvent s'effectuer à différentes échelles. D'abord concernant la réification de la structure à l'exécution, qu'il peut s'avérer inutile de préserver lorsque le système a été fiabilisé durant les phases de développement, et que la reconfiguration dynamique n'est pas nécessaire. Dans ce cas, il s'agit de mettre en œuvre la fusion des composants pour en obtenir un code monolithique (fusion du métier et des fonctionnalités implantées par les membranes dans le cas des mises en œuvre des concepts de TINAP). Un éventail de possibilités peut être considéré quant à la prise en charge de cette fusion : totale avec ou sans préservation de méta-données permettant la regénération de l'architecture, partielle en ne considérant de fusionner que certaines parties explicitement localisées de l'architecture (comme c'est actuellement le cas au sein de la chaîne NUPTSE, notamment concernant les liaisons), ou encore en considérant tous les niveaux d'abstraction jusqu'au système d'exploitation. De nombreuses optimisations peuvent être également considérées concernant les implantations des concepts TINAP de haut niveau.

Support de différentes infrastructures d'exécution. Nous nous sommes intéressé à exploiter le paradigme composant pour implanter le support d'exécution de TINAP et l'avons expérimenté pour le modèle d'éxécution de l'approche ACCORD. Il serait intéressant de mener cette expérience avec d'autres modèles de conception, par exemple pour implanter le support d'une spécification CLARA , SAVECCM , ou encore AUTOSAR (ce-dernier étant plus ambitieux, étant donné la richesse des spécifications du RTE[73] et du contrôle que celui-ci doit exercer sur l'applicatif). Il s'agirait alors de valider notre démarche à plus large échelle, et d'en évaluer la capacité à ainsi prototyper rapidement d'autres modèles d'exécution. Plus généralement, il serait également intéressant de considérer la cohabitation de ces différents modèles d'exécution, donc de manière hétérogène, au sein d'une démarche de conception commune.

[73] Le RTE AUTOSAR est succintement présenté dans l'état de l'art, page 22.

Vers un support outillé pour la description des membranes. À terme, il serait intéressant de développer le support pour pouvoir proposer facilement au sein de la chaîne d'outils la mise en œuvre de nouvelles membranes componentisées. Il s'agirait de manipuler les personnalités d'« intercepteurs » et de « contrôleurs » en tant qu'entité de première classe, ainsi que les interfaces de déroutement que nous avons définies pour caractériser précisément la manière dont les éventuelles indirections doivent s'opérer entre les occurrences d'entrées/sorties et les contrôleurs. Cependant, d'autres informations doivent être renseignées, notamment concernant les composants dont les implantations doivent être générées en fonction d'informations spécifiées par le concepteur.
Idéalement, il serait intéressant de travailler sur la possibilité de découpler différents aspects pris en compte au sein des membranes, par exemple la description de l'implantation des interactions entre les composants métier, découplée de la description de l'implantation des contrôleurs d'activation. Bien entendu, il s'agit d'un aspect complexe à mettre en œuvre dans le cas général, la problématique étant celle du tissage résultant de ces deux descriptions, non obligatoirement indépendantes.

Conception

Interfaçage avec des outils d'analyse et MARTE. Dans notre proposition, l'ensemble des informations qui permettent de modéliser un système TINAP, c'est-à-dire celles capturées au sein des quatre vues que nous avons définies, se projettent en une représentation basée sur des abstractions plus basiques, réduisant ainsi l'écart sémantique entre notre démarche de conception de haut niveau et les formalismes utilisés par les outils d'analyse formelle. À ce titre, une expérience à mener consiste à exploiter le canevas d'analyse proposé dans le contexte de MARTE[74] [OMG, 2007]. L'approche consiste à établir une représentation pivot du logiciel, alors exploitable vers différents outils d'analyse d'ordonnançabilité, de performance ou encore d'estimation de pires temps d'exécution, académiques ou issus de l'industrie. L'objectif serait alors d'assurer le lien entre notre représentation d'une spécification TINAP « applatie » et la représentation d'entrée du canevas d'analyse MARTE. De plus, il s'agirait d'expérimenter la manière dont les informations disponibles en sortie de ces processus d'analyse pourraient être exploitées au sein des spécifications de haut niveau TINAP, et ainsi influencer en retour la conception du système.
Il serait également intéressant de confronter l'espace de conception proposé par TINAP avec celui standardisé dans le contexte de MARTE (en ce qui concerne le niveau applicatif).

Encadrement de l'espace de reconfiguration. Il serait intéressant de proposer en amont de l'étape de conception, une prise en charge « encadrée » des possibilités de reconfigurations architecturales du système à l'exécution. Une perspective pourrait être d'utiliser FSCRIPT [David, 2005], « *un langage dédié pour la reconfiguration dynamique consistante de composants* FRACTAL » et de l'adapter aux concepts structuraux de TINAP (notamment les interfaces de flots et leurs liaisons correspondantes, pouvant être attachées à une interaction de type *broadcast*). L'objectif de ce langage serait alors de caractériser hors-ligne, l'ensemble des reconfigurations possibles à l'exécution. Ces informations pourraient notamment permettre d'analyser certaines propriétés, par exemple pour garantir le respect d'échéances temporelles en fonction des différents états reconfigurable du système, ou de chercher à établir les pires temps d'exécution des transitions entre deux états stables lors d'une reconfiguration effective.

Possibilité d'extension. Il serait intéressant de voir comment de manière générale le langage TINAP pourrait être étendu. Les études de cas que nous avons proposées dans le contexte de

[74] MARTE est un standard OMG pour le développement dirigé par les modèles des systèmes embarqués temps réel.

cette thèse présentent des formes d'extensions, par exemple avec les interfaces et les liaisons de « flot de données audio » dans le cas de DECKX, ou de la prise en charge d'une « personnalité ACCORD » pour les composants. Bien entendu, puisque nous cherchons à assurer le lien entre outils de modélisation et de génération/compilation, de telles extensions interviennent de manière complexe à différentes phases du flot de conception global, et touchent inéluctablement différents outils. Offrir concrètement au concepteur la possibilité de prendre en compte ses propres extensions, tout en minimisant son implication dans le flot de conception existant, et en cherchant à potentiellement détecter les conflits que ses extensions pourraient engendrer avec l'existant, constitue un problème difficile.
Au delà de cet aspect, de nombreuses extensions peuvent être considérées, pour par exemple proposer des personnalités de type « capteurs/actionneurs », permettant de rendre concret la manière dont les informations en provenance ou à destination de l'environnement peuvent interagir avec les composants de niveau applicatif, utiliser le même mécanisme que dans le cas de composants actifs mais pour spécifier des domaines de protection mémoire, etc.
D'un point de vue modélisation, il s'agit par exemple de proposer d'autres propriétés d'ordre non fonctionnel à externaliser sur les « interfaces riches », par exemple la concrétisation de l'empreinte mémoire nécessaire à un composant, de délais potentiels de transmission des informations sur une liaison entre deux composants, etc.

