Introduction Générale

«This is not a thesis book, this is Tezoumouk» (Tonton Isidore)

La communauté des systèmes répartis a atteint un certain consensus en ce qui concerne un ensemble de techniques et d'outils pour construire des applications **réparties et/ou parallèles** complexes. Ce consensus s'est concrétisé par la réalisation de normes (telles qu'ODP) et de plate-formes (telles qu'ANSA ou DCE). Néanmoins, elles n'aident pas toujours le concepteur dans le processus de création et de structuration des différentes entités logicielles d'une application.

Pour réduire la complexité de ces processus, le concepteur adopte souvent une approche modulaire. Une application est ainsi découpée en un ensemble **de composants logiciels**, chacun possédant son propre noyau fonctionnel. Ces composants logiciels sont alors testés, débogués, mis à jour et modifiés autant de fois que nécessaire. La construction de l'application elle-même, est ensuite réalisée par la mise en place d'un schéma d'interaction entre ses composants logiciels. Ce schéma est défini en fonction des interfaces d'accès des modules, qui ne doivent pas varier au cours du cycle de vie de l'application (sous peine de modifier le comportement attendu). Si ce schéma d'interaction est trop complexe, on a alors recours à des médiateurs (*Middleware*) spécialisés.

Dans un cadre de développement **client-serveur**, un module client interagit avec un module serveur en émettant des requêtes de services et en attendant les réponses du (ou des) serveur(s). Le schéma d'interaction étant clairement défini, on s'attend alors à ce que la construction de l'application en soit facilitée. On note néanmoins que l'utilisation de médiateurs ne pallie pas le manque de techniques **de structuration, de placement et d'exécution** d'applications exhibant une structure complexe [Ng & al. 95]. A cela s'ajoute le fait que le type des composants logiciels influe sur le choix des médiateurs à mettre en oeuvre. Ainsi, MTS le gestionnaire de transactions de Microsoft, ne fonctionne qu'avec des composants logiciels de type ActiveX. Le type des composants logiciels et des systèmes capables de les gérer est donc de première importance dans le choix des outils de conception, de développement et de déploiement utilisés.

Nos travaux de recherche visent à proposer une méthode de gestion du cycle de vie d'une application répartie et/ou parallèle. Pour cela, nous définissons un médiateur de description d'architecture d'applications en vue du calcul d'un placement et du suivi des éléments qui la constituent sur **une architecture matérielle hybride** (un ensemble de machines mono et multi-processeurs interconnectées). Ce médiateur adapte les contraintes de modélisation aux spécificités d'implémentation de l'application. Le calcul du placement est réalisé de manière statique lors du lancement de l'exécution (pour prévoir la localisation des composants logiciels), puis de manière dynamique durant l'exécution si besoin est (pour gérer dynamiquement la création et la mobilité de composants logiciels).

1. Outils et plate-formes de construction d'application

La recherche de l'architecture matérielle et logicielle adaptée à une application est devenue l'un des éléments clefs du cycle de vie du logiciel [Coulange 96].

Au niveau le plus haut du cycle de vie, on tente de spécifier une application et les composants logiciels qui la constituent. Les spécifications d'une application sont généralement réalisées avec des modèles de haut niveau [Sa & al. 96, Diagne & al. 96] et intégrées dans des méthodes de développement supportées par des outils (comme SA/RT [Ward & al. 85] dans ProTR [Azevedo & al. 94] ou OMT [Rumbaugh & al. 95] dans ObjectGEODE [Verilog 95]). Enfin, le modèle, lorsqu'il est formel, supporte de manière perti-

nente la validation de propriétés, la simulation du fonctionnement de tout ou partie du dit modèle et l'évaluation des performances.

A partir d'outils de spécification de haut niveau, il est possible de générer automatiquement des gabarits de code. Ces gabarits de code sont écrits dans un langage de programmation spécifique. Ils mettent en oeuvre la structure de contrôle d'un programme, tout en laissant au programmeur le soin de compléter le corps des fonctions et des procédures sans dépendance externe (on parle alors de confinement). On aide ainsi le programmeur à rester cohérent avec les spécifications, tout en lui imposant une architecture logicielle dépendante de la structure du code généré.

