# Chapitre 3 - VODEL, un langage de description d'architectures logicielles statiques et dynamiques

*«Examine soigneusement chaque voie. Essaye aussi souvent que tu le crois nécessaire. Puis pose toi la seule question : cette voie a t'elle une âme ? Si oui, c'est une bonne voie, sinon elle ne sert à rien» (Carlos Castaneda dans Les enseignements de Don Juan)*

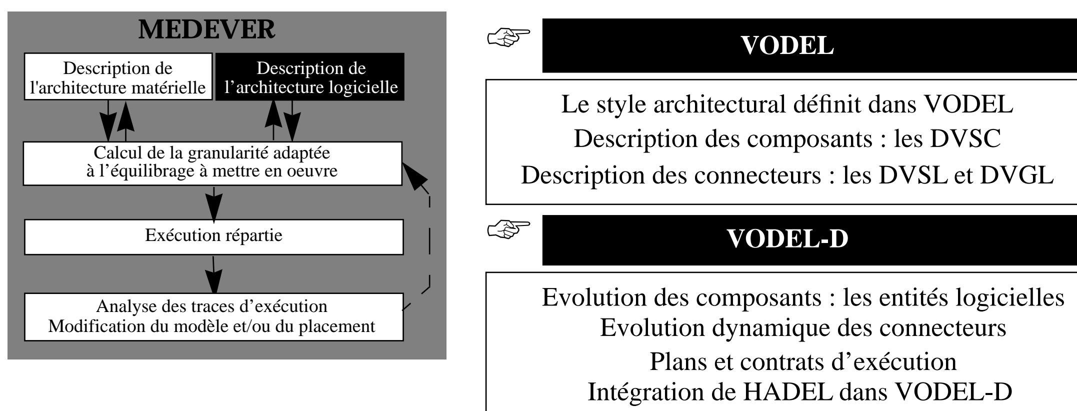
## Résumé

Ce chapitre est consacré à la présentation de VODEL, notre langage de description d'architectures logicielles statiques d'applications client-serveur. Dans VODEL, une application est constituée de composants logiciels communiquant (les DVSC) via des canaux simples (DVSL) ou groupés (DVGL). Le langage VODEL-D est une extension du langage VODEL qui permet de décrire l'évolution dynamique des composants logiciels susceptibles de migrer en cours d'exécution. On sépare alors la description de l'architecture virtuelle issue de la phase de conception (on parle alors d'entités logicielles de définition), de l'architecture réelle de l'application qui évolue durant l'exécution (on utilise alors les entités logicielles d'exécution). Toute modification de l'architecture entraîne la modification du plan d'exécution, conditionnée (ou pas) par un contrat d'exécution limitant le champs d'application de ces changements. L'Annexe C donne la grammaire du langage VODEL que nous avons implémentée.

## Apports scientifiques

Dans ce chapitre, nous montrons qu'il est possible de décrire simplement l'architecture logicielle d'une application en vue du calcul de son placement et de son exécution. Ceci est vrai dans le cas d'applications dont le placement et l'évolution sont déductibles avant l'exécution. On utilise alors le langage VODEL. La prise en compte de la création dynamique de composants logiciels et de leurs migrations (qui sont imprévisibles), entraînent de nouveaux besoins de description. Nous avons créé VODEL-D, qui est une extension du langage VODEL, pour décrire et contrôler l'évolution dynamique des composants d'une architecture logicielle donnée. L'idée principale de VODEL-D est de découpler l'architecture de définition de l'architecture d'exécution, pour isoler et évaluer les effets d'une exécution sur l'arrangement et la granularité des composants logiciels gérés.

## Plan




## Mots clefs

Langage de description d'architecture matérielle, DVSC, DVSL, contrat d'exécution, composants logiciels répartis, canaux de communication, plan d'exécution.

**L'architecture d'une application client-serveur** peut être modélisée et validée (par mise en évidence de problèmes structurels) avant d'être implémentée, dans le but d'écarter au plus tôt des erreurs de conception. Cette architecture peut ensuite être mise en oeuvre de manière automatique via de la génération de code ou de manière manuelle. Cette implémentation est plus ou moins facilitée par les langages de programmation et les systèmes d'exploitation disponibles. Ainsi, une application est développée de manière :

- auto-suffisante, c'est-à-dire qu'elle contient tous les éléments nécessaires à son exécution répartie ;
- dépendante des couches gérant la distribution comme les systèmes d'exploitation répartis ou des plate-formes de bus logiciels (l'axe système) ;
- autonome, dans le sens où elle est structurée en composants logiciels (mobiles ou fixes) et de granularité (variable ou fixe) plus ou moins imposée par le langage de programmation (l'axe langage).

Les solutions proposées pour résoudre ces trois cas sont basées sur des méthodes de description, des langages de programmation, des outils de développement et des environnements d'exécution spécifiques que nous avons passé en revue dans le chapitre précédent. La construction d'applications ne se fait donc plus en considérant l'application comme monolithique, au sein d'un environnement de développement et d'exécution unique. Elle passe dorénavant par l'intégration de différents composants logiciels, dont on ne connaît pas toujours le détail, au sein d'une architecture dite ouverte. Dans MEDEVER, notre approche est de tenter d'intégrer au sein d'une même méthode différents types d'applications, à toutes les étapes du cycle de vie d'une application.

Notre problématique est de compléter les démarches de développement existantes en prenant en compte la granularité des composants logiciels et leur schéma d'interaction. C'est pourquoi, nous avons cherché à renforcer la sémantique contenue dans la description d'une architecture logicielle. Ces informations sémantiques sont ensuite utilisées pour faciliter la conception, la mise en oeuvre, l'exécution et le déboggage des applications parallèles et réparties.

Dans la première section, nous présentons VODEL, qui est le langage pivot de notre méthode MEDEVER. VODEL décrit l'architecture logicielle d'une application de manière modulaire et statique. Avec VODEL, on considère une application répartie ou parallèle comme un ensemble de composants logiciels en relation, à placer sur une architecture matérielle. Ces composants doivent respecter des contraintes (de communication, de synchronisation et de précédence) et peuvent être hiérarchiques, ce qui permet d'adapter leur granularité. Néanmoins, certaines applications actuelles (telles que des applications à bases d'agents mobiles) disposent d'architectures logicielles susceptibles d'évoluer de manière non prévisibles. Le suivi dynamique de leur architecture n'est donc pas possible avec un langage comme VODEL. Nous proposons alors dans la section 2, le langage VODEL-D, qui une extension de VODEL créée pour supporter et éventuellement contrôler les évolutions dynamiques d'une architecture logicielle.

# 1. Le langage VODEL

Notre méthode de développement MEDEVER nécessite la description de l'architecture matérielle d'une application. En nous inspirant de la notion de machine virtuelle utilisée pour s'abstraire du matériel, nous introduisons la notion de logiciel virtuel pour nous abstraire des problèmes d'implémentation et d'exécution. Ainsi, une application est conceptuellement vue comme un ensemble de composants logiciels virtuels répartis communiquants. Ces composants sont des regroupements d'entités parallèles nécessaires à l'obtention d'un grain de parallélisme adapté à une exécution répartie sur une architec-

ture matérielle donnée. La mise en oeuvre de ces composants et de leur correspondance avec des programmes ou des morceaux de code peuvent modifier profondément l'architecture.

Il existe donc deux visions d'une même architecture logicielle. La première, appelée une architecture de définition, est utilisée pour décrire la granularité de chaque composant logiciel et ses interactions avec les autres composants au niveau du modèle. La seconde, appelée une architecture d'exécution, offre une vision opérationnelle de l'architecture, proche de son implémentation. Un langage de description d'architecture doit donc prendre en compte ces deux visions architecturales qui sont, de notre point de vue, capitales pour le placement et le suivi de l'exécution répartie d'une application.

Les algorithmes de placement sont alors soit basés sur l'architecture de définition, soit sur celle d'exécution. Le passage de l'une à l'autre pouvant modifier les paramètres du placement, il est indispensable de préserver la traçabilité sur tout le cycle de développement. Ceci reste vrai à la fois pour l'équilibrage de charge et d'application. Bien que ces types d'équilibrage manipulent des composants logiciels de granularité différente, il est indispensable de définir un style architectural commun pour unifier leur gestion au niveau de MEDEVER.

Un style architectural comprend un vocabulaire d'éléments conceptuels (les composants et les connecteurs), impose des règles de configuration entre les éléments du vocabulaire (ensemble de contraintes) et véhicule une sémantique qui donne un sens (non ambiguë) à la description structurelle [Shaw & al. 96]. Pour nous, il est important que ces styles architecturaux prennent en compte à la fois l'aspect conceptuel (les gabarits de conception) et opérationnel (les squelettes d'implémentation) des architectures logicielles. Le langage de description d'architecture que nous voulons créer doit donc prendre en compte ces contraintes, tout en respectant les objectifs généraux suivants :

1) **la flexibilité** qui est à la fois un objectif et une contrainte. Notre langage doit s'adapter et modéliser le plus grand nombre possible d'applications réparties et parallèles ;
2) **la portabilité** : se voulant le plus généraliste possible, notre langage doit être utilisé sur n'importe quel type de machine. Cet objectif de portabilité est atteint en utilisant des langages et des techniques communes à de grands nombres de systèmes pour l'implémentation de VODEL.
3) **la convivialité** : un langage complexe, verbeux et ambigu ne serait pas utilisé.
4) **la performance** : un des nombreux domaines d'application de notre langage est la description d'une application parallèle ou répartie pour l'équilibrage de charge et/ou d'application. Le langage et ses gestionnaires ne doivent entraîner de surcoût sous peine d'annihiler les gains de performances obtenus par le répartiteur de charge ou d'application.

Nous avons créé le langage VODEL (*Virtual sOtfware DEscription Language*) en prenant en compte toutes ces contraintes. Lors de la création de VODEL, nous nous sommes inspiré des ADL offrant une approche composant et du système d'exploitation Choices (notamment pour son utilisation des gabarits de conception).

**Le but du langage VODEL est de décrire une architecture logicielle pour automatiser et améliorer le calcul du placement des composants qui la constitue et non pas de servir de langage de construction d'applications.** Contrairement à un langage comme Olan, VODEL ne contrôle pas la bonne utilisation des interactions entre les composants logiciels d'une application. **VODEL est donc avant tout un langage intermédiaire de description d'architectures logicielles.** C'est aussi pourquoi, VODEL n'est pas un langage orienté objet, mais plutôt un langage basé sur des classes d'entités logicielles réutilisables.

La section 1 présente le style architectural général défini dans VODEL. La section 2 décrit les spécificités des composants du style architectural, appelés DVSC, qui sont soit actifs soit passifs. Enfin, la section 3, décrit les connecteurs du style architectural qui sont soit simples (les DVSL), soit groupés (les DVGL).

## 1.1. Style architectural définit dans VODEL

Dans le cadre du langage VODEL ne sont modélisées que les applications de type transformationnelles répondant à la deuxième classification de Herman [Herman & al. 83], à savoir, les applications décrivant des machines virtuelles. Dans ce cadre applicatif, on dénombre quatre composantes communes à toute application :

1) le coeur algorithmique : ce point décrit les spécificités de l'application ;
2) les entrées-sorties de stockage : elles correspondent aux accès fichier de l'application ;
3) les communications : elles déterminent l'interactivité de l'application avec son environnement ;
4) l'interface utilisateur : elle correspond à l'interactivité de l'application avec un processus particulier, l'utilisateur.

Ces quatre composantes ont été reprises lors de la création du langage VODEL. Notons tout de même que le coeur algorithmique n'est pas le sujet central de nos travaux. **En effet, notre méthode de gestion des applications client-serveur complète celles existantes, mais n'en redéfinit en aucun cas une nouvelle.**

Par contre, nous fournissons des squelettes d'implémentation (*framework*) pour les trois autres composantes, que nous qualifions d'entités pour simplifier le discours. Ces squelettes d'implémentation ont été mis au point dans un but unique : offrir une description d'une application via des composants logiciels communiquants à partir desquels, en coopération avec le coeur algorithmique, on répartit «au mieux» l'application. Ceci entraîne donc la prise en compte dans VODEL de :

- la description de l'application répartie (composants et communications) ;
- la gestion de la granularité des composants ;
- le suivi dynamique des composants (gestion de la mobilité).

Les deux premières fonctionnalités sont discutées dans la suite ce paragraphe, tandis que la troisième est développée au sein de la Section 2.

### *1.1.1. Description de l'application répartie*

La première composante du langage VODEL traite de la description de l'application répartie. **Cette description est statique** et modélise les différentes entités composant l'application ainsi que les relations entre ces entités.

Nos travaux sont principalement basés sur ceux de Choices. Choices est un système d'exploitation à objets répartis et un environnement de développement de services systèmes. La principale caractéristique de la méthode de conception mise en place est de permettre la construction du système de manière modulaire et indépendamment du support matériel. Pour atteindre cet objectif, un squelette d'implémentation dans Choices est créé en quatre phases :

1) modélisation de l'application avec définition sémantique des objets de base du sous-système en construction ;
2) transcription de cette modélisation en modèle objet ;
3) détermination du schéma de coordination entre objets (modèle entité-relation) ;
4) détermination du flot de contrôle, c'est à dire du séquencement des différents

appels de méthodes réalisés sur les composants.