Bibliographie

[Amnell *et al.*, 2002] Tobias Amnell, Elena Fersman, Leonid Mokrushin, Paul Pettersson, and Wang Yi. Times - a tool for modelling and implementation of embedded systems. In Springer-Verlag, editor, *8th International Conference, TACAS 2002, part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2002*, pages 460–464, Grenoble, France, April 2002. Lecture Notes in Computer Science, Vol.2280. 84

[Baduel *et al.*, 2006] Laurent Baduel, Françoise Baude, Denis Caromel, Arnaud Contes, Fabrice Huet, Matthieu Morel, and Romain Quilici. *Grid Computing : Software Environments and Tools*, chapter Programming, Deploying, Composing, for the Grid. Springer-Verlag, January 2006. 31

[Barais, 2005] Olivier Barais. *Construire et Maîtriser l'Evolution d'une Architecture Logicielle à base de Composants*. PhD thesis, Université des Sciences et Technologies de Lille, nov 2005. 78

[Basu *et al.*, 2006] Ananda Basu, Marius Bozga, and Joseph Sifakis. Modeling heterogeneous real-time components in bip. In *SEFM '06 : Proceedings of the Fourth IEEE International Conference on Software Engineering and Formal Methods*, pages 3–12, Washington, DC, USA, 2006. IEEE Computer Society. 8

[Bengtsson *et al.*, 1995] J. Bengtsson, K. G. Larsen, F. Larsson, P. Pettersson, and W. Yi. Uppaal – a tool suite for automatic verification of real-time systems. In *Workshop on Verification and Control of Hybrid Systems*, October 1995. 84

[Berbers *et al.*, 2005] Y. Berbers, P. Rigole, Y. Vandewoude, and S. Van Baelen. CoConES : An approach for components and contracts in embedded systems. In *Component-Based Software Development for Embedded Systems : An Overview of Current Research Trends*, volume 3778 of *Lecture Notes in Computer Science*, pages 209–231. Springer-Verlag, September 2005. 8

[Beugnard *et al.*, 1999] A. Beugnard, J-M. Jézéquel, N. Plouzeau, and D. Watkins. Making components contract aware. *Computer*, 32(7) :38–45, July 1999. 7

[Bouyer, 2005] Patricia Bouyer. An introduction to timed automata. In *Actes École d'été ETR'05*, pages 79–94, 2005. 84

[Bouyssounouse and Sifakis, 2005] Bruno Bouyssounouse and Joseph Sifakis, editors. *The ARTIST Roadmap for Research and Development*, volume Vol. 3436. Lecture Notes in Computer Science – Springer, 2005. 6

[Bozga *et al.*, 1998] Marius Bozga, Conrado Daws, Oded Maler, Alfredo Olivero, Stavros Tripakis, and Sergio Yovine. Kronos : A model-checking tool for real-time systems. In *Formal Techniques in Real-Time and Fault-Tolerant Systems*, 1998. 84

[Bruneton *et al.*, 2004] E. Bruneton, T. Coupaye, M. Leclercq, V. Quéma, and J-B. Stefani. An open component model and its support in java. In Springer, editor, *Lecture Notes in Computer Science*, pages 7–22. Ivica Crnkovic, Judith A. Stafford, Heinz W. Schmidt, and Kurt C. Wallnau, 2004. 11

[Bruneton *et al.*, 2006] Eric Bruneton, Thierry Coupaye, Matthieu Leclercq, Vivien Quéma, and Jean-Bernard Stefani. The fractal component model and its support in java : Experiences with auto-adaptive and reconfigurable systems. *Software - Practice and Experience (SP&E)*, 36(11&dash ;12) :1257–1284, 2006. 9, 32

[Budinsky *et al.*, 2004] F. Budinsky, D. Steinberg, R. Ellersick, E. Merks, S.A. Brodsky, and T.J. Grose. *Eclipse Modeling Framework*. Addison-Wesley, 2004. 147

[Carlson *et al.*, 2006] Jan Carlson, John Haakansson, and Paul Pettersson. SaveCCM : An analysable component model for real-time systems. In Z. Liu and L. Barbosa, editors, *Proceedings of the 2nd Workshop on Formal Aspects of Components Software (FACS 2005)*, volume 160 of *Electronic Notes in Theoretical Computer Science*, pages 127–140. Elsevier, 2006. 18

[D. Isovic, 2002] C. Norstrm D. Isovic. Components in real-time systems. In *The 8th Int. Conf. On Real-Time Computing Systems and Applications (RTCSA'2002)*, 2002. 8

[David, 2005] Pierre-Charles David. *Développement de composants Fractal adaptatifs : un langage dédié à l'aspect d'adaptation*. PhD thesis, Ecole des Mines de Nantes, 2005. 26, 134

[de Oliveira, 2005] Jaime de Oliveira. Le dimensionnement temps réel dans l'automobile – étude de cas. In *Actes École d'Été Temps Réel (ETR'05)*, 2005. 24

[DeAntoni, 2007] Julien DeAntoni. *SAIA : un style architectural pour assurer l'indépendance vis-à-vis d'entrées / sorties soumises à des contraintes temporelles*. PhD thesis, Institut National des Sciences Appliquées de Lyon, 2007. 27

[DeMichiel and Keith, 2006] Linda DeMichiel and Michael Keith. Jsr 220 : Enterprise javabeans, version 3.0. Technical report, Sun Microsystems, 2006. 32

[Dijkstra, 1976] E.W. Dijkstra. *A Discipline of Programming*. Prentice-Hall, 1976. 6

[Durand, 1998] Emmanuel Durand. *Description et vérification d'architecture temps réel : CLARA et les réseaux de Pétri temporels*. PhD thesis, École Centrale de Nantes, Nantes, 1998. 8, 17

[Déplanche and Faucou, 2005] Anne-Marie Déplanche and Sébastien Faucou. Les langages de description d'architecture pour le temps réel. In *actes École dÉté Temps Réel (ETR'05)*, 2005. 8