La répartition du code et la gestion du parallélisme inter et intra-application sont prises en compte lors du placement de l'application sur une architecture matérielle. L'exécution de l'application est ensuite réalisée en s'appuyant sur un environnement d'exécution plus ou moins ouvert et performant. Il est souhaitable que l'architecture logicielle de l'application soit la plus extensible possible, pour gérer l'hétérogénéité du matériel et l'évolution des environnements d'exécution dans le temps [Khokar & al. 93]. L'apparition récente de supports d'exécution répartis de code fixe ou mobile laisse par ailleurs envisager deux niveaux d'intervention : au niveau du langage (par analyse, parallélisation, compilation et optimisation du code) ou/et au niveau de l'exécution (par analyse du comportement) [Chatonnay & al. 96].

Enfin, dans le cas où des dysfonctionnements apparaissent, il est toujours possible de modifier le modèle, de générer du code, de l'exécuter et de recommencer le cycle à nouveau. Ce cycle de développement rapide d'applications (souvent qualifié de prototypage rapide), s'il est maîtrisé, conduit alors à des applications plus fiables ayant un degré de maintenabilité plus élevé [Luqi & al. 93].

2. Intégration de paradigmes de conception dans le cycle de vie

Pour s'y retrouver parmi toutes ces possibilités et bénéficier des techniques de prototypage rapide, le concepteur doit intégrer dans le cycle de vie de l'application des paradigmes de conception de haut niveau. Cette prise en compte, et c'est là que réside la difficulté, se situe à trois niveaux différents (cf. Figure 1) :

- *au niveau de la spécification opérationnelle* : dans la partie amont du cycle de vie, lorsque le squelette de l'application est produit automatiquement à partir des spécifications d'une application répartie [Kordon 94]. Le cadre architectural étant alors fixé, il ne reste plus au développeur qu'à compléter le code source du prototype généré.
- *au niveau de l'implémentation* : à un niveau intermédiaire du cycle de vie, lorsqu'on dispose déjà d'un programme écrit dans un langage de programmation séquentiel. Des techniques de parallélisation automatique de code source (telles que celles pour Fortran [Fox & al. 91]) sont alors envisageables. Un autre approche consiste à programmer directement une application avec des langages séquentiels étendus gérant le parallélisme ou offrant nativement des primitives de répartition (tels que Ada 95, SR [Atkins& al. 88] ou Eiffel// [Caromel 93]).
- *au niveau de l'environnement d'exécution* : dans la partie aval du cycle de vie, lorsque seul le programme compilé est disponible. Des techniques de placement des composants de l'application, sur une architecture matérielle statique ou reconfigurable, sont alors exploitables. Ce placement est soit dirigé par des primitives de configuration fournies par le programmeur (dans le code source ou à l'extérieur) [Magee & al. 92], soit laissé à la charge de l'environnement d'exécution [Zhou & al. 93].