Le squelette d’implémentation de Choices a été développé pour supporter des contraintes de programmation en vue de la validation des sous systèmes avant leur intégration dans un système final. Le squelette d’implémentation définissant alors des relations inter-entités. **La problématique de VODEL est autre et vise à définir une méthode pour décrire les applications réparties en modifiant la granularité des composants gérés si nécessaire.** Ces composants étant des entités actives (en général des processus) interagissant soit avec des entités actives, soit avec des entités passives (fichier, processeur, mémoire etc...). Même si notre problématique se situe à un autre niveau que celle de Choices, car centrée sur la granularité des composants d’une architecture, notre approche reste identique.

#### *1.1.2. Définition de la granularité*

Donner une définition de la granularité d’un composant n'est pas chose simple. La granularité est une notion générique. La granularité dans notre cas est liée au regroupement d’un ensemble d’entités actives ou passives. Le composant construit étant alors calculé en fonction de contraintes de placement telles que le temps CPU consommé pour l’exécution d'une application, les temps de communication entre entités ou les spécificités des entités à l'égard d'un serveur particulier.

Dans le cas le plus général, la granularité d'une application répartie correspond au nombre de composants (actifs et/ou passifs) qui la composent et à leurs caractéristiques (temps d’exécution et taille des données manipulées). Ainsi, une application de forte granularité disposera de composants actifs consommant beaucoup de temps CPU et manipulant beaucoup de données.

Le calcul de la granularité d’un composant logiciel est aussi lié aux communications entre entités actives, car plus le grain de parallélisme est faible plus le coût des communications entre entités actives est important. Le but est alors de trouver le meilleur compromis entre les communications et le grain de parallélisme souhaité d’une application donnée. Il faut donc décrire cette dernière et la modifier dynamiquement pour obtenir les meilleures performances. La migration d’entité(s) active(s) est une solution à ce problème. Néanmoins, au niveau système et réseau, migrer des processus coûte cher et le coût de cette migration peut ruiner les performances d’exécution d’une application.

Enfin, à des critères de bas niveau s’ajoutent d'autres critères, liés aux spécifications de l'application. Ces informations extraites de la spécification donnent des précisions sur la manière dont vont être utilisés les composants. Moyennant une sémantique de description connue à l’avance, il est possible de tirer partie des descriptions de haut niveau pour concevoir la granularité des composants de l’application [El Kaim & al. 94].

#### *1.1.3. La granularité gérée par VODEL*

Dans VODEL, nous considérons que les entités logicielles à gérer dans une application client-serveur réelle sont globalement de quatre types : les fichiers, les structures de données, les tâches et les processus et les objets :

- un fichier est un conteneur persistant d’informations structurées et indivisibles. Il est réplicable avec ou sans respect des cohérences entre les copies. Les codes des processus sont en général stockés dans des fichiers.
- les phénomènes ou les systèmes complexes qu’on cherche à modéliser afin d’en étudier le comportement (comme par exemple pour comprendre les phénomènes météorologiques) utilisent des structures de données sur lesquelles on applique des traitements. Ces structures ont dans certains cas des tailles importantes à gérer au niveau du support de stockage et de la mémoire dynamique.
- une application complexe est composée de tâches indépendantes qui s’échangent ou non des données de calcul ou de contrôle. Les tâches sont constituées de processus qui correspondent aux entités actives. Le déroulement de l’exécution de

l'application est alors un graphe de dépendance des tâches. La tâche a donc par définition une granularité plus forte que le processus.

- enfin, l'émergence des technologies objets a abouti à la création d'applications basées sur des objets fixes ou mobiles et dotés de plus ou moins d'intelligence. La représentation des objets et leur granularité sont très variables.

#### *1.1.4. Les entités logicielles (DVSC) et leurs liens (DVSL)*

C'est pourquoi VODEL est construit autour de deux composantes : les entités logicielles et les relations de dépendance entre ces entités. Deux types d'objets distincts ont alors été définis :

- les DVSC (*Distributed Virtual Software Component*) qui sont des composants logiciels virtuels et répartis. Ils représentent soit des entités actives qui s'exécutent (un ensemble de processus par exemple), soit des entités passives (des ressources systèmes par exemple). Les DVSC sont hiérarchiques et des DVSC d'un même niveau sont agglomérables dans un seul et unique DVSC de niveau immédiatement supérieur. Cette hiérarchie joue un rôle dans la construction d'une entité parallèle ayant une bonne granularité pour chaque machine candidate à l'exécution répartie.
- les DVSL (Distributed Virtual Software Links) qui représentent des liens de communication virtuels entre les DVSC. Chaque DVSL est caractérisé par un poids qui exprime le coût relatif d'un accès et sa fréquence. Les informations attachées à un DVSL évoluent en cas de migration. Enfin, les DVSL sont agglomérables au sein de DVGL (*Distributed Virtual Group of Links*).

Nous approfondissons dans la suite la description des DVSC avec le langage VODEL.

## 1.2. Description des entités logicielles (ou DVSC)

Dans VODEL, les entités composant le squelette d'implémentation sont décrites au moyen de classes. Comme dans Olan, nous avons introduit la notion de classe et d'instance pour renforcer la distinction entre l'unité de réutilisabilité et d'encapsulation (la classe de composant) et l'entité effectivement instanciée à l'exécution (le composant).

Nous précisons dans la suite de ce paragraphe ce qu'est une classe d'entités logicielles, avant de passer en revue les deux types d'entités logicielles existantes, à savoir les entités passives logiques et les entités actives.

#### *1.2.1. Les classes d'entités*

Trois éléments de base composent les classes de VODEL : le type de la classe, son caractère (abstraite ou concrète) et ses relations de dépendances.

*Type d'une classe*

Toute classe dans VODEL est typée. Le type permet de regrouper différentes entités composant un squelette d'implémentation. Toute entité appartient à un groupe, composé de toutes les entités de même type. Comme une entité possède un type unique, celui-ci n'appartient qu'à un seul groupe. Le type de la classe doit être prédéfini au moyen d'un identifiant de classe.

*Caractères de la classe*

Deux caractères de classes sont définis.

- **classe concrète** : cette classe possède toutes les informations nécessaires à son instanciation. Cette classe définit une représentation opérationnelle des entités.
- **classe abstraite** : une classe abstraite est une classe dont toutes les composantes liées à la définition de l'objet ne sont pas présentes dans la classe (telle que la localité). Cette classe est la seule à pouvoir être modifiée dynamiquement. Ce type de classe sert de méta-modèle. Ce type de classe permet de définir une représentation

fonctionnelle des entités.

*Dépendances entre classes*

Toutes les classes dans VODEL sont typées et peuvent dépendre d'une autre classe (relation «est un» ou *is-a*). Cet dépendance sera interprétée de manière différente suivant le type de la classe fille.

### *1.2.2. Les entités passives logiques*

Dans la plupart des langages de description tels que CSP [Hoare 78], Estelle [Turner 93] et Lotos [Boloynesi & al. 87], les seules composantes prises en compte pour la description des applications parallèles sont les processus et les liens de communication entre eux. Pourtant, les composantes externes à l'application font partie intégrante de celle-ci. Ces composantes externes peuvent être de différents types (des fichiers, du code, des programmes). Actuellement très peu de systèmes de programmation ou d'exploitation considèrent les accès aux ressources comme partie intégrante de l'application. Ainsi par exemple sous Unix, l'accès à ces composantes externes est réalisé en passant l'application du mode utilisateur au mode noyau [Bach 91]. Dans ce système une distinction est néanmoins effectuée car l'accès aux composantes externes est commun à toutes les applications.

Si le programmeur peut définir tous les types de ressources qu'il désire, VODEL dispose d'un ensemble de classes prédéfinies telles que les classes Fichier, Code et Initialisation (cf. Annexe C).

#### L'entité passive logique Fichier

Le caractère abstrait ou concret de la classe fichier correspond à la production d'un fichier. Un fichier en attente de production, donc sans information de localisation sera déclaré de type abstrait. Lorsque le fichier est créé, la classe est raffinée et remplacée par une classe concrète. Les différents champs sont le nom du fichier, le mode d'accès au fichier et la localisation du fichier sur le réseau.

#### L'entité passive logique Code

L'objet associé au code donne le nom de l'application et le chemin localisant le code de celle-ci, en donnant l'identifiant d'un objet de type fichier. Une application recompilée sur plusieurs machines possède plusieurs codes disponibles attachés à un objet CPU. Nous verrons que l'objet modélisant le processus pourra posséder plusieurs héritages d'objets de type code.

#### L'entité passive logique Initialisation

Cette entité contient toutes les valeurs d'initialisation, de paramétrage et de configuration de l'application, que ce soit pour de l'équilibrage de charge ou d'application.

### *1.2.3. Les entités actives*

L'entité active représente le plus bas niveau de parallélisme d'une application. L'exécution d'une entité active est séquentielle et c'est l'instanciation de la classe entité active qui crée le processus. Dans VODEL, deux types d'entités actives existent : les processus et les processus légers.

*Les processus*

Un processus possède son propre flot de contrôle et un contexte mémoire qui lui est propre. Il n'existe donc pas de recouvrement mémoire entre processus, ce qui est un gage de sécurité. En effet, la «mort brutale» d'un processus ne remet pas en cause l'exécution des autres processus. Ce contexte mémoire comprend le code de l'exécutable et ses données, la pile ainsi que le contexte d'exécution (état du processus, valeur du compteur ordinal et

de certains registres du processeur, informations statistiques, etc.). Dans le cas de plusieurs processus se partageant le même processeur, les changements de contexte se révèlent très coûteux en temps et ralentissent les performances du système. Les processus entre eux forment des entités distinctes pouvant communiquer entre elles. Les processus possèdent leur propre espace d'adressage et parcourent un unique fil d'exécution.

*Les processus légers*

Les processus légers ont été créés dans l'optique de minimiser la durée du changement de contexte. Ainsi, plusieurs flots de contrôle s'exécutent dans un même contexte mémoire, et plus généralement sont subordonnés à un processus. Dans Chorus [Jacquemot 94], une nouvelle terminologie est employée : les processus légers sont des entités et les processus sont des acteurs.

Les processus légers permettent de définir une application comme un ensemble d'entités partageant le même espace mémoire et communiquant entre elles par mémoire partagée. Ce type de communication simplifie et accélère les synchronisations entre entités parallèles ou concurrentes. Si les processus peuvent appartenir à plusieurs utilisateurs, les processus légers par contre n'appartiennent qu'à un unique utilisateur.

Une très bonne synthèse de ce sujet est faite par [Demeure & al. 94] ou par le chapitre 15 de [Mullender 93].

*Caractérisation des processus*

On distingue globalement quatre types de processus :

- les filtres acceptent des données en entrée, réalisent des modifications sur ces données et les transfèrent à d'autres processus ;
- le client est un processus qui fait des demandes de traitements et qui attend qu'on lui envoie la réponse ;
- le serveur est un processus qui réagit aux demandes de traitement de la part de clients et qui renvoie les résultats des traitements demandés au client ;
- les processus apparentés sont les instances distinctes d'une même classe de processus. Leur comportement interne est strictement identique, seules les variables d'initialisation changent.

Ces quatre types de processus sont gérés par VODEL, en spécialisant les liens entre entités actives (appelés les DVSL). On décrit alors précisément les interactions mises en oeuvre entre deux entités actives (type de communication, ordonnancement des messages, etc.).

*Modélisation des dépendances intrinsèques aux processus*

Les descripteurs d'une entité active sont :

1) les ports d'entrée-sortie de l'entité ;
2) les besoins en mémoire de l'entité active ;
3) un ou des codes, permettant d'identifier sur quel type de machine l'entité active peut être exécutée. Il est à noter que le terme code désigne tout fichier de données interprétable par une classe machine via son processeur.

Une fois les DVSC présentés, il nous reste à présenter comment sont décrites les interactions entre DVSC.

## 1.3. Les entités de communication dans VODEL : DVSL et DVGL

De nombreux langages ont offert des techniques de description des communications. Dans CSP [Hoare 78], la communication est uniquement synchrone et nominative. Dans le langage SDL [SDL 92], les communications sont décrites au moyen de canaux. Un

canal est alors une structure contenant des informations internes propres à sa gestion, telle que la capacité du canal. Dans un tel langage, les communications sont modélisées hiérarchiquement pour décrire des protocoles sous forme de couches.

Dans VODEL, la description des communications est basée sur la notion de canal issue de SDL. En utilisant un canal comme lien de communication, les entités actives n'ont pas la connaissance de l'identité du consommateur ou du producteur. Elles sont alors réutilisables. Un canal est donc une entité à part entière, ce qui permet de lui associer plus d'informations qu'un simple lien de communication.