[Déplanche, 2005] Anne-Marie Déplanche. Rapport de synthèse - Action Spécifique CNRS 195 - Composants et Architectures Temps Réel. Technical Report RI2005_1, IRCCyN, March 2005. 8

[Escoffier and Donsez, 2005] C. Escoffier and D. Donsez. FractNet : A Fractal implementation for .NET, September 2005. 11

[Espinoza, 2007] Huascar Espinoza. *An Integrated Model-Driven Framework for Specifying and Analyzing Non-Functional Properties of Real-Time Systems*. PhD thesis, Université d'Evry, 2007. 84

[Fassino *et al.*, 2002] J.-P. Fassino, J.-B. Stefani, J. Lawall, and G. Muller. Think : A software framework for component-based operating system kernels. In *Proceedings of the USENIX Annual Technical Conference*, pages 73–86, June 2002. 11

[Fassino, 2001] Jean-Philippe Fassino. *THINK : vers une architecture de systèmes flexibles*. PhD thesis, Ecole Nationale Supérieure Des Télécommunications, December 2001. 11, 91

[Faucou *et al.*, 2004] Sébastien Faucou, Anne-Marie Déplanche, and Yvon Trinquet. An adl centric approach for the formal design of real-time systems. In Springer, editor, *Architecture Description Language Workshop at IFIP World Computer Congress (WADL'04)*, Volume 176 of IFIP series, pages 67–82, Toulouse, France, August 2004. 17, 58

[Faucou, 2002] Sébastien Faucou. *Description et construction d'architectures opérationnelles validées temporellement*. PhD thesis, University of Nantes, France, 2002. Ph.D Thesis in Control and Applied Computer Science. 17, 18, 59

[Feiler *et al.*, 2004] P H. Feiler, B. Lewis, S. Vestal, and E. Colbert. An overview of the sae architecture & design language (aadl) standart : A basis for model-based architecture-driven embedded systems engineering. In *Architecture Description Language, workshop at IFIP World Computer Congress*, 2004. 19

[Fennel *et al.*, 2006] Helmut Fennel, Stephan Bunzel, and al. Achievements and exploitation of the autosar development partnership, 2006. 22

[França *et al.*, 2007] Ricardo Bedin França, Jean-Paul Bodeveix, Mamoun Filali, Jean-François Rolland, David Chemouil, and Dave Thomas. The aadl behaviour annex – experiments and roadmap. *Engineering Complex Computer Systems, 2007. 12th IEEE International Conference on*, pages 377–382, July 2007. 20

[Garavel, 2003] H. Garavel. Défense et illustration des algèbres de processus. In *Actes de l'Ecole d'été Temps Réel ETR 2003*, Toulouse, France, September 2003. In Zoubir Mammer. 7

[Garlan and Shaw, 1993] D. Garlan and M. Shaw. *An introduction to software architecture*. World Scientific Publishing, 1993. 6

[Garlan, 2000] David Garlan. Software architecture : a roadmap. In A. Finkelstein, editor, *The Future of Software Engineering*. ACM Press, 2000.

[GdR, 2006] Autosar GdR. Autosar software component template, june 2006. 23

[Group, 2006a] Object Management Group. Meta object facility (mof) core specification, 2006. 147

[Group, 2006b] Object Management Group. Xml meta interchange (xmi), 2006. 147

[Gérard *et al.*, 2002] S. Gérard, F. Terrier, and Y. Tanguy. The model paradigm working for real-time applications development : Accord/uml. In *OOIS02*, May 2002. 117

[Gérard *et al.*, 2005] Sébastien Gérard, Yann Tanguy, and David Servat. Introduction of rt-qos in component interfaces. Projet Families - WP 3 - Family Quality - Task 3.3 - Evolution, Adaptation and Maintenance Qualities, June 2005. 145

[Gérard, 2002] Sébastien Gérard. The accord/uml profile. Technical report, CEA-LIST, 2002. 117

[Hakansson, 2007] John Hakansson. The saveccm language reference manual. Technical report, Dept. of Information Technology Uppsala University, 2007. 19

[Hamid *et al.*, 2005] Irfan Hamid, Elie Najm, Laurent Pautet, and Thomas Vergnaud. Extending the aadl to provide flexible software components. White paper submitted to AADL standardization committee, July 2005. 24

[Hansson *et al.*, 2004] Hans Hansson, Mikael Åkerholm, Ivica Crnkovic, and Martin Törngren. Saveccm - a component model for safety-critical real-time systems. In *Euromicro Conference, Special Session Component Models for Dependable Systems*, Rennes, France, September 2004. IEEE. 8, 18

[Harbour *et al.*, 2001] M. González Harbour, J.J. Gutiérrez García, J.C. Palencia Gutiérrez, and J.M. Drake Moyano. Mast : Modeling and analysis suite for real time applications. In *13th Euromicro Conference on Real-Time Systems*, pages 125–134, Delft, The Netherlands, June 2001. IEEE Computer Society Press. 84

[Heinecke *et al.*, 2006] Harald Heinecke, Jürgen Bielefeld, and al. Autosar - current results and preparations for exploitation. In *7th EUROFORUM conference "Software in the vehicle"*, Stuttgart, Germany, May 2006. 22

[Henia *et al.*, 2005] R. Henia, A. Hamann, M. Jersak, R. Racu, K. Richter, and R. Ernst. System level performance analysis - the symta/s approach. In *Computers and Digital Techniques, IEE Proceedings*, volume 152, Issue 2, pages 148–166, Mar 2005. 84

[Hon, 1998] Honeywell Technology Center. *MetaH Users Manual - version 1.27*, 1998. 8, 19

[HUGUES *et al.*, 2007] Jerome HUGUES, Bechir ZALILA, Laurent PAUTET, and Fabrice KORDON. Rapid prototyping of distributed real-time embedded systems using the aadl and ocarina. *rsp*, 0 :106–112, 2007. 21

[Hugues, 2005] Jérôme Hugues. *Architecture et Services des Intergiciels Temps Réel*. PhD thesis, l'École Nationale Supérieure des Télécommunications, 2005. 27

[IEEE, 2000] IEEE. Ieee recommended practice for architectural description of software-intensive systems, 2000. IEEE Std 1471-2000. 36, 72

[ISTCompare, 2005] ISTCompare. IST FP6 - COMPARE document D2.3-1 - framework architecture and artefact, July 2005. 12

[Kristian Sandstrom, 2004] Mikael Akerholm Kristian Sandstrom, Johan Fredriksson. Introducing a component technology for safety critical embedded real-time systems. In *International Symposium on Component-based Software Engineering (CBSE7)*, 2004. 8