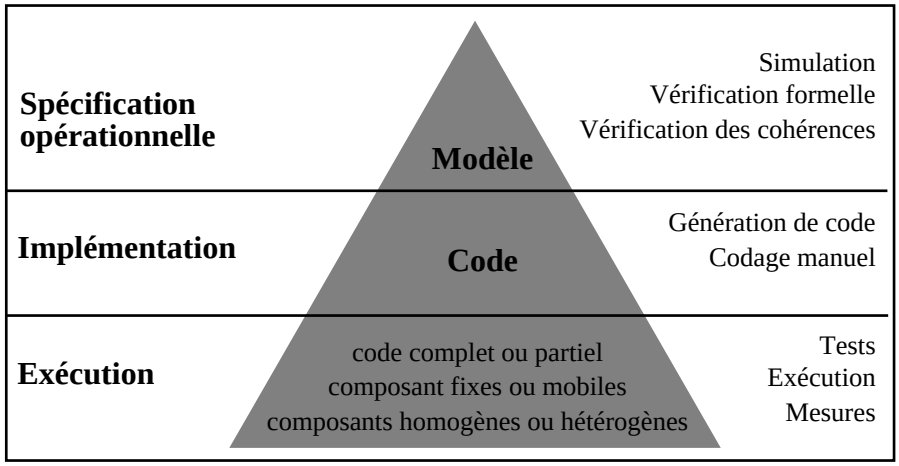


Figure 1 : Le cycle de vie d'une application constituée de composants logiciels

L'unification de ces trois niveaux au sein d'une même méthode, serait un gage de simplification du processus de développement d'une application répartie ou parallèle.

Cette volonté d'unification nous a amenés à considérer une application comme un ensemble de composants logiciels interagissants qu'il convient de placer au sein d'une architecture logicielle (pour un bon arrangement logique et pour respecter des règles de programmation) et matérielle (pour obtenir de bonnes performances d'exécution).

La prise en compte de cette problématique couvrant un ensemble de domaines très large, nous avons été conduits à privilégier deux axes de travail :

1) un axe visant à l'unification de l'équilibrage de charge et d'application ;
2) un axe visant à l'unification de la gestion des applications parallèles et réparties.

Dans les deux cas, nous avons étudié la construction, l'assemblage, le placement et l'exécution des composants logiciels ainsi que l'influence sur les performances et l'architecture de l'application.

2.1. Equilibrage de charge et équilibrage d'application

Notre première contribution au processus de constructions d'application répartie, a été d'identifier et de formaliser clairement deux problématiques de placement et d'exécution de composants logiciels. La première est appelée communément équilibrage de charge et a déjà été le fruit de nombreuses recherches [Zhou & al. 93, Bernard & al. 96]. Nous avons nommé la seconde équilibrage d'application, pour mieux la dissocier de l'équilibrage de charge.

2.1.1. Equilibrage de charge

L'équilibrage de charge est un équilibrage orienté machine. Son principal objectif est de tirer partie de toutes les ressources matérielles disponibles à un instant donné pour minimiser le temps d'exécution d'une application. Le regroupement et le placement des composants logiciels sont basés sur des critères liés à la charge des machines (charge des processeurs, taille mémoire libre) ou à la charge du réseau (débit, contention, routage).

Le grain de parallélisme des composants logiciels est en général du niveau programme [Folliot & al. 95], sauf dans le cas de code dit mobile (Java ou Odyssey) où le grain est celui du morceau de code téléchargeable.

L'équilibrage de charge est souvent utilisé lorsque les temps d'exécution des programmes sont inconnus et difficilement quantifiables a priori ou lorsque les applications ont

une durée de vie longue et qu'elles s'exécutent dans un environnement matériel non fiable (avec des mécanismes de reprise sur pannes).

2.1.2. Equilibrage d'application

L'équilibrage d'application est un équilibrage orienté logiciel. Son objectif majeur reste, certes de profiter au mieux des ressources matérielles présentes sur le réseau, mais en prenant en compte prioritairement les contraintes issues des spécifications de la structure de l'application. La traçabilité entre la spécification et l'implémentation des composants logiciels est donc l'élément primordial de ce type d'équilibrage. Les performances sont secondaires dans le sens ou le concepteur cherche d'abord à valider des choix architecturaux et des règles de structuration et de placement des composants logiciels. Il faut par contre s'assurer que le niveau de performances offert n'a pas d'influence notable sur les propriétés testées.

Le grain de parallélisme dans le cas de l'équilibrage d'application est extrêmement variable, ce qui complique la tâche d'équilibrage. Ainsi, si l'on utilise un langage de programmation orienté objet comme BOX [Geib & al. 93], où la distribution des classes est statique, mais la distribution des instances est dynamique, le grain de parallélisme est du niveau objet. Dans les nouveaux systèmes d'exploitation répartis orientés objets comme Choices et Taligent, le grain de parallélisme est le «framework», agglomération d'objets coopérants qui permet la décomposition du système en modules.