**Le canal est le résultat de notre conviction concernant la nécessité de décrire des interactions variées au niveau architectural et des médiateurs adaptés pour leur implémentation**

### *1.3.1. L'objet canal*

Comme la classe entité active, les canaux possèdent une entrée et une sortie modélisée sous la forme de deux descripteurs. Nous posons néanmoins comme condition que le canal ne possède qu'une seule entrée et sortie à la fois (condition uniquement établie pour se placer dans un contexte connu de la communication).

Le canal est donc le moyen de communication entre les entités actives et possède différentes propriétés qui sont :

- **le type de la classe** : *abstraite* ou *concrète*. Le type du canal est abstrait s'il est raffinable et donc non directement instanciable. Le caractère abstrait permet de modifier certaines propriétés du canal, et autorise un spécification des attachements particuliers à ce dernier de façon dynamique.
- **les descripteurs d'entrées-sorties** : *IN* et *OUT.* Comme pour la classe entité active, les canaux possèdent des descripteurs qui sont les points d'entrées (*in*) et de sorties (*out*) du canal. Ils possèdent un champ capacité qui correspond au nombre d'octets que le canal peut mettre en attente (tampon d'émission et de réception).
- **le type de communication** : *synchrone, asynchrone, send-reply (RPC).* Ce type d'information permet dès lors de modéliser tous les types de communications entre applications parallèles.
- **la politique de gestion du canal** :
  - *F.I.F.O* : le premier message arrivé sera le premier message sorti.
  - *L.I.F.O* : le dernier arrivé sera le premier sorti.
  - *Générique* : aucun ordre de sortie n'est implémenté.
- **le mode d'évolution** : *partageable*, *réplicable*, *migrable*. Ces informations sont nécessaires pour la gestion dynamique du canal, que nous étudions en Section 2.3.
- **le mode d'accès aux entités passives** : *création*, *modification* et *lecture*.

La gestion des groupes de communication a aussi été prévue aux sein des canaux.

*La gestion des groupes*

Les canaux peuvent être partagés, ce qui donne la possibilité de gérer la diffusion totale (*broadcast*) et sélective (*multicast*). Cela peut sembler redondant, mais, grâce à cette astuce on définit deux types de groupes (cf. Figure 18) :

1) les groupes composés d'entités actives fortement couplées.
Ces groupes d'entités logicielles sont fortement couplés car les instances de classes VODEL possèdent une bonne partie (voire la totalité) du code en commun. Une instance de classe fait partie d'un groupe si elle dépend d'une classe entité active possédant plus d'un fils. Autrement dit, on réalise la mise en commun d'un certain nombre de descripteurs d'objets VODEL et il est déconseillé de les découpler.

2) les groupes composés d'entités actives faiblement couplées.
A l'opposé des premiers, ces groupes sont composés d'entités actives toutes disjointes, mais ayant par définition la volonté de partager un canal de communication.

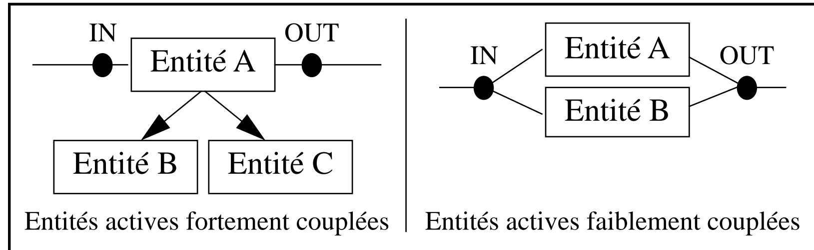



**Figure 18 :** Groupes d'entités actives fortement et faiblement couplées

Ces deux sémantiques nuancent la modélisation de la communication sur groupe et peut entraîner des prises en compte différentes des entités actives par les langages et les systèmes sous-jacents. Pour exemple, la modélisation de l'abonnement à un serveur de communication s'effectue par la deuxième technique.

*Les DVSL*

**Les DVSL** (*Distributed Virtual Software Links*) définissent l'attachement d'un canal à une entité active (cf. Figure 19). Ils reprennent les mécanismes de Lotos [Boloynesi & al. 87] pour la prise en compte de la hiérarchique et ceux de SDL [SDL 92] pour la notion de boites noires.

Un DVSL comprend sept paramètres qui définissent le lien virtuel entre une entité active et une autre via un canal :

1) identification de l'émetteur
   - identifiant de l'entité active source
   - identifiant du descripteur émetteur de la source
2) identification du canal
   - identifiant du canal
   - identifiant du descripteur récepteur du canal
   - identifiant du descripteur émetteur sur le canal
3) identification du récepteur
   - identifiant de l'entité active réceptrice
   - identifiant du descripteur récepteur de la destination

Ces différents champs décrivent les entités actives communiquantes, tout en identifiant les descripteurs associés à cette communication.

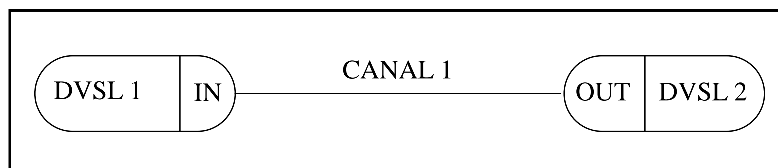



**Figure 19 :** Deux DVSC reliés par un DVSL

*Les DVGL*

**Un DVGL** (*Distributed Virtual Software Links*) comprend un ensemble de DVSL et éventuellement d'autres DVGL.

Le système de description hiérarchique permet de :

1) définir différents types de liaisons pour une même entité active et pour des contextes différents d'exécution. Cela correspond à la réutilisation de l'algorithmique de l'entité active dans différents DVSL ;
2) décrire une application répartie de manière modulaire, pour faciliter la conception et réaliser des systèmes emboîtables (sur le modèle des poupées russes) ;
3) réutiliser d'autres DVGL ;

Ainsi, un canal peut être aggloméré dans un autre canal. Ceci permet de définir des groupes de canaux comme les groupes d'entités actives fortement couplées. Tout canal appartenant à un groupe possède les mêmes attributs de gestion que les autres canaux composant ce groupe. Cela permet de définir et de raffiner les canaux en fonction du contexte d'exécution en subdivisant un canal quand celui ci est invoqué par un ensemble de processus répartis sur plusieurs machines. Le canal devient alors un commutateur redirigeant les canaux concrets à travers les canaux abstraits. Un canal concret correspond à un canal instanciable ce qui fait de lui le canal de plus bas niveau (donc ne pouvant être subdivisé). Un canal abstrait est non instanciable et sert de meta-modèle.

La notion de groupe de canaux (ou DVGL) permet de modéliser une communication sur groupe de façon transparente comme le montre la Figure 20. L'entité active 1 envoie ses messages sur le canal abstrait A. Tous les canaux concrets (trait simple sur la Figure 20) héritant du canal A, acheminent le message. L'entité active 2 réceptionne tous les messages transportés par les canaux concrets du canal virtuel A. Dans le contexte représenté par la figure, l'entité C réceptionne les messages émis par les entités actives 1 et 2. On remarque enfin, que les canaux et les entités actives possèdent tous des ports d'entrée et de sortie. Ces ports sont destinés à être connectés.

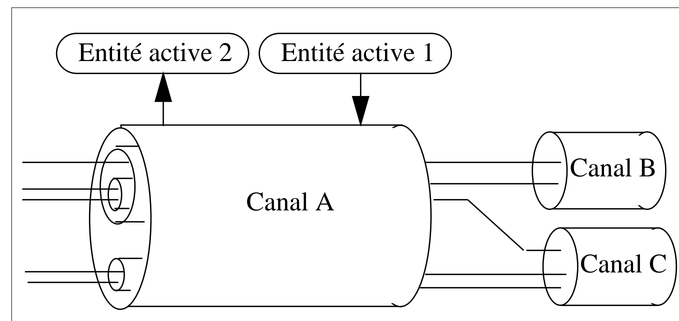



**Figure 20 :** Communication sur groupe de canaux et commutation

### 1.3.2. *Rattachement d'une entité à un canal*

La dynamicité de la représentation permet de définir à l'exécution, le nombre de liens de communication qui vont être nécessaires. Cette dynamicité est réalisée par une méthode de description de canaux abstraits, puis d'un renommage. Ainsi, lorsqu'on utilise des canaux abstraits, le descripteur de communication abstrait est non directement interprétable par le gestionnaire de communication, car il définit une association inter-classe plutôt qu'une réelle liaison. Il faut donc «déplier» le modèle en créant les canaux concrètement.

Cette fonctionnalité est pratique dans le cas où on gère une application parallèle à forte symétrie de code dont seuls les liens de communication inter-entités changent. Le renommage modifie le source d'un descripteur de communication (ou DVSL) abstrait et lui assigne une valeur. On gère ainsi par la même occasion le modèle multiproducteurs-multiconsommateurs. Chaque producteur spécialise alors le canal en créant une classe canal i, dépendante de la classe canal générale propre à son instance. Chaque consommateur i s'attache ensuite à cette classe canal i, créée par le producteur i.

La Figure 21 décrit une application parallèle en pipeline. Le fichier VODEL source (dans l'encadré) correspond à la définition d'un DVGL composé d'entités actives (nommées «enA») et de DVSL (nommés «canal»). Cette définition est paramétrée par un indice i, qui varie de 1 à 3 en définissant ainsi les différents liens de l'application.

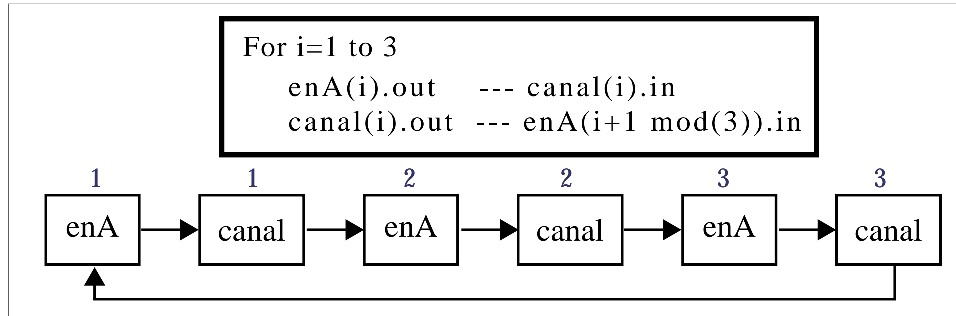



**Figure 21 :** Communication d'une application data-parallèle au moyen du renommage

### *1.3.3. Choix du protocole de communication d'un DVSL*

En partant du principe que les applications gérées par VODEL étaient dynamiques par nature, nous avons défini un champ spécifique indiquant si la communication est locale ou globale. Cette information a des répercussions, aussi bien au niveau du protocole de communication utilisé, qu'en terme de vitesse et de fiabilité des communications. Dans l'idéal, si la communication est locale les entités logicielles communiquent via un segment de mémoire partagée. Comme l'entité active ne connaît pas la nature du canal, mais seulement son point d'entrée, cette modification n'entraîne pas un changement de code (aussi bien de l'exécutif que du source) pour l'entité active. Les communications globales sont par défaut basées sur des sockets et le protocole TCP/IP.

### *1.3.4. Communication entre les diverses instances d'une même entité active*

La description de la communication dans le langage VODEL est basée sur le canal et les classes entités actives. Mais, jusqu'à présent, la communication inter-processus n'a pas été décrite (communication entre des instances de la même classe).

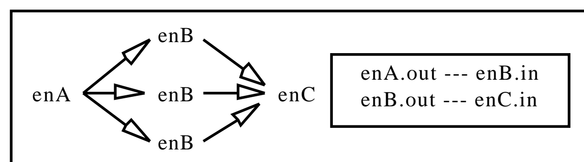



**Figure 22 :** Communication entre les instance d'une même classe

Si la redirection des flots de données est identique, la description est implicitement réalisée par le DVSL comme le montre la Figure 22. Trois processus *enB* sont exécutés. Mais les flots de données sont identiques et se réfèrent au même DVSL. Les trois processus reçoivent les messages de *enA* et émettent à *enC*.

Dans le cas particulier de redirection d'un flot propre à chaque processus, leurs définitions doivent être effectuées en utilisant la hiérarchie présente dans les DVGL.

### *1.3.5. Communication dans un groupe de processus*

La modélisation de la communication dans groupe de processus est une autre fonctionnalité de VODEL. Une classe entité active représente les liens de communication entrants et sortants. Le processus est une instance de cette classe. Grâce à l'agrégation, on décrit la communication dans un groupe. En effet toute classe d'entités actives dépendant d'une autre classe entité active possède les mêmes propriétés que son père. Ainsi, différents fils d'une classe entité active partagent les mêmes descripteurs donc les mêmes liens de communication. Sur l'exemple de la Figure 23, deux classes sont définies (A et B). Chaque

classe possède des descripteurs. L'instanciation de la classe B entraîne «l'héritage» des descripteurs de la classe A et le processus possède un descripteur en entrée et trois en sortie. Si la classe C est instanciée, les descripteurs 1, 2 et 3 seront partagés. Tous les messages reçus sur le descripteur 1, seront lus par les processus A et B.