[Layaïda and Hagimont, 2005] O. Layaïda and D. Hagimont. Plasma : A component-based framework for building self-adaptive applications. In Springer, editor, *Symposium On Electronic Imaging, Conference on Embedded Multimedia Processing and Communications (SPIE/IS&T)*, San Jose, CA, USA, January 2005. Ivica Crnkovic and Judith A. Stafford and Heinz W. Schmidt and Kurt C. Wallnau. 11

[Leclercq *et al.*, 2007] Matthieu Leclercq, Ali Erdem Özcan, Vivien Quéma, and Jean-Bernard Stefani. Supporting heterogeneous architecture descriptions in an extensible toolset. In *ICSE '07 : Proceedings of the 29th International Conference on Software Engineering*, pages 209–219, Washington, DC, USA, 2007. IEEE Computer Society. 11

[Lipari and Bini, 2005] Giuseppe Lipari and Enrico Bini. A methodology for designing hierarchical scheduling systems. *J. Embedded Comput.*, 1(2) :257–269, 2005. 41

[Lobry and Polakovic, 2008] Olivier Lobry and Juraj Polakovic. Controlling the performance overhead of component-based systems. In *7th International Symposium on Software Composition (SC 2008)*, 2008. 11, 68

[Magee, 1999] J. Magee. Behavioral analysis of software architectures using ltsa. In *Proceedings of the 21st international conference on Software engineering*, pages 634–637. IEEE Computer Society Press, 1999. *ISBN : 1-58113-074-0*. 78

[Medvidovic and Taylor, 2000] Nenad Medvidovic and Richard N. Taylor. A classification and comparison framework for software architecture description language. In *IEEE Transactions on Software Engineering*, pages 26(1) :70–93, January 2000. 1, 6

[Merz, 2001] Stephan Merz. Model checking : A tutorial overview. In F. Cassez et al., editor, *Modeling and Verification of Parallel Processes*, pages 3–38. Springer-Verlag, 2001. 84

[Microsoft, 2002] Microsoft. .net – http ://www.microsoft.com/net, 2002. 32

[Müller *et al.*, 2001a] P. Müller, C. Stich, and C. Zeidler. Components @ work : Component technology for embedded systems. In *Proceedings of the Component-based Software Engineering Track at the 27th IEEE Euromicro Conference (Euromicro CBSE'01)*, September 2001. 14

[Müller *et al.*, 2001b] Peter Müller, Christian Stich, and Christian Zeidler. Components@work : Component technology for embedded systems. In *27th International Workshop on Component-Based Software Engineering, EUROMICRO*, 2001. 8

[Nierstrasz, 1987] O. Nierstrasz. Active objects in hybrid. In ACM SIGPLAN Notices, editor, *OOPLSA'87*, pages 243–253, 1987. 117

[OMG, 1999] OMG. *CORBA Component Model*, February 1999. Document ad/99-02-05. `www.omg.org`. 12

[OMG, 2002a] OMG. *Minimum CORBA specification*, 2002. Document formal/02-08-01. `www.omg.org`. 12

[OMG, 2002b] OMG. Object management group - corba components, v3.0 (full specification). Document formal/02-06-65, June 2002. 32

[OMG, 2003] OMG. *Real-time CORBA*, 2003. Document formal/03-11-01. `www.omg.org`. 14

[OMG, 2006a] OMG. Object management group - meta object facility (mof) core specifications, 2006. 36

[OMG, 2006b] OMG. Object management group - object constraint language (ocl), 2006. 37

[OMG, 2007] OMG. Object management group – a uml profile for marte, 2007. 84, 134

[OSEK, 2003] OSEK. Osek/vdx operating system specification, 2003. http ://www.osek-vdx.org. 66, 105

[Özcan, 2007] Ali Erdem Özcan. *Conception et Implantation d'un Environnement de Développement de Logiciels à Base de Composants – Applications aux Systèmes Multiprocesseurs sur Puce*. PhD thesis, Institut National Polytechnique de Grenoble, 2007. 11

[Parnas, 1972] D. L. Parnas. On the criteria to be used in decomposing systems into modules. *Communications of the ACM*, 15(12) :1053–1058, 1972. 1, 6

[Polakovic *et al.*, 2006] Juraj Polakovic, Ali Erdem Ozcan, and Jean-Bernard Stefani. Building reconfigurable component-based os with think. In *EUROMICRO '06 : Proceedings of the 32nd EUROMICRO Conference on Software Engineering and Advanced Applications*, pages 178–185, Washington, DC, USA, 2006. IEEE Computer Society. 27

[Project, 2004] EAST-EEA Project. Definition of language for automotive embedded electronic architecture. Version 1.02, 2004. 8

[Puaut, 2005] Isabelle Puaut. Méthodes de calcul de wcet (worst-case execution time) – etat de l'art. In *Actes École d'Été Temps Réel (ETR'05)*, Nancy, 2005. 84

[Quinn and Hannan, 2001] B. G. Quinn and E. J. Hannan. *The Estimation and Tracking of Frequency*. Cambridge University Press, 2001. 110

[Quéma, 2005] Vivien Quéma. *Vers l'exogiciel - Une approche de la construction d'infrastructures logicielles radicalement configurables*. PhD thesis, Institut National Polytechnique de Grenoble, December 2005. 27, 32

[rob, 2003] Robust open component based software architecture for configurable devices project. Technical report, 2003. 8

[Robert, 2002] Paul Robert. Le petit robert - dictionnaire de la langue française, 2002. 38

[SAE, 2004] SAE. *Architecture Analysis & Design Language (AADL)*. As-2 Embedded Computing Systems Committee SAE, nov 2004. SAE Standards AS5506. 8, 19

[(SAE), 2005] Society For Automotive Engineers (SAE). Aadl meta model and interchange formats, 2005. SAE AADL Meta Model/XMI V0.9. 21

[Seinturier *et al.*, 2006] L. Seinturier, N. Pessemier, L. Duchien, and T. Coupaye. A component model engineered with components and aspects. In *Proceedings of the 9th International SIGSOFT Symposium on Component-Based Software Engineering (CBSE'06)*, volume 4063 of *Lecture Notes in Computer Science*, pages 139–153. Springer, June 2006. 11, 27, 32

[Servat *et al.*, 2003] D. Servat, S. Gérard, A. Lanusse, P. Vanuxeem, and F. Terrier. Doing real-time with a simple linux kernel. In *Real-Time Linux Workshop*, 2003. 121