L'équilibrage d'application est surtout utilisé lors du développement d'applications constituées de nombreux composants logiciels de granularité faible (des objets par exemple) à durée de vie courte ou d'applications client-serveur dont l'architecture est difficile à mettre en oeuvre.

2.2. Applications parallèles et applications réparties

La prise en compte de ces deux types d'équilibrage au sein d'une même méthode de développement d'applications réparties est primordiale si l'on désire supporter à la fois les applications réparties et parallèles et l'interopérabilité de leurs composants logiciels. La volonté d'intégration de ces deux types d'application dans une méthode de développement commune est la seconde contribution de notre travail. En partant du principe qu'une application est constituée d'un ensemble de composants logiciels interagissants, la question que l'on se pose alors est la suivante : en quoi des architectures réparties et parallèles diffèrent-elles et quelles sont les conséquences sur les composants logiciels que l'on utilise pour les réaliser ?

2.2.1. Caractéristiques d'une application parallèle

Les recherches sur les applications parallèles ont produit différents paradigmes de programmation, différentes architectures d'exécution et différents algorithmes de résolution de problèmes communs. On peut citer comme exemple, en ce qui concerne les paradigmes de programmation :

- le paradigme *SPMD* utilisant des bibliothèques de communication telles PVM [Geist & al. 93] ou MPI [MPIF 93] ;
- le paradigme *Data-parallel* basé sur des langages tels que Fortran-D [Hiranandani & al. 92] ou HPF [HPF 94] ;
- le paradigme *orienté objet* utilisant des langages de programmation tels que Charm++ [Kale & al. 93].

Ces paradigmes de programmation entraînent la création d'architectures d'applications spécifiques qui sont adaptées à des architectures matérielles données (la machine Cray T3D ou la CM5 de Thinking Machine sont considérablement différentes), ou à des algorithmes spécifiques (calcul d'algèbre linéaire, prévisions météorologiques, etc.).

Le grain de parallélisme des composants logiciels d'une application parallèle est du niveau du programme exécutable monolithique. La granularité des données manipulées par un composant logiciel sur un processeur donné est variable en fonction du type, de la puissance et du nombre de processeurs. Ainsi, par exemple, une machine bi-Pentium et une Connection Machine gèrent des composants logiciels de grains différents. La gestion du contrôle est elle aussi variable en fonction des langages et des systèmes d'exploitation utilisés. Enfin, les directives de placement des différents composants logiciels et des données manipulées sont généralement intégrées dans le code source du programme (soit par des mots clefs spécifiques, soit par des commentaires pris en compte lors de la phase de liaison de code).

2.2.2. Caractéristiques d'un application répartie

En ce qui concerne les applications réparties, elles sont construites en permettant à des composants logiciels de communiquer par des techniques d'envoi de messages. Ces composants logiciels s'exécutent soit localement, soit sur des réseaux de stations de travail, ce qui entraîne :

- la prise en compte des techniques de gestion et de migration des processus et des processus légers par les systèmes d'exploitation ;
- la prise en compte de l'hétérogénéité matérielle (différentes machines disposant de processeurs de puissance variable) et logicielle (systèmes d'exploitation et alignement des données différents) ;
- la recherche d'un protocole de communication commun (tel que TCP/IP) ;
- la prise en compte des délais de communication variables entre les processus de l'application.

Le grain des composants logiciels était généralement du niveau du programme exécutable monolithique. La tendance actuelle est de réduire cette granularité pour permettre la gestion de composants logiciels basés sur des dizaines d'objets répartis. Le niveau de granularité du parallélisme et des données à gérer est alors bien plus fin que dans le cas des applications parallèles. Il dépend, par contre, lui aussi du paradigme de programmation utilisé. Enfin, le placement des différents composants logiciels est de plus en plus décrit et stocké dans un fichier de configuration situé en dehors de l'exécutable, qui est utilisé pour paramétrer le lancement (et parfois le déroulement) de l'exécution.