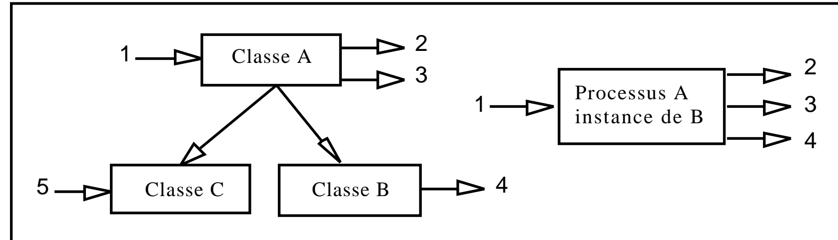



**Figure 23 :** Communications dans un groupe de processus

### 1.3.6. *Synchronisations par rendez-vous ou moniteur*

**La synchronisation entre entités actives revient à réaliser une communication avec deux entités actives particulières le moniteur et le rendez-vous** :

- le rendez-vous consiste à établir une communication synchrone avec la classe rendez-vous ;
- le moniteur consiste à établir un communication RPC avec la classe moniteur.

*Rendez-vous*

Le canal de communication est de type synchrone. Sur la Figure 24, les ports de sortie des entités actives 1 et 2 sont reliés aux ports d'entrée du canal. Une commutation est effectuée au sein de canal pour rediriger les entrées vers une seule sortie. Enfin, le port de sortie du canal est alors relié au moyen d'un DVSL au port d'entrée de l'entité rendez-vous.

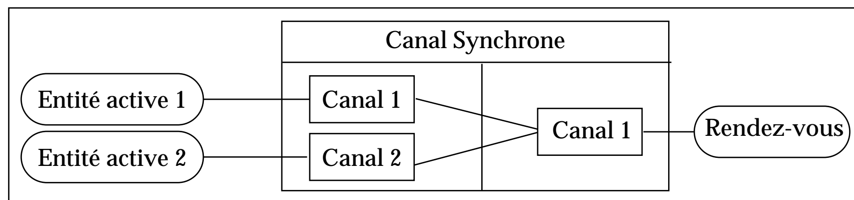



**Figure 24 :** Synchronisation par rendez-vous

*Le Moniteur*

La Figure 25 présente une communication de type Send-Reply.

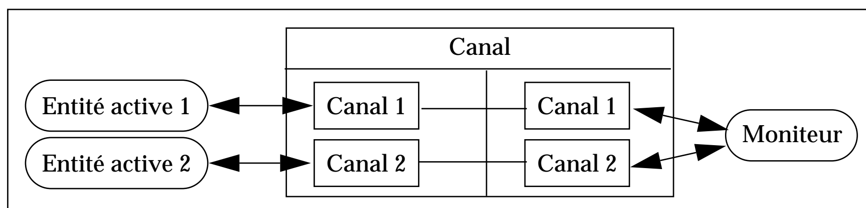



**Figure 25 :** Synchronisation avec un moniteur

Les ports de sortie des entités actives 1 et 2 sont reliés aux ports d'entrée du canal. Le canal, abstrait, est subdivisé en plusieurs canaux au moyen du renommage et chacun de ces canaux est du type Send-Reply. Il accèdent ensuite via un canal à l'entité Moniteur. Le source du moniteur correspond à une incrémentation ou une décrémentation d'une variable.

## 1.4. Synthèse

Les quatre points clefs fixés lors de la construction du langage VODEL à savoir la flexibilité, la portabilité, la convivialité et la performance ont donc été respectés. VODEL est flexible car c'est un langage basé sur des composants logiciels prédéfinis et/ou définissables. Il est portable, car il est basé sur des objets qui encapsulent complètement les composants réels de l'application (comme le code source par exemple). Il est convivial car basé sur un langage de description déclaratif (cf. BNF en Annexe C) qui peut être repris dans un éditeur graphique. Enfin, il est performant dans le sens ou il permet la description d'architecture d'applications client-serveur, selon deux points de vue différents en utilisant les mêmes squelettes d'implémentation.

Le langage VODEL est pour cela composé de trois types d'objets : les DVSC, les DVSL et les DVGL. Les DVSC sont des instances des entités actives ou passives et représentent des composants logiciels de l'application. Les entités actives communiquent entre elles via des liens de communications virtuels appelés des canaux, qui une fois instanciés deviennent des DVSL. Ces canaux sont eux mêmes agrégés au sein de canaux de plus hauts niveaux, les DVGL. Les différents types de DVSC et de DVSL sont résumés dans la Figure 26.

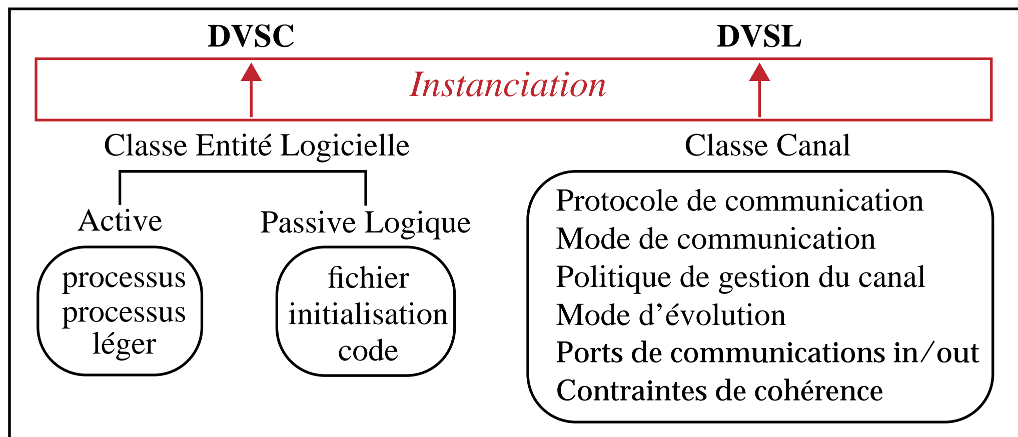



**Figure 26 :** Les principales entités définies dans VODEL

VODEL tel que nous l'avons défini jusqu'à présent est apte à définir des architectures logicielles de manière statique. Or, la prise en compte du point de vue équilibrage de charge entraîne la prise en compte de la gestion des évolutions de l'architecture au cours de la durée de vie de l'application. Nous avons donc étendu VODEL, pour qu'il gère la dynamicité. Nous présentons dans la suite ces extensions qui ont donné naissance au langage VODEL-D («D» pour Dynamique).

# 2. Intégration de la dynamicité dans VODEL : VODEL-D

Jusqu'à présent, les composants logiciels décrits dans VODEL étaient composés d'entités actives et passives. En ce qui concerne les entités actives, support de la dynamicité, elles sont représentées une fois instanciées par la classe de l'entité active et le descripteur de l'entité de communication. On en déduit donc que les entités logicielles définissent l'application par des groupes statiques et hiérarchiques d'instances de composants logiciels. Les entités logicielles ne disposent donc pas d'une composante de suivi de l'exécution réelle du ou des exécutables composant l'application. Or cette composante devient necessaire dans le cas de placement de programmes parallèles irréguliers ou des traitements répartis non répétitifs.

Ainsi, le placement statique de VODEL est ensuite susceptible d'évoluer en fonction de la charge des machines et de la gestion dynamique des tâches. Le langage VODEL-D est une extension du langage VODEL qui permet de décrire l'évolution dynamique des composants logiciels susceptibles de migrer ou d'être créés en cours d'exécution.

**On sépare alors la description de l'architecture virtuelle issue de la phase de conception (on parle alors d'entités logicielles de définition), de l'architecture réelle de l'application qui évolue durant l'exécution (on utilise alors les entités logicielles d'exécution).**

Toute modification de l'architecture entraîne la modification d'un plan d'exécution, conditionnée (ou pas) par un contrat d'exécution limitant le champ d'application des évolutions dynamiques de l'architecture.

Dans la suite de cette section nous présentons d'abord les entités logicielles de définition, puis les entités logicielles d'exécution. Nous décrivons ensuite les extensions apportées à l'objet canal pour la gestion des communications entre des composants qui évoluent dynamiquement. Enfin, nous présentons le concept de plan d'exécution, qui dévient réellement intéressant lorsque la description de l'architecture matérielle HADEL est intégrée dans VODEL-D.

## 2.1. Les entités logicielles de définition : étendre les DVSC

Les entités logicielles de définition étendent les DVSC définis dans VODEL en intégrant des contraintes d'évolution de la granularité et de gestion des interactions. Des attributs spécifiques ont donc été ajoutés. Ces attributs sont soit générés à partir de la spécification opérationnelle de l'application (si elle existe), soit précisés par le concepteur de l'application. Les attributs que nous avons définis qui concernent la gestion de la granularité sont les suivants (cf. Annexe C) :

- **hiérarchie** : un composant logiciel peut être *primitif* ou *hiérarchique*. S'il est hiérarchique, il contient la description des entités logicielles qu'il contient ;
- **regroupement** : un composant logiciel hiérarchique est *dissociable* si on peut le migrer par partie, sinon, il est *indissociable* ;
- **liens** : un composant logiciel peut avoir des liens vers des objets noirs qu'il faut prendre en compte.

Les attributs utilisés pour le calcul du placement sont les suivants (cf. Annexe C) :

- **type** : un composant logiciel est *actif* s'il contient au moins un processus, sinon il est *passif* ;
- **évolution** : un composant logiciel est *libre* s'il peut être déplacé, sinon il est *fixe* ;
- **répartition** : un composant logiciel est *réplicable, migrable* ou *centralisé*.
- **interactions** : deux composants logiciels supposés communiquer intensivement doivent rester proches pour éviter la dégradation des performances. On définit alors un lien d'*attraction* entre eux. Deux composants logiciels qui se repoussent pour cause de conflits, possèdent par contre un lien de *répulsion*. L'attachement qui existe entre entités logicielles correspond donc à une force d'attraction d'instances. Si la valeur de celle-ci est infinie, les processus résultant des différentes instanciations sont placés sur la même machine. Dans un cas contraire, la modification dynamique et le placement tient compte des différents attachements afin de sélectionner au mieux des spécifications, les instances à migrer (de machine à machine, ou d'entité logicielle à entité logicielle). De la notion d'attachement on déduit immédiatement la propriété suivante : toutes les communications dans une entité logicielle attachée sont locales.

Une fois l'architecture de l'application virtuelle décrite, il reste à la mettre en place réellement et à suivre son évolution en terme de répartition de charge et de migrations éventuelles. On utilise alors des entités logicielles d'exécution.

## 2.2. Les entités logicielles d'exécution

**Les entités logicielles d'exécution ont été créées pour prendre en compte les contraintes d'exécution et sont composées des instances des processus qui s'exécutent réellement.**

Des travaux ont déjà été réalisés sur cette notion d'entité logicielle d'exécution. Citons entre autres ceux réalisés sur COOL. La principale différence entre COOL et VODEL-D réside dans le fait que dans le système COOL, la notion d'entité logicielle de définition n'existe pas. Le placement des objets par ce système ne tient donc pas compte des spécifications de l'application, mais seulement des contraintes d'exécution. Dans notre cas, la combinaison des deux types d'entité logicielle permet de placer les entités logicielles de définitions attachées en fonction de la configuration courante des communications et de la description faite par le programmeur de son application.

Les paramètres supplémentaires à prendre en compte par rapport aux entités logicielle de définition sont les suivants (cf. annexe C) :

- **groupe** : les entités logicielles d'exécution imposent que chaque processus appartienne à un groupe unique, représenté par un numéro. Le fait d'appartenir à un groupe signifie que l'entité active réalise des communications importantes avec les autres entités actives de ce groupe. Un groupe possède un indice de liaison qui correspond au coût de la communication entre entités logicielles et permet de spécifier des liaisons d'attraction-répulsion.
- **charge** : un indice de charge correspondant à la charge totale du groupe a aussi été défini. Cet indice est un facteur déterminant lors de la migration des processus. En effet, lors du lancement du programme, un groupe de coût 0 est créé avec tous les processus. Dès que les communications commencent, l'évolution de l'indice de charge suit le nombre et le coût des communications réalisées par un groupe. Si ce n'est pas la première fois qu'on exécute une application, on dispose de la description hiérarchique des groupes qui ont été mémorisés précédemment. Le gestionnaire de placement est alors à même de prendre en compte cette information lorsqu'un nouveau processus est lancé.
- **placement** : il est possible d'indiquer sur quel processeur ou sur quelle machine sera placée l'entité logicielle.
- **si modification, quelle est sa raison** : si l'entité logicielle de définition est modifiée lors de l'exécution, on indique alors pour quelle raison : équilibrage de charge ou d'application. La traçabilité des composants entre l'architecture de définition et d'exécution est alors assurée.
- **importation** : toute entité logicielle d'exécution importe la description des entités passives et actives qui la compose.

La gestion de la dynamicité a de plus entraîné l'extension des propriétés de l'objet canal.