[Stankovic and Rajkumar, 2004] John A. Stankovic and R. Rajkumar. Real-time operating systems. *Real-Time Systems*, Volume 28 :237–253, 2004. 105

[Stankovic, 2001] J. Stankovic. Vest : A toolset for constructing and analyzing component based embedded systems. In *Springer-Verlag Lecture Notes*, pages 390–402, 2001. 8

[Szypersky *et al.*, 2002] C. Szypersky, D. Gruntz, and S. Murer. *Component Software. Beyong Object-Oriented Programming*. ACM Press, 2002. 1, 6, 40

[Terrier *et al.*, 1993] François Terrier, D. Bras, and Patrick Vanuxeem. Intellingent real-time control : Real-time scheduling design in object-oriented approach. In *OOPSLA'93*, 1993. 118

[Tessier *et al.*, 2003] P. Tessier, S. Gérard, C. Mraidha, F. Terrier, and J.M. Geib. A component-based methodology for embedded system prototyping. *Proceeding 14th IEEE International Workshop on Rapid System Prototyping (RSP'03)*, (ISBN : 0-7695-1943-1), June 2003. 117

[Tešanović *et al.*, 2004] Aleksandra Tešanović, Dag Nyström, Jörgen Hansson, and Christer Norström. Aspects and components in real-time system development : Towards reconfigurable and reusable software. *Journal of Embedded Computing*, 1(1), feb 2004. 8

[Tournier, 2005] Jean-Charles Tournier. A survey of configurable operating systems. Technical report, University of New Mexico, November 2005. 11

[Trialog, 1992] Trialog. Sceptre 2, rapport sur les services offerts aux applications temps réel à contraintes strictes par les exécutifs temps réel. Groupe Sceptre – Trialog Informatique, Juin 1992. 30

[van Ommering *et al.*, 2000] R. van Ommering, F. van der Linden, J. Kramer, and J. Magee. The koala component model for consumer electronics software. In *IEEE Computer*, volume 33, pages 78–85, 2000. 8

[Vergnaud, 2007] Thomas Vergnaud. *Modélisation des systèmes temps-réel répartis embarqués pour la génération automatique d'applications formellement vérifiées*. PhD thesis, Ecole Nationale Supérieure des Télécommunications, 2007. 27

[Vestal, 2005] Steve Vestal. Error model annex. AADL User Workshop, 2005. 20

[Wallnau, 2003] Kurt C. Wallnau. Volume iii : A technology for predictable assembly from certifiable components. Technical report, CMU/SEI-2003-TR-009, 2003. 8

[Wardle, 2006] Scott Wardle. Mspinkys vinyl tracking object - software specification version 0.0.5, August 2006. 109

Annexe A

Autres méta-modèles TINAP

L'objectif de cette annexe est de présenter les méta-modèles évoqués dans le document et qui nous semblait judicieux de reléguer en annexe.

A.1 Comportement TINAP de « niveau architecture »

Dans le chapitre 4 (page 77), nous avons présenté la manière avec laquelle se construit la représentation du comportement global d'une architecture, celle-ci étant définie par un assemblage de composants TINAP au sein de la vue dynamique. L'information obtenue suite à ce processus peut être sérialisée sous la forme du méta-modèle présenté figure A.1. Elle permet de « stocker le comportement » d'un assemblage TINAP sous forme transactionnelle, notamment en exprimant concrètement les liens de causalité entre activités, et/ou entre activités et l'environnement. Elles se définissent alors par un ensemble de « pas », eux-mêmes composés de « séquences d'activité globale » (caractérisées par l'`IOAutomataStep` du méta-modèle). Ces notions sont respectivement abordées pages 82 et 78. La partie gauche de la figure correspond à des méta-classes définies au sein des autres vues TINAP de telle sorte à assurer la traçabilité entre elles et le comportement global. La figure 4.41 (page 83) illustre une représentation graphique possible d'une instance de ce méta-modèle (on y retrouve donc les « méta-entités » qui correspondent, par exemple `SequentialExecutionStep`, `ProtectedActivationStep`, etc).

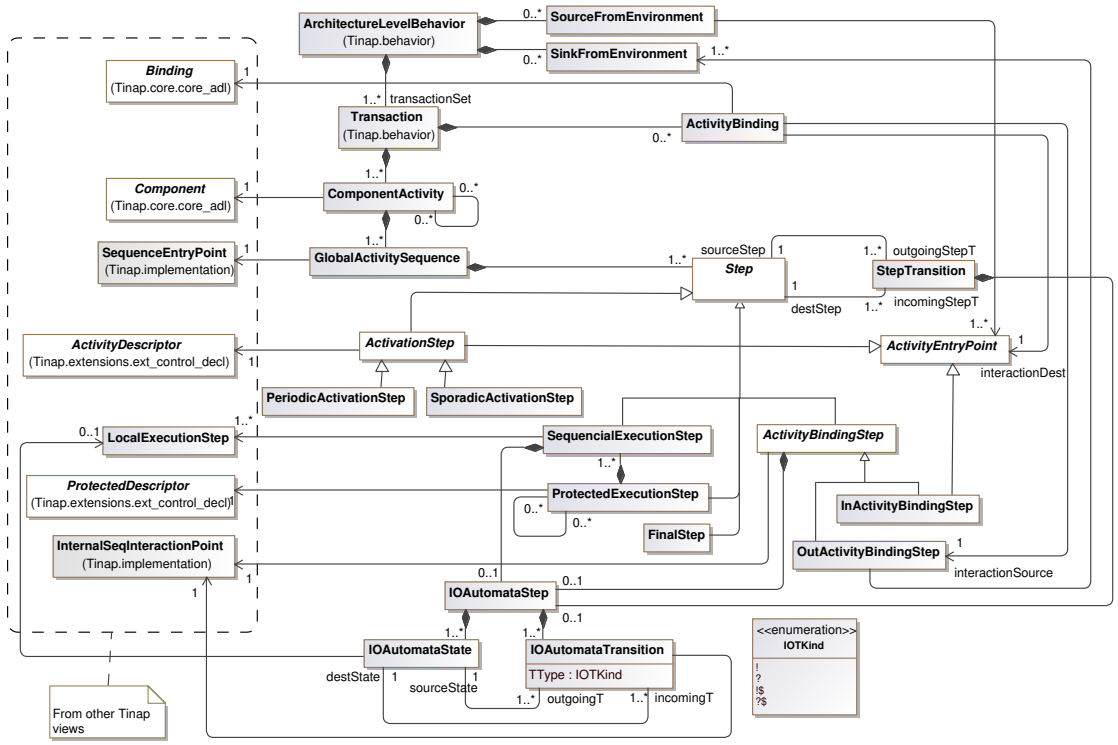


Figure A.1 – Méta-modèle pour sérialiser les informations relatives à la « vue comportement de niveau architecture » TINAP.