2.3. Unification des deux problématiques

L'unification des méthodes de construction d'applications parallèles et réparties est difficile à réaliser car elle doit être mise en oeuvre à deux niveaux : au niveau logiciel et au niveau matériel.

Au niveau logiciel, car de très nombreux langages propriétaires existent et tirent pleinement partie de la machine parallèle à laquelle ils sont dédiés (comme C* qui est une extension du langage C réalisée pour la Connection Machine). L'interopérabilité avec d'autres applications devient donc difficile.

Au niveau matériel, car l'accès aux machines parallèles se fait généralement par l'intermédiaire d'une machine hôte, avec laquelle il faut se synchroniser en terme de débit, d'alignement des données et de protocoles de communication. Enfin, à cela s'ajoute le fait qu'une station de travail est multi-utilisateur et multi-applications, alors qu'une machine parallèle est plutôt mono-utilisateur et mono-application.

Cette unification passe aussi par le choix d'un point de vue unique parmi les deux cités ci-dessous :

- **le point de vue langage** qui s'appuie sur des techniques récentes des langages de programmation ou de configuration pour la construction d'architectures logicielles. Il a pour but de faciliter la construction et l'interopérabilité d'applications réparties

et parallèles.

- **le point de vue système** qui s'appuie sur des systèmes répartis ouverts, sur des médiateurs (*Middleware*) ou sur des plate-formes d'exécution d'applications à la fois parallèles et réparties. L'architecture de l'application est alors construite en faisant abstraction des couches systèmes de bas niveau.

Une fois les applications construites, selon un point de vue donné, leur placement et leur exécution sont soumis à des contraintes variant selon le type d'équilibrage utilisé. Il est donc nécessaire de disposer d'une description de l'architecture logicielle et de l'architecture matérielle commune, pour pouvoir calculer, via des algorithmes adaptés à chaque type d'équilibrage, le placement des applications parallèles et réparties. Pour aider le concepteur dans cette tâche, nous proposons de **découpler le noyau fonctionnel de l'application des facettes techniques liées à son déploiement (distribution, partage, synchronisation, etc.)** [Kazman & al. 94]. Ce découplage, bien qu'indispensable, doit préserver la traçabilité des composants de l'application tout au long du cycle de vie. Dans le cas contraire, il est difficile de corriger les erreurs ou les mauvais choix détectés et de les propager jusqu'au modèle conceptuel (comme indiqué sur la Figure 1). C'est pourquoi, de notre point de vue, un langage de description d'architecture logicielle doit implémenter nativement cette séparation.

2.4. Synthèse

Nos travaux ne visent pas à créer une nouvelle méthode de développement de systèmes parallèles et répartis. Nous avons, par contre, cherché à intégrer et à homogénéiser les approches de développement et les outils actuels.

Pour cela, nous avons mis au point un langage de description pour chaque type d'architecture (logicielle et matérielle). Ces langages sont ensuite utilisés pour décrire, structurer, placer et exécuter des composants logiciels sur des machines cibles.

Notre approche fonctionne à condition que l'on ait préalablement répondu aux trois questions suivantes :

1) Quel est le type d'application que l'on désire concevoir (répartie, parallèle, une combinaison des deux) ?
2) Quel est le point de vue adopté (point de vue langage ou système) ?
3) Quel est le type d'équilibrage que l'on va mettre en oeuvre (équilibrage de charge ou d'application) ?

Les réponses à ces questions entraînent l'utilisation d'outils de conception et de génération d'applications particuliers qui doivent s'insèrer, et c'est là que réside la difficulté, dans une méthode de conception homogène. Nous proposons une telle méthode dans la section suivante.