## 2.3. Modélisation des contraintes d'évolution dynamique d'un canal

Comme on se place dans un contexte d'applications client-serveur qui sont composées d'entités logicielles mobiles, on se doit de prendre en compte les modifications que cela engendre pour les canaux de communication concernés. Si les entités actives migrent, le canal doit accompagner cette migration. Néanmoins, on peut empêcher toute évolution pour des questions de bon fonctionnement (mémoire insuffisante sur la machine réceptrice, code non recompilé pour celle ci, etc.), le canal restant attaché à une machine.

Enfin, il est parfois nécessaire de pouvoir *dupliquer* un canal. En effet, si les entités actives destinatrices sont subdivisées en deux sous parties pour des problèmes d'équilibrage (de charge ou d'application) et placées sur des machines différentes, il est alors primordial de pouvoir dupliquer le canal. Si un canal est dupliqué, les contraintes de cohérence liées aux interactions entre les producteurs et les consommateurs doivent être réglées.

Ainsi, la contrainte de duplication peut être :

- **causale** : un message est délivré si tous les messages dont il dépend causalement ont été délivrés ;
- **séquentielle** : le message est délivré si tous les messages inférieurs, à l'ordre total établi, ont été délivrés.

La contrainte de cohérence sur les canaux n'est pas seulement réservée au seul problème de la gestion du canal. Elle peut être vue aussi comme un filtre de messages. Si un canal est utilisé dans un système où l'environnement ne gère pas les messages transmis, ces derniers n'ont pas de raison d'être envoyés puisqu'ils demeurent intraitables dans l'environnement de l'entité réceptrice. Ce filtrage est en fait un constat d'impuissance (ou un refus) de l'application qui ne traite pas les messages reçus. Nous avons donc défini dans le langage VODEL-D, un champ de définition de la contrainte de cohérence d'un canal si une réplique du canal est nécessaire.

Ce champ peut prendre deux valeurs (cf. Annexe C) :

1) **duplication sans contrainte de cohérence**
Dans ce cas précis, les canaux peuvent être dupliqués sans aucune contrainte. Ceci implique que la gestion du canal n'a pas à être faite et que le canal est créé localement.

2) **duplication avec contrainte de cohérence**
Un canal possédant cette propriété doit, lors de son instanciation, être intercepté par le module implémentant le protocole de cohérence. Celui-ci vérifie alors si le canal n'existe pas déjà dans l'environnement d'exécution. Si c'est le cas, un gestionnaire de communication est chargé de maintenir la cohérence entre les différentes instances d'un même canal.

Enfin, pour compléter le langage, nous avons créé la notion de plan d'exécution, qui facilite le suivi dynamique d'une application client-serveur.

## 2.4. Les plans d'exécution

Un plan d'exécution correspond à un instantané de la répartition et de l'état des entités logicielles d'exécution. Chaque changement de plan indique les évolutions en terme de placement, de granularité et d'état des entités logicielles de définition. Les changements de plans sont déclenchés par interruption :

- **logicielle** (axe langage) : cette interruption est réalisée au niveau applicatif. C'est le programmeur, en définissant la précédence entre les plans qui indique les conditions de changement de phase. **Cette technique correspond à réaliser des points d'arrêt dans l'application pour synchroniser l'architecture de définition et l'architecture d'exécution.**
- **matérielle** (axe système) : le changement de plan est conditionné par les migrations d'entités logicielles d'exécution pour des raisons systèmes. Sont alors prises en compte les techniques d'équilibrage de la charge et de réduction de coûts de communication. **On répercute alors les changements ayant des causes non applicatives, ce qui a un intérêt pour le concepteur de l'application qui peut ainsi tester l'architecture matérielle la mieux adaptée à son application.**

Le passage d'un plan à un autre entraîne donc une synchronisation entre l'architecture de définition et d'exécution. Tout changement de plan est régi par **un contrat imposant le**

**respect de POST conditions**. Le contrat sert à la validation du plan courant et le changement de plan n'est autorisé que si le contrat est vérifié. Cette notion de contrat est empruntée au langage Eiffel [Meyer 88]. Dans Eiffel, si un client qui invoque une méthode M remplit des pré-conditions (clause *require*) avant l’appel de M, alors il est assuré que toutes les post-conditions qui ont été définies seront remplies après l’exécution de M (clause *ensure*). Il en est de même pour le changement de plan qui n’est possible et garanti que si le contrat est respecté.

Un contrat d’exécution indique par exemple :

- un contexte d’exécution à respecter (type de système d’exploitation, utilisation d'une unité de calcul flottante) ;
- les types de traces à collecter ;
- les contraintes de placement à respecter (aucune ou respect de l’architecture de définition).

La notion de plans et de contrats d’exécution servent :

- *à la répartition de charge* : le plan pilote la répartition de charge en respectant les contraintes de la politique de transfert de charge. Au moyen des entités logicielles d’exécution, les coûts de communication ou les migrations éventuelles sont transférés du gestionnaire de placement vers le gestionnaire de plan. Celui-ci vérifie alors que les changements sont compatibles avec le contrat qu’il gère. Si le contrat est vérifié, l’accord est donné au répartiteur de charge d’effectuer les changements et un changement de plan est réalisé. Sinon, l’exécution doit continuer. Dans ce cas, ce sont les contraintes logicielles qui contraignent les évolutions et pas les critères de partage de charge (liée à la sous utilisation ou à la sur utilisation de sites).
- *à la modélisation de la précédence* : un outil de répartition de charge peut tirer avantage de la connaissance de la précédence des programmes qui composent une application. La représentation de la précédence consiste alors à composer un ensemble de plans. L'ensemble des processus à lancer est contenu dans les entités logicielles, elles mêmes contenues dans un plan. Ainsi chaque plan possède une liste de plans suivants correspondant à l'ensemble des processus à exécuter après la fin des processus du ou des plans courants. Le contrat entre les plans est alors la fin de tous les processus du plan courant.
- *au suivi dynamique de la granularité* : dans la plupart des cas, le calcul de la granularité est déjà effectué avant le lancement de l'application. Mais celui-ci peut se révéler inadapté puisque calculé uniquement sur les spécifications de l'application. Comme il ne tient pas compte de l'état actuel de l’environnement d’exécution à savoir, la charge des machines, la charge du réseau, les contraintes de migration éventuelles d'entités logiques non détectées avant le calcul. Ce calcul effectué n’est qu’à priori et une modification de la granularité peut être nécessaire si des contraintes matérielles apparaissent.

Si la description et la gestion des plans d’exécution n’est pas difficile, il n’en est pas de même de l’élaboration du contrat d’exécution. On se pose alors le problème de décrire des contraintes complexes et les évaluer en temps réel. A cela s’ajoute le fait que ces contraintes s’appliquent dans des cadres différents d’équilibrage et dans des environnements variés. La solution à ce problème consiste peut être à s’appuyer sur des technologies médiateurs adaptés à la granularité des composants logiciels et à leur environnement d’exécution. Ainsi, si les applications sont constituées d’une grand nombre de composants logiciels mobiles à grain fin, on utilise plutôt des bus logiciels (de type CORBA) et des médiateurs de type MOM (gestion de file d’attente locale et/ou distribuée). Alors que si on travaille avec des application monolithiques, construites avec des composants logiciels à gros grain et à durée de vie longue, on utilise plutôt des plate-formes d’intégration ou des systèmes ouverts via des médiateurs lourds (gestionnaire de transaction, de configuration, de sécurité, etc.).

VODEL-D étant un langage devant gérer dynamiquement l'évolution des composants logiciels sur une architecture matérielle, il nous a semblé judicieux d'y intégrer les entités de description de l'architecture matérielle. Il est ainsi plus facile, en utilisant des relations de dépendance, de représenter le placement des composants logiciels sur l'architecture matérielle.

## 2.5. Intégration d'éléments de description de HADEL dans VODEL-D

Après avoir défini VODEL-D, il nous a semblé intéressant d'intégrer certains éléments de la description d'architecture matérielle, faite en HADEL. Notre intérêt était triple. D'abord, cette intégration ne nécessitait que peu de modifications des classes existantes. Ensuite, cela permettait de travailler sur un graphe unique décrivant à la fois logicielle et son placement sur une architecture matérielle. Enfin, l'avantage de la modélisation des composantes externes à l'architecture logicielle est qu'elle améliore la visibilité du concepteur d'application. Ainsi, il n'est plus possible qu'un serveur ou qu'un processus invoqué pour l'exécution de l'application n'apparaisse pas dans la modélisation (plus aucun résidu algorithmique fantôme n'est à déplorer).

**Ainsi, toute entité ou objet invoqué par l'application est mentionné dans la description de celle-ci. Ces entités sont appelées dans VODEL-D des entités passives physiques.**

### 2.5.1. *Les entités passives physiques*

La description matérielle dans VODEL consiste à représenter les entités passives physiques présentes sur le site où s'exécutera l'application répartie. Un site est composé de processeurs, de mémoires et de support de stockage. Ces trois composantes sont regroupées dans la classe **machine (décrivant l'entité H-Machine définie dans Hadel). Ces composantes sont déductibles d'une description HADEL, dans le sens ou elles en sont un sous ensemble.**

La classe machine est soit concrète soit abstraite :

- *la classe concrète machine* représente une machine réelle. Deux champs sont présents dans la classe : la vitesse de communication entre deux processus (communication locale) et la vitesse d'accès disque (bus disque).
- *la classe abstraite machine* représente des machines virtuelles qui ne correspondent pas à des machines réelles. Elles existent pour regrouper des machines réelles (et/ou virtuelles). Les machines virtuelles permettent également de modéliser la topologie d'interconnexion entre les machines. Le champ vitesse de communication présent dans la classe, définit dans ce cas la vitesse de communication entre les classes filles.

**La classe CPU** (processeur) est un raffinement de la classe machine. Elle définit les caractéristiques d'un CPU machine. Les propriétés prédéfinies correspondent aux informations servant aux systèmes sous-jacents. On peut ainsi retrouver la vitesse du processeur, sa faculté à répondre aux exigences des calculs scientifiques, ainsi que le temps du changement de contexte. Le CPU est une entité passive physique, dont la relation de dépendance indique sa localisation sur une machine.

**La classe Disque** définit les unités de stockage accessibles sur le réseau. Le champ temps d'accès donne la vitesse en milliseconde du disque. La classe disque peut dépendre d'une classe machine virtuelle ou concrète. Dépendre d'une classe machine virtuelle signifie que le disque est monté virtuellement sur toutes les classes machines concrètes dépendant de la classe machine virtuelle. Dans le cas inverse, cette dépendance indique que le disque est monté localement et que seule la machine concrète père peut accéder aux données présentes sur le disque. Les données d'un disque sont regroupées dans un objet persistant modélisant des entités passives logiques.

Le prochain paragraphe présente les extensions que nous avons réalisé au sein des entités passives logiques lors de l'intégration du langage HADEL dans VODEL-D.

### 2.5.2. *Extension des entités passives logiques*

Certaines ressources matérielles peuvent aussi être considérées comme des entités passives logiques. Ainsi, par exemple, deux types d'entités mémoires ont été définies :

- la mémoire physique d'une machine composée de deux valeurs : sa taille et sa vitesse d'accès. Cet objet est du type **entité passive physique** ;
- la mémoire virtuelle est du type **entité passive logique**. On définit alors un objet particulier nommé *root* qui définit les besoins en mémoire du système. Ces besoins caractérisent la mémoire virtuelle maximale du système. Les objets dépendant de cette classe indique la mémoire nécessaire au fonctionnement correct des processus de l'application.

Nous présentons ci-dessous un exemple de description d'architecture matérielle.

### 2.5.3. *Exemple de description d'architecture matérielle*

Une description d'un réseau de machines est donnée par la Figure 27. Il est composé de trois stations de travail connectées entre elles par un câble Ethernet. Dans ce réseau, deux stations accèdent au disque 1. Celui-ci est présent logiquement sur chacune des deux stations (en utilisant la commande *mount* du système de gestion de fichiers NFS).

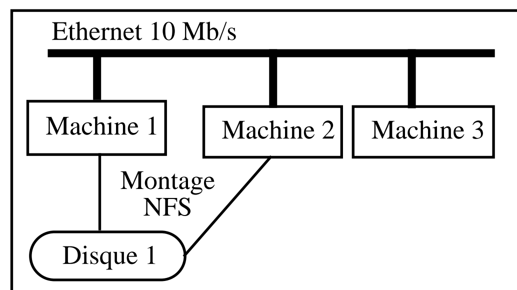



**Figure 27 :** Exemple d'un réseau local comprenant trois machines et un disque partagé

Supposons que par la suite, le concepteur souhaite spécifier un groupe de machines représentant celles qui ont accès au disque 1. Il définit dans un premier temps les classes concrètes en spécifiant les temps d'accès disque, puis décrit le groupe souhaité en concevant la machine virtuelle 1 (cf. Figure 28). La machine virtuelle 2 correspond à définir un groupe de machines qui communiquent via le réseau Ethernet.