A.2 Descripteur de contrôle ACCORD

Dans la section 6.3.2, page 121, nous avons présenté une implantation du modèle d'exécution de l'approche ACCORD (détaillée page 117). Au niveau applicatif de TINAP, un « composant ACCORD » peut se spécifier simplement avec un descripteur de contrôle dont le méta-modèle est donné figure A.2[75].

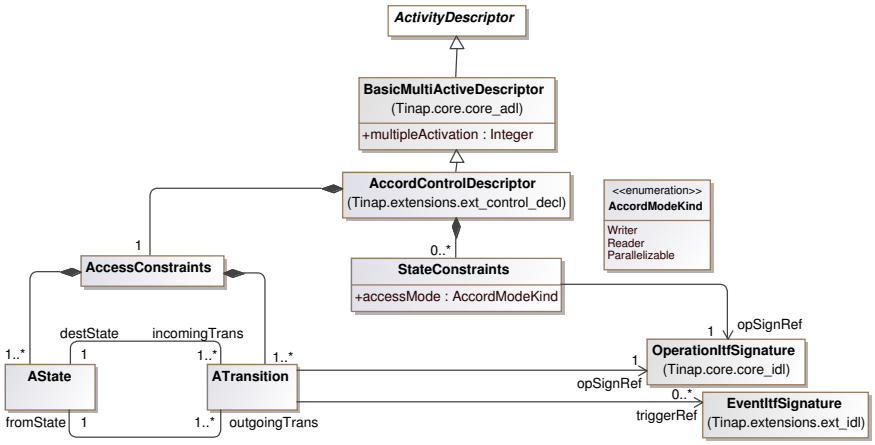


Figure A.2 – Descripteur de contrôle TINAP de niveau applicatif pour la spécification d'un « composant ACCORD ».

Il s'agit d'un descripteur de contrôle de type « multi-actif » (défini page 54). Il permet simplement de spécifier les contraintes d'états du composant (méta-classe `AccessConstraints`, une simple machine à états dont les transitions sont étiquetées par l'opération d'une interface fournie du composant et éventuellement par un événement d'une interface d'entrée d'événement qui lui est attachée) et les contraintes d'accès (`StateConstraints`) pour exprimer le mode d'accès d'une opération relativement à l'état interne du composant.
Bien entendu, l'utilisation d'un tel descripteur implique de nombreuses contraintes qu'il faut respecter pour définir une spécification cohérente et qui ne sont pas capturées au sein de la structure du méta-modèle. Pour exemple, il s'agit de s'assurer que toutes les références vers les signatures d'opérations ou d'événements sont effectivement définies pour les « interfaces d'entrées » attachées à ce composant, ou que les interfaces d'entrée d'évenements ne soient pas des « interfaces implantées » (en effet, nous considérons dans ce cas que les événements ne peuvent être spécifiés que dans l'unique but de déclencher l'exécution d'une opération fournie par le composant, tel que cela est spécifié par la méthodologie ACCORD). Il est du ressort des outils de modélisation de s'assurer du respect de ces contraintes d'utilisation du « descripteur ACCORD ».

[75] Au sein de la méthodologie ACCORD, l'implantation de ces concepts a été proposée en utilisant le mécanisme d'extension d'UML2, pour de plus amples informations, on peut se reporter à [Gérard *et al.*, 2005].

Annexe B

Intégration ECLIPSE

L'objectif de cette annexe est de donner une vue d'ensemble des outils TINAP d'aide à la conception qui ont été prototypés dans le contexte de cette thèse.

Intégration ECLIPSE. Les méta-modèles des concepts TINAP que nous avons définis nous ont permis d'implanter un prototype d'éditeur au sein de la plate-forme ECLIPSE. Pour cela, nous avons utilisé EMF [Budinsky *et al.*, 2004]. Il s'agit d'un framework de modélisation et de génération de code qui permet de faciliter la conception d'outils et d'applications basés sur des modèles de données structurées (les méta-modèles). EMF peut être assimilé à une implantation Java (nommée Ecore) d'un sous-ensemble des concepts définis dans le MOF (*Meta Object Facility* [Group, 2006a]) par l'OMG. À partir d'un méta-modèle Ecore, le framework fournit un générateur utilisé pour créer automatiquement l'ensemble des classes Java et une API et pour manipuler les instances du modèle conforme à ce méta-modèle. Ces instances peuvent alors être rendues persistantes grâce au support de la sérialisation XMI [Group, 2006b]. Une extension de ces fonctionnalités de base du framework EMF permet également le support de la génération automatique d'un éditeur basique de type arborescent (*tree view*) intégré dans la plate-forme ECLIPSE sous forme de plugin. À titre d'illustration, la figure B.1 présente une capture d'écran de l'éditeur de niveau applicatif. Il s'agit de la spécification de la vue structurelle de l'exemple de spécification TINAP donné figure 4.14 (page 48).

TINAP et FRACTAL-ADL. TINAP est basé sur le modèle structurel issu de FRACTAL et les outils de conception prototypés dans la plate-forme ECLIPSE sont implantés en tant que *front-end* de FRACTAL-ADL. Ce dernier définit le langage de description d'architecture utilisé pour spécifier une application FRACTAL sous la forme d'une syntaxe XML (dont la grammaire est spécifiée par une DTD). Grâce à ce langage, les concepts architecturaux du modèle FRACTAL, à savoir les définitions de composants (ainsi que les mécanismes d'extension associés), les composites, les liaisons peuvent être décrits.
Il est alors intéressant de faire une comparaison des principaux concepts définis au sein de TINAP par rapport à ceux initialement proposés par FRACTAL-ADL.

Dans notre approche, le langage est spécifié par un méta-modèle et à peu de détails près, les capacités d'expression de celui-ci au sein du paquetage `core_adl` (page 4.2.1.2) sont identiques à celles de FRACTAL-ADL. Cependant, nous avons fait le choix de préciser finement la syntaxe du langage contrairement à ce dernier, par exemple en proposant les méta-classes *ComponentType*, *PrimitiveComponent* et *CompositeComponent* (figure 4.3, page 39). Cette distinction entre nature de

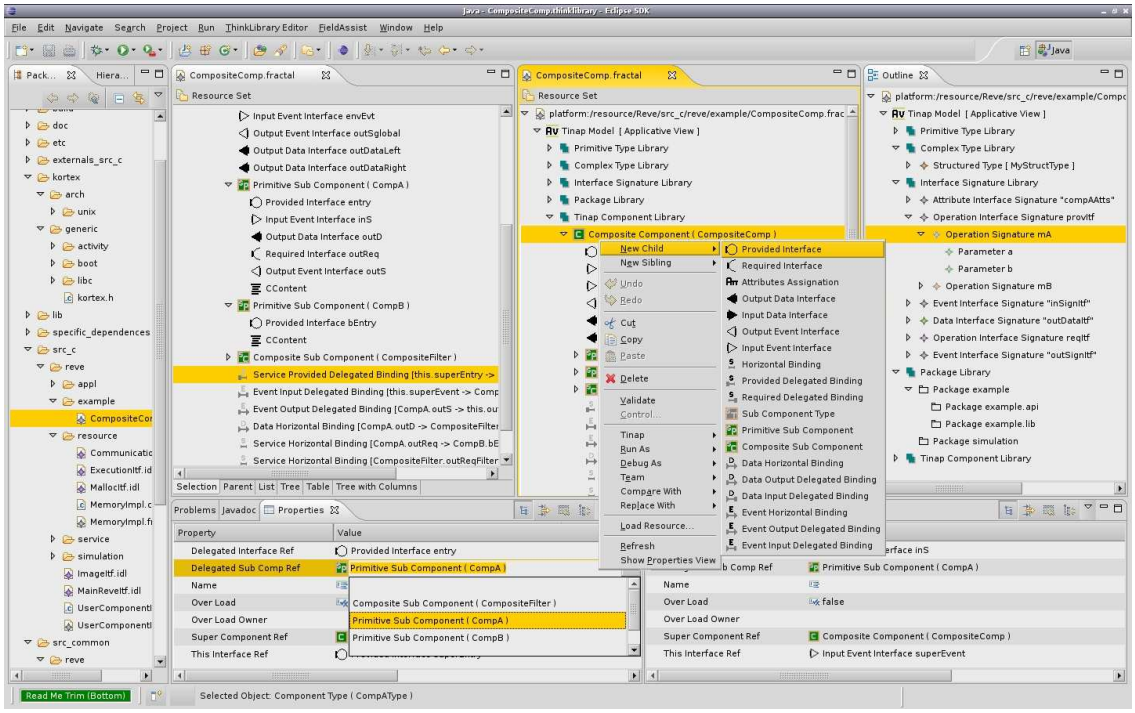


Figure B.1 – Superposition de captures d'écran de l'éditeur arborescent TINAP de niveau applicatif (vue structurelle).

composant au niveau de la syntaxe du langage était initialement issue de THINK V2. Mais la motivation essentielle de cette organisation des entités de première classe du langage réside dans le fait qu'il est possible de préciser finement certaines contraintes au sein de l'outil de modélisation que nous avons prototypé en fonction de leur typage. À titre d'illustration, la figure B.2 schématise sous forme de méta-modèle un extrait de la grammaire de l'ADL telle qu'elle est définie de manière standard par FRACTAL-ADL (à comparer aux extraits de méta-modèles des figures 4.3, 4.4, 4.8 et 4.11, données entre les pages 39 et 45).

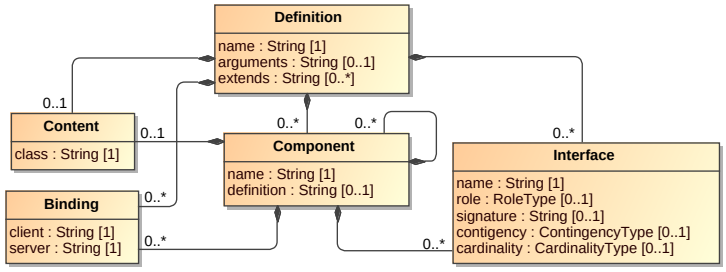


Figure B.2 – Représentation sous forme de méta-modèle des principales balises définies dans la DTD standard de FRACTAL-ADL

Si on se restreint aux aspects structurels définis dans le paquetage `core_adl`, les deux syntaxes sont très proches, une description architecturale spécifiée selon notre approche est totalement projetable en FRACTAL-ADL et réciproquement en appliquant les transformations adéquates. Bien entendu, ce n'est par contre pas le cas des interfaces (données et événements) et

liaisons spécifiées dans le paquetage `ext_adl` (cf. page 37) ainsi que les spécifications des descripteurs de contrôle qui demandent une extension du langage proposé par FRACTAL-ADL.

En terme d'outillage, plutôt qu'une édition brute en XML, l'utilisation de technologies autour de la méta-modélisation au sein de la plate-forme ECLIPSE permet de générer des éditeurs plus ergonomiques. L'intérêt de ce type d'éditeur est également d'offrir la possibilité d'implanter des mécanismes d'assistance au concepteur. En particulier, ces fonctionnalités lui permettent de spécifier en ligne des descriptions architecturales cohérentes du point de vue de la sémantique du langage, contrairement aux outils de FRACTAL-ADL qui nécessitent une passe de traitement pour vérifier la bonne cohérence des descriptions. En effet, dans notre approche, la gestion des contraintes du langage TINAP sont directement implantées à partir du code de l'éditeur généré par EMF.

Dans le contexte de nos travaux, nous n'avons pas cherché à prototyper des éditeurs graphiques au-dessus de notre méta-modèle TINAP. Une telle piste peut être explorée avec GMF[76], qui au même titre qu'EMF, permet la génération automatique d'une partie d'un éditeur graphique basé sur un méta-modèle ECORE. Une autre possibilité consisterait à spécialiser la plate-forme de modélisation UML PAPYRUS[77], à partir d'un profil UML de TINAP.

FRACTAL-ADL est accompagné d'un canevas modulaire dont l'architecture (également décrite et implantée par des composants FRACTAL en Java) permet d'intégrer relativement aisément des extensions de l'ADL élémentaire et d'implanter les fonctionnalités qui les prend en charge au sein de la chaîne de compilation existante. Cet aspect d'extensibilité est théoriquement possible au sein de notre démarche mais elle s'avère en pratique plus fastidieuse que dans le contexte de FRACTAL-ADL, essentiellement due à la diversité et à la complexité des outils et technologies utilisés pour la mettre en œuvre, il s'agit donc d'un aspect qui reste à étudier.