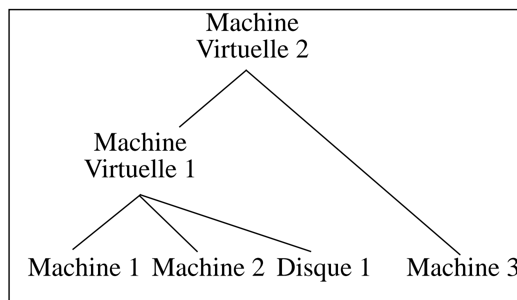



**Figure 28 :** Représentation d'un réseau via des entités passives issues de la classe machine

### 2.5.4. *Exemple de description d'une architecture basée uniquement sur des entités passives*

Sur la Figure 29 est décrite une architecture logicielle composée d'entités passives. Tous les fichiers dépendent d'une entité disque. Cette entité est une entité physique attachée à une entité de la classe machine. Cet attachement définit la machine sur laquelle le disque

est monté. Le fichier 2 est présent sur deux disques parce qu'il a été dupliqué. La modélisation du système de fichier NFS se réalise en attachant les objets disques sur une ou plusieurs machines virtuelles, et non pas en attachant un fichier sur plusieurs disques. Les objets codes dépendent d'un objet CPU (ce qui détermine sur quel type de processeur le code peut être exécuté) et d'un objet fichier (ce qui localise le fichier contenant le code). Chaque CPU dépend d'au moins une machine, ce qui détermine sa localisation sur le réseau.

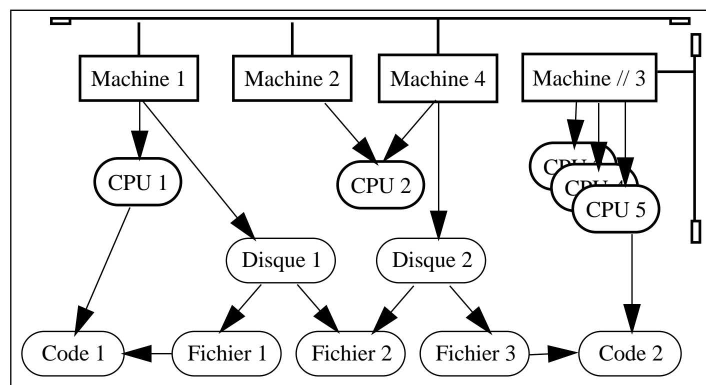



**Figure 29 :** Architecture logicielle composée de ressources passives

#### 2.5.5. *Synthèse*

La description des H-machines dans VODEL-D est réalisée via la classe Machine. Les H-liens sont décrits dans VODEL-D via un même groupe de machines possédant des caractéristiques communes tel que le partage d'un lien de communication.

Une grande partie des attributs des objets d'HADEL ont été transférés dans VODEL-D. Ainsi, l'adresse, le nom, le système d'exploitation, la vitesse de lecture et l'indice de charge sont présents dans la classe Machine. Le type de machine et les champs correspondant à la spécification du processeur se retrouvent dans la classe CPU. La vitesse du disque est présente dans la classe Disque. La taille mémoire est présente dans la classe Mémoire. Quant au nombre de processeurs, il se retrouve en comptabilisant les objets de type CPU dépendant de la classe machine.

**En conclusion, nous pouvons dire qu'aussi bien au niveau gros grain qu'au niveau grain fin, tous les attributs définis dans HADEL se retrouvent dans les différentes classes VODEL-D.**

### 2.6. Synthèse

Pour faire face à des contraintes de réactivité dynamique de notre environnement, nous avons découplé la partie statique de la description de l'application (qu'on peut considérer comme un placement initial statique) de son évolution dynamique et des changements que cela entraîne sur l'architecture logicielle de l'application. Ainsi, toute modification de la granularité de l'application durant son exécution entraîne une modification de l'architecture logicielle d'exécution et pas de celle de définition. Les plans d'exécution jouent alors un double rôle :

1) ils sont utilisés comme des gardes fous pour éviter que la répartition dynamique de charge ne remette en question l'architecture de l'application que le concepteur désire tester telle quelle ;
2) ils sont utilisés pour prévoir à l'avance les états stables de l'application et d'imposer des contraintes de continuation (ou simplement de synchronisation) à la fois matérielles (attente d'une machine libre puissante pour l'exécution du plan suivant) et logicielles (points de reprise ou de déboggage par exemple).

Une fois l'exécution terminée, il est possible d'indiquer à l'utilisateur toutes les modifications d'architecture logicielle ou matérielle (grâce à l'intégration d'HADEL) qui ont eu lieu et les décisions qui ont été prises pour les surmonter.

Enfin, la Figure 30 résume les principales entités définies dans Vodel-D et la distinction entre architecture logicielle de définition et architecture logicielle d'exécution.

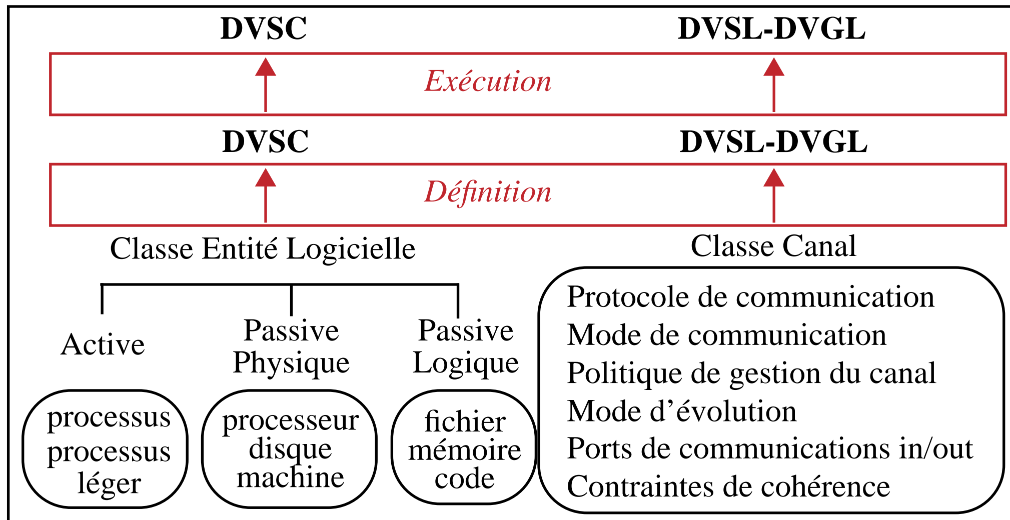



**Figure 30 :** Les principales entités définies dans VODEL-D

## 3. Un exemple de description d'une application répartie

Une application répartie AR est constituée de cinq programmes A, B, C, D, E qui réalisent un calcul réparti (cf. Figure 31).

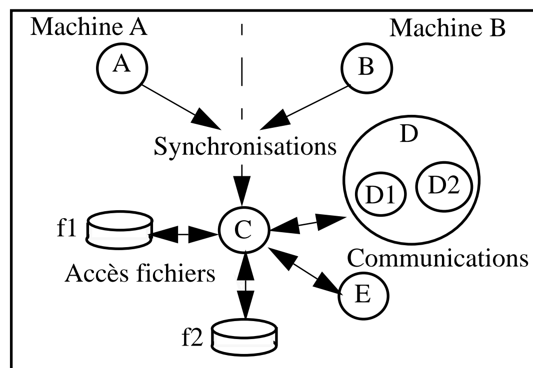



**Figure 31 :** Un exemple de programme réparti

Les contraintes de l'exécution sont les suivantes :

- A et B s'exécutent avant C, D et E ;
- C, D et E s'exécutent après la terminaison de A et B ;
- A se synchronise avec B et ils terminent leur exécution ;
- A et B ne s'exécutent pas sur la même machine, mais sur le même type de machine (une SPARC 10) ;
- C et E doivent s'exécuter sur la même machine et communiquer de manière asynchrone. C et D doivent s'exécuter sur des machines distinctes.

- D est en fait composé de D1 et de D2, deux programmes indépendants susceptibles de migrer en cours d'exécution ;
- C envoie des messages à D1 et D2, mais eux n'envoient pas de messages ;
- C ne peut migrer de la machine sur laquelle il est lancé ;
- C lit dans un fichier F1 et produit un fichier F2 qui sont situés sur un disque local nommé Disque 1.

## 3.1. Description des entités passives physiques et logiques

Supposons que l'architecture matérielle support de l'exécution soit constituée d'un réseau Ethernet à 10 Mbits/s, sur lequel sont connectées trois machines SPARC 10. Ce réseau, nommée RESEAU_SPARC_10, est modélisé par l'intermédiaire d'une machine virtuelle, définie avec une classe abstraite :

```
ABSTRAITE ENT_PAS_PHY ClassMach RESEAU_SPARC_10
VITESSE_ACCES_DISQUE INDEFINI  -- pas de disque partagé sur le réseau
VITESSE_COMMUNICATION 10       -- 10 Mbits car réseau Ethernet
FIN ClassMach
```

Puis, nous définissons la classe abstraite machine SPARC_10 qui modélise les types de machines où auront lieu l'exécution :

```
ABSTRAITE ENT_PAS_PHY ClassMach SPARC_10
DEPEND RESEAU_SPARC_10            -- cette machine appartient à ce réseau
VITESSE_ACCES_DISQUE 20           -- 20 ms
VITESSE_COMMUNICATION INDEFINI    -- vitesse de changement de contexte inconnue
FIN ClassMach
```

Le processeur de la machine SPARC_10 est appelé PROC_SPARC_10 et est décrit de la manière suivante :

```
CONCRETE ENT_PAS_PHY ClassCPU PROC_SPARC_10
DEPEND SPARC_10     -- c'est le processeur de la machine PROC_SPARC_10
FAMILLE SPARC       -- famille de processeur
TYPE SPARC10        -- nom du processeur
MIPS 100            -- puissance donnée en exemple
TAILLE_CACHE 64     -- 64 Ko
FPU OUI             -- unité de calcul flottant présente
TPS_CONTEXT 10      -- temps moyen du changement de contexte = 10 ms
FIN ClassCPU
```

Les descriptions des machines de type SPARC_10, sont similaires à celle de SPARC_10_1 présentée ci-dessous :

```
CONCRETE ENT_PAS_PHY ClassMach SPARC_10_1
DEPEND SPARC_10        -- elle dépend de la classe abstraite SPARC_10
FIN SPARC_10_1
```

Le disque DISQUE_1, lié à la machine SPARC_10_1 sur lequel le programme C va lire et écrire des fichiers est décrit par une classe concrète Disque :

```
CONCRETE ENT_PAS_PHY ClassDisque DISQUE_1
DEPEND SPARC_10_1      -- c'est le disque de la machine SPARC_10_1
TAILLE 2000            -- 2 Go
TEMPS_ACCES 20         -- 20 ms
FIN ClassDisque
```

Enfin, nous indiquons que les deux entités passive logiques fichiers F1 et F2 sont situées sur le DISQUE 1 :

```
CONCRETE ENT_PAS_LOG ClassFichier F1 -- concrète : le fichier existe déjà
DEPEND DISQUE_1        -- F1 est sur le disque 1
NOM_FICHIER F1         -- identifiant logique du fichier
CHEMIN ~william/ar     -- identifiant logique de localisation
MODE TypeMode L        -- accès au fichier en lecture
FIN ClassFichier
```

```
ABSTRAITE ENT_PAS_LOG ClassFichier F2 -- abstraite : le fichier n'existe pas
DEPEND DISQUE_1        -- F2 sera sur le disque 2
NOM_FICHIER F2         -- identifiant logique du fichier
CHEMIN ~william/ar     -- identifiant logique de localisation
MODE TypeMode E        -- accès au fichier en écriture
FIN ClassFichier
```

La Figure 32 résume le graphe de dépendance des classes des entités passives créées.