[76] *Graphical Modeling Framework*, `www.eclipse.org/gmf`.
[77] `www.papyrusuml.org`

Glossaire

Composant activement partagé : Un tel composant se caractérise par le fait que les méthodes qu'il implante sont susceptibles d'être exécutées dans différentes activités (ou contextes de tâches distincts), et donc de manière concurrente. Ses interfaces d'entrée sont alors liées à de multiples composants actifs. *page 81*

Descripteur de contrôle : De manière générale, il s'agit d'informations liées aux aspects non fonctionnels du logiciel qui sont concrètement spécifiées au niveau du langage de description d'architecture, manipulées par le concepteur de manière déclarative et dont l'implantation est prise en charge de manière automatique. *page 53*

Implantation mono-séquence : Il s'agit de l'implantation attachée à un composant primitif et qui présente la particularité de fournir un unique point d'entrée, typiquement l'implantation d'une unique méthode. À opposer à une « implantation multi-séquence ». *page 65*

Implantation multi-séquence : Implantation attachée à un composant primitif qui fournit de multiples points d'entrée et donc de multiples blocs de code séquentiels. À opposer à une « implantation mono-séquence ». *page 65*

Intercepteur asymétrique : Un intercepteur qui présente une interface externe de type différent que son interface interne. Un tel composant d'interception est généralement utilisé pour assurer la « glue » entre une interface fonctionnelle et un protocole d'intercation (ou connecteur) présentant une interface spécifique pour sa mise en œuvre. *page 91*

Intercepteur symétrique : Un intercepteur dont l'interface interne et l'interface externe sont strictement du même type. *page 91*

Interface d'activation : Interface d'entrée spécifique d'un « composant actif » dont une sollicitation depuis l'environnement engendre une demande d'activation de ce composant. À opposer à une « interface passive ». *page 55*

Interface d'interaction interne : Interface d'un composant spécifiant les interactions susceptibles de se produire entre le début et la fin de l'exécution d'un bloc de code séquentiel au composant. À opposer à une « interface implantée ». *page 47*

Interface de déroutement : Au niveau de l'infrastructure d'exécution, s'emploie pour caractériser le déroutement d'un fil d'exécution entrant ou sortant d'un composant fonctionnel vers un composant de contrôle. *page 91*

Interface externe : S'emploie au niveau de l'infrastructure d'exécution pour caractériser l'interface que présente un intercepteur à l'environnement. *page 91*

Interface implantée : Interface dont les entrées correspondent à des blocs de code fonctionnels implantés par le composant (primitif). *page 47*

Interface interne : S'emploie au niveau de l'infrastructure d'exécution (c'est-à-dire des membranes) pour caractériser l'interface que présente un intercepteur à son contenu (une implantation pour un primitif ou un sous-composant pour un composite). *page 91*

Interface multi-liée : interface d'un composant attachée à plusieurs liaisons. Une sémantique et des contraintes spécifiques sont alors définies pour l'interface en question. *page 44*

Interface passive : Interface d'entrée qui n'est pas attachée à une activité, au contraire d'une « interface d'activation ». Un « composant passif » ne présente que des interfaces passives, un « composant actif » peut éventuellement en définir. *page 55*

Liaison multiple : ce terme est utilisé pour caractériser des schémas d'interconnexions « complexes » entre liaisons, typiquement de type `1-N`, `N-1` et `N-M`. *page 44*

Occurrence d'événement : On parle ici d'événement au sens large du terme, à savoir une opération définie au niveau du langage de description d'architecture qui change l'état du système (appel de méthode entrant, production d'une donnée, etc). *page 52*

Point d'entrée de séquence : Relativement à l'implantation d'un composant primitif, on parle de point d'entrée pour qualifier un point de l'architecture pour lequel une méthode est implantée, dont le code est susceptible d'être traversé par un fil d'exécution. En d'autres termes, un point d'entrée est toujours attaché à une unique séquence. *page 65*

Point de synchronisation : Il s'agit d'un point de l'architecture (plus précisément un point d'interaction atomique donc attaché à un composant donné) sujet à de potentielles synchronisations avec l'environnement (changement de l'état de l'activité). *page 65*

Séquence étendue : D'un point de vue dynamique, il s'agit d'un bloc de code séquentiel implanté par un primitif invoquant un service susceptible de provoquer une attente (ou bloquage) du fil d'exécution qui le parcours. *page 66*

Séquence autiste : Bloc de code séquentiel implanté par un primitif qui n'interagit en aucune manière avec son environnement (ni par l'intermédiaire d'interface de sortie, ni par l'intermédiaire d'interface d'interaction interne). *page 66*

Séquence basique : D'un point de vue dynamique, il s'agit d'un bloc de code séquentiel implanté par un primitif qui n'invoque aucun service susceptible de provoquer une attente (ou bloquage) du fil d'exécution qui le parcours. *page 66*

Séquence d'activité globale : Ensemble des actions observables susceptibles de s'exécuter dans le contexte d'une activité et dont le fil d'exécution, initié par un composant actif est généralement transverse à de multiples composants passifs de l'architecture. *page 78*

Séquence locale : Ce terme définit une suite séquentielle d'instructions au sein de l'implantation d'un primitif. *page 65*

Index

AADL, 19
Accord, 96, 117
ADL, 6
Annotations, 49, 83
AOKell, 27, 32
Autosar, 22

CCM, 12
Clara, 17, 58
Compare, 12
Composant, 6, 39
    actif, 53, 54
    passif, 53
    protégé, 54, 61
Composite, 8, 40
Connecteur, 7, 95, 102
Contrôleur, 92

DeckX, 108

Eclipse, 147
EMF, 147
Encapsulation, 41, 63, 79

Fractal, 9, 50, 54, 68

IDL, 45
Implantation, 68
    mono/multi-séquence, 65
Intercepteur, 91
Interface, 7, 41
    TINAP, 41, 112
    audio, 112
    d'activation, 55, 62, 63
    d'interaction interne, 47, 55, 63, 65
    de déroutement, 91, 96
    implantée, 47
    interne/externe, 91
    multi-liée, 44, 47, 52
    nature, 41
    passive, 55, 63

Kortex, 11, 104, 125

Liaison, 7, 43, 48, 58

Marte, 84, 134
Mast, 84
Membrane, 31, 90
MetaH, 19
Modèle de concurence, 53

Nuptse, 11, 68

Pecos, 14

Séquence
    d'activité globale, 78
    locale, 65
SaveCCM, 18

Think, 11, 50, 68, 104, 125
Tinap, 29

Vues TINAP, 36
    comportement, 36, 72
    dynamique, 36, 52
    implantation, 36, 65
    structurelle, 36, 39

WCET, 83