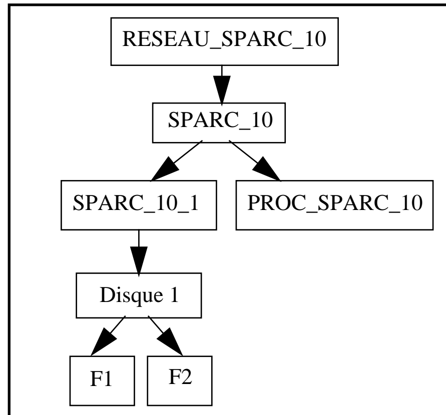



**Figure 32 :** Description des entités passives et physiques de l'application AR

## 3.2. Description des entités actives

L'application AR est composée de sept entités actives : A, B, C, E, D, D1, D2. Les quatre premières ont une description similaires, c'est pourquoi, nous n'en décrivons qu'une.

```
CONCRETE ENT_ACT ClassE_A A    -- A est une entité active concrète
IN IN_A             -- nom du descripteur en entrée
OUT OUT_A           -- nom du descripteur en sortie
MEMOIRE INDEFINI    -- besoins en mémoire (ici inconnus)
CODE CODE_A         -- l'entité CODE_A indique où est localisé le code
FIN ClassE_A
```

Décrivons maintenant les entités D, D1 et D2. D1 et D2 sont des entités actives concrètes contenues dans une entité active abstraite D. D est en fait un composant logiciel conteneur de D1 et D2. Il communique virtuellement avec C, car en fait ce sont D1 et D2 qui reçoivent des messages de C.

```
ABSTRAITE ENT_ACT ClassE_A D   -- D est une entité active abstraite
IN IN_D             -- nom du descripteur en entrée
OUT OUT_D           -- nom du descripteur en sortie
MEMOIRE INDEFINI    -- besoins en mémoire (ici inconnus)
CODE CODE_D         -- l'entité CODE_D indique où est localisé le code de D
FIN ClassE_A

CONCRETE ENT_ACT ClassE_A D1   -- D1 est une entité active concrète
IN IN_D1=D.IN_D     -- nom du descripteur en entrée de D
OUT OUT_D1=D.OUT_D  -- nom du descripteur en sortie de D
MEMOIRE INDEFINI    -- besoins en mémoire (ici inconnus)
CODE CODE_D1        -- l'entité CODE_D1 indique où est localisé le code de D1
FIN ClassE_A

CONCRETE ENT_ACT ClassE_A D2   -- D2 est une entité active concrète
IN IN_D2=D.IN_D     -- nom du descripteur en entrée de D
OUT OUT_D2=D.OUT_D  -- nom du descripteur en sortie de D
MEMOIRE INDEFINI    -- besoins en mémoire (ici inconnus)
CODE CODE_D2        -- l'entité CODE_D2 indique où est localisé le code de D2
FIN ClassE_A
```

## 3.3. Description des entités logicielles de définition

Dans notre exemple, nous avons indiqué que l'entité active D contenait deux entités actives D1 et D2, susceptibles de migrer lors de l'exécution. L'entité logicielle contenant D, nommée ELD_D, est :

- une entité active : elle est composée de deux processus ;
- libre : elle peut migrer en cours d'exécution ;
- dissociable : lors de l'exécution D peut être scindée en D1 et D2 ;
- migrable : une fois dissociées, les entités contenues dans D sont migrables ;
- ELD_D contient : deux entités logicielles de définition ELD_D1 et ELD_D2 ;
- ELD_D attache : aucun lien d'attachement n'est défini pour ELD_D;
- ELD_D repousse : le programme D doit s'exécuter sur un site distinct de C, c'est pourquoi les deux entités logicielles ELD_C et ELD_D se repoussent ;
- ELD_D importe : l'entité active D, le descripteur de ses canaux de communication et son entité d'initialisation.

La description de l'entité logicielle de définition EL_D est donc la suivante :

```
EL DEFINITION ELD_D
TYPE ACTIF
EVOLUTION LIBRE
REGROUPE DISSOCIABLE
REPARTITION REPLICABLE
ATTACHE NON
REPOUSSE ELD_C
CONTIENT ELD_D1, ELD_D2
IMPORTE (D, [IN_D, OUT_D], Init_D) -- entité logicielle, canaux et initialisation
FIN EL
```

L'entité logicielle de définition intégrant C, nommée ELD_C est :

- une entité active : elle est composée d'un processus ;
- fixe : elle ne peut changer de machine une fois l'exécution lancée ;
- indissociable : ELD_C ne peut pas être scindée ;
- centralisée : car ELD_C est une entité fixe ;
- ELD_C attache ELD_E car le programme E doit s'exécuter sur le même site que C. C est aussi attachée à deux entités de définition passives ELD_F1 et ELD_F2 qui correspondent aux deux fichiers qui sont manipulés par C ;
- ELD_C repousse : le programme D doit s'exécuter sur un site distinct de C, c'est pourquoi les deux entités logicielles ELD_C et ELD_D se repoussent ;
- ELD_C contient : aucune entité ;
- ELD_C importe : l'entité logicielle de définition ELD_C importe l'entité active C.

La description de l'entité logicielle ELD_C est alors la suivante :

```
EL DEFINITION ELD_C
TYPE ACTIF
EVOLUTION FIXE
REGROUPE INDISSOCIABLE
REPARTITION NON
ATTACHE ELD_E, ELD_F1, ELD_F2
REPOUSSE ELD_D
CONTIENT NON
IMPORTE (C, [IN_C, OUT_C], Init_C) -- C gère deux canaux
FIN EL
```

Décrivons maintenant l'entité de définition ELD_F1 :

- une entité passive : elle ne contient pas de processus ;
- fixe : elle ne peut changer de machine une fois l'exécution lancée ;
- indissociable : ELD_F1 ne peut pas être scindée ;
- centralisée : car ELD_F1 est une entité fixe ;
- lien d'attachement : ELD_F1 est attachée au disque DISQUE_1.
- importe : ELD_F1 importe l'entité passive F1.

La description de l'entité logicielle ELD_F1 est alors la suivante :

```
EL DEFINITION ELD_F1
TYPE PASSIVE
EVOLUTION FIXE
REGROUPE INDISSOCIABLE
REPARTITION NON
ATTACHE DISQUE_1
REPOUSSE NON
CONTIENT NON
IMPORTE F1                      -- pas d'initialisation
FIN EL
```

Il en est de même pour ELD_F2 qui dispose un lien d'attachement à ELD_C, car c'est le programme C qui va la créer.

```
EL DEFINITION ELD_F2
TYPE PASSIVE
EVOLUTION FIXE
REGROUPE INDISSOCIABLE
REPARTITION NON
ATTACHE DISQUE_1, ELD_C
REPOUSSE NON
CONTIENT NON
IMPORTE F2                      -- pas d'initialisation
FIN EL
```

## 3.4. Descriptions des liens de communication entre entités logicielles de définition

A et B se synchronisent et terminent leur exécution. Pour décrire le lien de communication qui les relie, on doit d'abord décrire les entités logicielles ELD_A et ELD_B. Pour cela, on utilise la classe canal, en indiquant que le flux d'information entre ELD_A et ELD_B est synchrone et est appelé CANAL_SYNCHRONE.

```
CONCRETE ClassCanal CANAL_SYNCHRONE
IN CANAL_SYNC_IN          -- descripteur des données en entrée du canal
OUT CANAL_SYNC_OUT        -- descripteur des données en sortie du canal
CAPACITE INDEFINI         -- capacité du canal inconnue
TYPE_COM SYNCHRONE        -- propriété de la communication du canal
ADAPTE NON                -- alignement des données inutile (même type de machine)
GESTION FIFO              -- gestion de l'ordonnancement des messages
EVOLUTION AUCUNE          -- évolution du canal lors de l'exécution
ACCES_EP AUCUN            -- mode d'accès aux entités passives
FIN ClassCanal
```

Puis on définit le DVSL qui relie A et B :

```
CONCRETE DVSLClass A_B
EL1 ELD_A                 -- entité active source
LIEN CANAL_SYNCHRONE      -- canal
EL2 ELD_B                 -- entité active destination
DIRECTION BI              -- communication bi-directionnelle
POIDS INDEFINI            -- poids non défini
FIN DVSLClass
```

Le canal asynchrone qui relie C et E est défini de la même manière, à la différence prêt que la réception des messages est non ordonnée (mot clef RAND) :

```
CONCRETE ClassCanal CANAL_ASYNCHRONE
IN CANAL_ASYNC_IN         -- descripteur des données en entrée du canal
OUT CANAL_ASYNC_OUT       -- descripteur des données en sortie du canal
CAPACITE INDEFINI         -- capacité du canal inconnue
TYPE_COM ASYNCHRONE       -- propriété de la communication du canal
ADAPTE NON                -- alignement des données inutile (même type de machine)
GESTION RAND              -- la réception des messages n'est pas ordonnée
EVOLUTION AUCUNE          -- évolution du canal lors de l'exécution
ACCES_EP AUCUN            -- mode d'accès aux entités passives
FIN ClassCanal
```

Le DVSL entre ELD_C et ELD_E est alors défini de la manière suivante :

```
CONCRETE DVSLClass C_E
EL1 ELD_C                 -- entité active source
LIEN CANAL_ASYNCHRONE     -- canal
EL2 ELD_E                 -- entité active destination
DIRECTION BI              -- communication bi-directionnelle
POIDS INDEFINI            -- poids non défini
FIN DVSLClass
```

Nous décrivons maintenant le DVSC reliant C à F1, en recréant une nouvelle classe canal. La classe ACCES_FICHIER_L gère des tampons d'entrées-sorties de 16Ko, des communications synchrones et n'offre que l'accès en mode lecture aux entités passives :

```
CONCRETE ClassCanal ACCES_FICHIER_L
IN CANAL_FIC_IN           -- descripteur des données en entrée du canal
OUT CANAL_FIC_OUT         -- descripteur des données en sortie du canal
CAPACITE 16               -- 16 KO
TYPE_COM SYNCHRONE        -- propriété de la communication du canal
ADAPTE NON                -- alignement des données inutile (même type de machine)
GESTION FIFO              -- la réception des messages est ordonnée
EVOLUTION AUCUNE          -- évolution du canal lors de l'exécution
ACCES_EP L                -- mode d'accès aux entités passives
FIN ClassCanal
```

Le DVSC reliant C à F1 utilise ensuite le canal de type ACCES_FICHIER_L :

```
CONCRETE DVSLClass C_F1
EL1 ELD_C                 -- entité active source
LIEN ACCES_FICHIER_L      -- canal
EL2 ELD_E1                -- entité active destination
DIRECTION ELD_F1          -- communication DE C vers F1 uniquement
POIDS INDEFINI            -- poids non défini
FIN DVSLClass
```

Enfin, nous présentons un exemple de la combinaison de description de DVGL et de DVSL. Les communication entre C et D, C et D1 et C et D2 utilisent deux DVSL et un DVGL. Comme D peut-être scindé en deux composants logiciels migrables (D1 et D2), alors il faut redéfinir une classe canal concrète, appelée CANAL_ASYNCHRONE_M :

```
CONCRETE ClassCanal CANAL_ASYNCHRONE_M
IN CANAL_ASYNC_M_IN       -- descripteur des données en entrée du canal
OUT CANAL_ASYNC_M_OUT     -- descripteur des données en sortie du canal
CAPACITE INDEFINI         -- capacité du canal inconnue
TYPE_COM ASYNCHRONE       -- propriété de la communication du canal
ADAPTE NON                -- alignement des données inutile (même type de machine)
GESTION FIFO              -- la réception des messages n'est pas ordonnée
EVOLUTION MIGRABLE        -- le canal peut migrer avec D1 ou D2 lors de l'exécution
ACCES_EP AUCUN            -- mode d'accès aux entités passives
FIN ClassCanal
```

Si on suppose que C envoie des messages distincts à D1 et D2, alors, la description des DVSC et du DVGL impliqués est la suivante (cf. Figure 33) :

```
CONCRETE DVSLClass C_D1
EL1 ELD_C                 -- entité active source
LIEN ACCES_ASYNCHRONE_M   -- canal
EL2 ELD_D1                -- entité active destination
DIRECTION EL12            -- communication de c vers D1
POIDS INDEFINI            -- poids non défini
FIN DVSLClass

CONCRETE DVSLClass C_D2
EL1 ELD_C                 -- entité active source
LIEN ACCES_ASYNCHRONE_M   -- canal
EL2 ELD_D2                -- entité active destination
DIRECTION EL12            -- communication de c vers D2
POIDS INDEFINI            -- poids non défini
FIN DVSLClass

ABSTRAITE DVGLClass C_D
IMPORTE C_D1, C_D2        -- DVSL intégrés
CONTIENT AUCUN            -- DVGL intégrés
POIDS INDEFINI
FIN DVGLClass
```

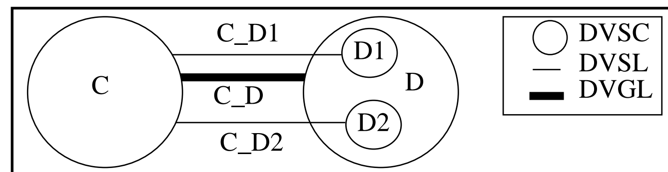



**Figure 33 :** DVSL et DVGL reliant C, D, D1 et D2

## 3.5. Description des entités logicielles d'exécution

Pour donner un exemple d'utilisation des entités logicielles d'exécution, nous avons choisi de décrire les interactions entre les entités d'exécution ELE_C, ELE_D, ELE_D1 et ELE_D2. Chaque entité logicielle d'exécution contient des informations concernant son initialisation, son placement, sa charge et la fréquence de ces communications :

- ELE_C et ELE_D étant en communication, on décide de leur affecter un même indice de groupe.
- ELE_D contenant deux processus possède un indice de charge de 2 et ELE_C de 1.
- comme ELE_D et ELE_C se repoussent, elles ne seront pas placées sur la même machine, à l'inverse de EL_D, ELD_D1 et ELD_D2 qui seront sur la même machine au démarrage de l'exécution. EL_D étant dissociable et migrable, son évolution en cours d'exécution est par contre non prévisible.

La description de ces entités logicielles d'exécution est donnée ci-dessous :

```
EL EXECUTION ELE_C
GROUPE 1                  -- numéro du groupe
CHARGE 1                  -- indice de charge
LIAISON UNKNOWN           -- indice de liaison
PLACEMENT SPARC_10_1      -- placement sur la machine SPARC_10 numéro 1
MODIF_ELD NON             -- pas de modification par rapport à ELE_C
EVOLUTION FIXE            -- pas d'évolution possible
REGROUPE INDISSOCIABLE    -- impossible de casser ELE_C en d'autres entités
REPARTITION NON           -- car pas de mouvement
ATTACHE ELE_E, ELE_F1, ELE_F2
REPOUSSE ELE_D
CONTIENT NON
IMPORTE (C, [IN_C, OUT_C], Init_C)
FIN EL
```

```
EL EXECUTION ELE_D1       -- c'est la même description pour ELE_D2
GROUPE 1                  -- numéro du groupe
CHARGE 1                  -- indice de charge
LIAISON UNKNOWN           -- indice de liaison
PLACEMENT SPARC_10_2      -- placement sur la machine SPARC_10 numéro 2
MODIF_ELD NON             -- pas de modification par rapport à ELE_D1
EVOLUTION LIBRE           -- évolution libre lors de l'exécution
REGROUPE INDISSOCIABLE    -- impossible de casser ELE_D1
REPARTITION MIGRATION     -- la migration est possible
ATTACHE NON               -- pas de lien d'attachement
REPOUSSE ELE_C            -- impossible d'être sur la même machine que ELE_C
CONTIENT NON              -- pas de hiérarchie
IMPORTE (D1, [IN_D1, OUT_D1], Init_D1) -- importe la description de l'entité active
FIN EL

EL EXECUTION ELE_D
GROUPE 1                  -- numéro du groupe
CHARGE 2                  -- indice de charge
LIAISON UNKNOWN           -- indice de liaison
PLACEMENT SPARC_10_2      -- placement sur la machine SPARC_10 numéro 2
MODIF_ELD NON             -- pas de modification par rapport à ELE_D
EVOLUTION LIBRE           -- évolution libre lors de l'exécution
REGROUPE DISSOCIABLE      -- possible de casser ELE_D en ELE-D1 et ELE_D2
REPARTITION MIGRATION     -- la migration de ELE_D est possible
ATTACHE NON               -- pas de lien d'attachement
REPOUSSE ELE_C            -- impossible d'être sur la même machine que ELE_C
CONTIENT ELE_D1, ELE_D2   -- un niveau de hiérarchie
IMPORTE (D, [IN_D, OUT_D], Init_D) -- importe la description de l'entité active D
FIN EL
```

## 3.6. Description des plans d'exécution

A et B doivent se terminer avant que C puisse commencer. Pour exprimer cela, nous allons utiliser deux plans d'exécution : P1 et P2.

Dans P1, nous indiquons que les entités logicielles d'exécution ELE_A et ELE_B doivent s'exécuter ensemble. Le passage à P2 ne sera possible que lorsque tous les processus de P1 se seront terminés. Le contrat d'exécution à respecter est l'attente de la terminaison de tous les processus (contrat nommé TERMINAISON_PROCESSUS). Le passage au plan P2 lancera l'exécution des processus C, D et E. Il ne sera pas suivi d'autres plans, c'est pourquoi, nous indiquons que le processus suivant est FIN. Enfin, comme pour P1, le contrat d'exécution consiste à attendre la fin d'exécution de tous les processus. La description des plans P1 et P2 est donnée ci-dessous :

```
PLAN P1
SUIVANTS P2
ENTITES ELE_A, ELE_B
CONTRAT TERMINAISON_PROCESSUS
FIN PLAN

PLAN P2
SUIVANTS FIN
ENTITES ELE_C, ELE_D, ELE_E
CONTRAT TERMINAISON_PROCESSUS
FIN PLAN
```

## 3.7. Analyse près exécution

Supposons que lors de l'exécution, une surcharge apparaisse sur la machine SPARC_10_2. ELE_D étant dissociable et ayant un indice de charge égal à 2, la décision est alors prise de le scinder en deux et de migrer ELE_D2 (en laissant ELE_D1 sur la même machine). ELE_D et ELD_D2 sont donc modifiées et positionnent leur champs MODIF_ELD à OUI. ELE_D2 est migrée sur la machine SPARC_10_3, car elle ne peut pas aller sur la machine SPARC_10_1, où ELE_C, qui s'y trouve, la repousse. L'indice

de charge de ELE_D passe alors à 1. La description des entités logicielles d'exécution est alors :

```
EL EXECUTION ELE_D1
GROUPE 1                  -- numéro du groupe
CHARGE 1                  -- indice de charge
LIAISON UNKNOWN           -- indice de liaison
PLACEMENT SPARC_10_2      -- placement sur la machine SPARC_10 numéro 2
MODIF_ELD NON             -- pas de modification par rapport à ELE_D1
EVOLUTION LIBRE           -- évolution libre lors de l'exécution
REGROUPE INDISSOCIABLE    -- impossible de casser ELE_D1
REPARTITION MIGRATION     -- la migration est possible
ATTACHE NON
REPOUSSE ELE_C
CONTIENT NON
IMPORTE (D1, [IN_D1, OUT_D1], Init_D1)
FIN EL

EL EXECUTION ELE_D2
GROUPE 1                  -- numéro du groupe
CHARGE 1                  -- indice de charge
LIAISON UNKNOWN           -- indice de liaison
PLACEMENT SPARC_10_3      -- placement sur la machine SPARC_10 numéro 3
MODIF_ELD OUI RAISON EQU_CHARGE -- modification à cause de l'équilibrage de charge
EVOLUTION LIBRE           -- évolution libre lors de l'exécution
REGROUPE INDISSOCIABLE    -- impossible de casser ELE_D2
REPARTITION MIGRATION     -- la migration est possible
ATTACHE NON
REPOUSSE ELE_C
CONTIENT NON
IMPORTE (D2, [IN_D2, OUT_D2], Init_D2)
FIN EL

EL EXECUTION ELE_D
GROUPE 1                  -- numéro du groupe
CHARGE 1                  -- indice de charge
LIAISON UNKNOWN           -- indice de liaison
PLACEMENT SPARC_10_2      -- placement sur la machine SPARC_10 numéro 2
MODIF_ELD OUI RAISON EQU_CHARGE -- modification à cause de l'équilibrage de charge
EVOLUTION LIBRE           -- évolution libre lors de l'exécution
REGROUPE DISSOCIABLE      -- possible de casser ELE_D en ELE-D1 et ELE_D2
REPARTITION MIGRATION     -- la migration de ELE_D est possible
ATTACHE NON
REPOUSSE ELE_C
CONTIENT ELE_D1
IMPORTE (D, [IN_D, OUT_D], Init_D)
FIN EL
```

## 3.8. Discussion

De cet exemple, nous pouvons déduire que :

1) dans VODEL-D, les classes abstraites sont en fait des gabarits de description d'une architecture logicielle et matérielle. Les classes concrètes correspondent par contre aux squelettes d'implémentation et de déploiement ;
2) VODEL-D n'étant pas un langage de programmation ou de configuration, on évite ainsi de gérer l'évolution dynamique du canal reliant ELE_C à ELE_D2, lors de la migration de ELE_D2. Par contre, on prévoit le fait que cela puisse arriver ;
3) si un des deux composants logiciels relié par un DVSL est dissociable et migrable, alors le DVSL qui les relie à son champ évolution qui est obligatoirement positionné à migrable ;
4) avec les entités logicielles d'exécution, on détecte les composants logiciels qui ont évolués, comment ils ont évolué et pour quelle(s) raison(s). Ces informations sont

ensuite utiles au concepteur de l'application pour modifier son architecture matérielle et/ou son architecture logicielle

5) les contrats d'exécution définis dans les plans d'exécution P1 et P2 (nommés FIN et TERMINAISON_PROCESSUS) ont servi à modéliser un graphe de précédence entre les différentes entités logicielles d'exécution. Ce type de contrat est plutôt lié à l'équilibrage de charge. Il est bien entendu possible de créer d'autres types de contrats plus en relation avec l'équilibrage d'application.

## 4. Conclusion

Pour répondre à nos besoins de conception, de placement et d'exécution d'applications client-serveur, nous avons défini un langage unique VODEL. Avec VODEL, une architecture logicielle est décrite via un ensemble de composants logiciels (les DVSC) de granularité variable, qui dialogue via des liens de communication simples (DVSL) ou groupés (DVGL). Ces canaux de communication génériques ont été mis en place pour faciliter la description des couches médiateurs actuelles, en enrichissant la notion de canal notamment par la hiérarchie. Les composants logiciels sont, quant à eux, de deux types différents. Les DVSC actifs représentent en général des processus et les DVSC passifs logiques des ressources logicielles. L'intégration d'HADEL dans VODEL-D entraîne la création d'un troisième type d'entité les entités passives physiques.

La prise en compte de deux points vues différents, relatifs à l'équilibrage de charge et d'application, a entraîné l'extension de VODEL vers VODEL-D et la séparation du modèle de définition et du modèle d'exécution. Le premier est dédié au placement statique et le second à la gestion dynamique de l'exécution, notamment en utilisant les plans d'exécution. Les plans d'exécution représentent un découpage de l'exécution répartie en un ensemble d'évolutions validées de l'architecture d'exécution.

Enfin, pour conclure ce chapitre, nous avons reproduit le Tableau 18 en le complétant avec les spécificités de VODEL-D (cf. Tableau 25). Ainsi, dans VODEL-D, la hiérarchie est disponible aussi bien au niveau des composants que des connecteurs. Par contre, il n'existe de fonction de validation sur le modèle.

**Tableau 25:** Comparaison des langages de description d'architectures et de configuration

| Langages (type) | Granularité | Connecteur | Hiérarchie au niveau des | Vérification, validation | En Entrée | En Sortie |
|---|---|---|---|---|---|---|
| Aesop (ADL) | composant | connecteur | composants et des connecteurs | contraintes de topologie du style | un style d'architecture | un environnement exécutable (une fable) |
| Aster (MIL) | composant | système d'exécution spécialisables | composant | appariement de spécifications formelles | une description Aster | un système spécialisé pour une application |
| Durra (MIL) | tâches | canal | tâches | à la compilation | une description de l'architecture | un programme réparti exécutable |
| Olan (ADL) | composant | connecteur | composants | contraintes aux interfaces | une description OCL | structures réparties et interprétées |
| Rapide (ADL) | module | connections | interfaces | systèmes réactif à base de règles | un programme orienté objet | une architecture simulable |
| Regis/Darwin (MIL) | composant | liaison (bind) | composants | formelle et basé sur le π calcul | un programme Darwin | un programme réparti exécutable |
| Unicon (ADL) | composant | connecteur | composants | Vérification de types | un système | un programme exécutable |
| VODEL-D (ADL) | composant (DVSC) | connecteurs (DVSL) | composant et connecteur | calcul du placement et de la granularité des composants | une description VODEL-D | dépend des vues - du graphe de précédence ou du fichier de configuration |

# 5. Références bibliographiques


[Bach 91] M.J. Bach, «Conception des Systèmes Unix», Masson Eds., 1991.

[Bellissard & al. 96] L. Bellissard, S. BenAtallah, F. Boyer & M. Riveill, «Distributed Application Configuration», Proceedings of the 16th International Conference on Distributed Computing Systems, ICDCS'96, pp. 579-595, IEEE Computer Society, Hong-Kong, May 1996.

[Boloynesi & al. 87] T. Boloynesi & E. Brinksma, «Introduction to the ISO Specification Language LOTOS», Computer Networks and ISO System, Vol 14, pp 25-29, 1987.

[Demeure & al. 94] I. Demeure & J. Farhat, «Système de processus légers : concepts et exemples», Techniques et Science Informatiques, Vol. 13 (6), 1994, pp 765-795.

[El Kaim & al. 94] W. El Kaim & F. Kordon, «An Integrated Framework for Rapid System Prototyping and Automatic Code Distribution», Vth IEEE Workshop on Rapid System Prototyping, pp. 52-62, Juin 1994, Grenoble, France.

[Folliot & al. 96] B. Folliot & P.-G. Raverdy, «Adaptative Partitionning and Dynamic Allocation for Large Computing Systems»,Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications, Sunnyvale, California, pp. 1268-1279, August 1996.

[Herman & al. 83] D. Herman, F. Andre & J.P. Verjus, «Synchronisation de programmes parallèles», Dunod Eds., 1983.

[Hoare 78] C. Hoare, «Communicating Sequential Processes», Communications of the ACM, Vol. 21 (8), August 1978.

[Jacquemot 94] C. Jacquemot, «Chorus/COOL V2 Reference Manual», Technical Report, CS/TR-94-16, Chorus System, 1994.

[Meyer 88] B. Meyer, «Object-Oriented Software Construction», Prentice Hall International, 1988.

[Mullender 93] S. Mullender, «Distributed Systems», 2nd edition, Addison-Wesley, 1993, 595 pages.

[SDL 92] «Functional Specification and Description Language (SDL)», CCITT Blue Book, Recomendation Z.100, Geneva, 1992.

[Shaw & al. 96] M. Shaw & D. Garlan, «Software Architecture: Perspective on an Emerging Discipline», Prentice Hall Eds., 1996.

[Turner 93] K. Turner, «Using Formal Description Techniques», Wiley and Sons Eds, 1993.