3. Positionnement et démarche méthodologique

Nos travaux se placent dans le cadre de la Méthode d'Analyse et de Réalisation de Systèmes (**MARS**) [Estraillier & al. 92]. MARS est une méthode de gestion du cycle de vie d'applications parallèles et réparties. Elle est basée sur l'association du formalisme réseaux de Petri avec des méthodes de structuration par le biais de formalismes de haut niveau en vue de valider et vérifier des systèmes répartis [Diagne & al. 96]. MARS supporte l'équilibrage d'application grâce à des techniques de génération automatique d'applications à partir de modèles formels et semi-formels. Elle gère aussi l'équilibrage de charge, car elle prend en compte la gestion des prototypes générés et leur placement sur des architectures matérielles (placement de code OCCAM sur Transputer [Bréant &

al. 94] et placement de code Ada sur un réseau de station de travail [Kordon 92]). Enfin, MARS est adaptée aussi bien aux applications réparties que parallèles, car elle gère la description (formelle ou semi-formelle) d'entités concurrentes de granularité variable.

3.1. Intégration de la notion d'architecture

Nous avons cherché à étendre la méthode MARS, en recentrant notre approche sur la gestion des architectures. Notre première volonté a été de séparer la description de l'architecture matérielle support de l'exécution de l'architecture logicielle. Nous avons alors constaté qu'il était possible de raffiner encore ces descriptions de la manière suivante :

- les architectures matérielles actuelles sont composées de stations de travail et de machines parallèles connectées sur un même réseau. Ces architectures, appelées hybrides, offrent des puissances de calcul très importantes. Leur utilisation est par contre rendue difficile par l'hétérogénéité du matériel et du logiciel et par des techniques de placement de programmes non triviales.
- la description des architectures logicielles connaît depuis peu un essor important [Coad 92] et [Gamma & al. 94]. On cherche alors à capturer le savoir-faire et les morceaux de code qui lui sont associés dans le but de les réutiliser et de les maintenir tout en les documentant. **Dans nos travaux, nous séparons la description conceptuelle (par des gabarits de conception) d'une architecture logicielle, de sa description opérationnelle (via des squelettes d'implémentation). Cette double vision d'une même architecture est née d'une volonté de séparation des préoccupations [Hürsch & al. 95]**. Par ce moyen, on sépare formellement l'algorithme principal et les questions annexes telles que la gestion de la distribution, la gestion des synchronisations ou l'organisation des composants manipulés et leurs interactions.

Ces deux types d'architectures étant décrits de manière distincte, il devient possible de mettre au point **des algorithmes de placement génériques et non dédiés** à une plate-forme ou à des programmes donnés. Le suivi des évolutions dynamiques de ces architectures, durant l'exécution, est aussi prise en compte et fournit un support aux algorithmes de placement dynamique. Ces algorithmes dépendent bien entendu du type d'équilibrage souhaité. Ainsi, si l'on se place dans le cadre de l'équilibrage de charge, on privilégiera les évolutions de l'architecture matérielle en terme de performance et de disponibilité. Par contre, si l'on se place dans le cadre de l'équilibrage d'application, on privilégiera les contraintes liées aux composants logiciels par rapports à celles de l'architecture matérielle (on réalise les choix de contrôle de la création dynamique, de la migration et de la synchronisation des composants logiciels et on les soumet à l'architecture matérielle et non pas l'inverse).

3.2. Approche méthodologique

Nous avons mis au point une démarche méthodologique, nommée **MEDEVER** (*MEthode de DEveloppement pour un enVironnement d'Exécution Réparti*), gérant la construction, l'assemblage, le placement et l'exécution des composants logiciels sur une architecture matérielle hybride.

MEDEVER décompose le cycle de développement en quatre étapes (cf. Figure 2) :

1) **Description des architectures**. Cette phase est dédiée à la description de l'architecture matérielle support de l'exécution répartie et de l'architecture logicielle de l'application en vue de son placement. Ces descriptions sont soit générées automatiquement à partir de modèles de plus haut niveau, soit construites par l'utilisateur.
2) **Calcul de la granularité et placement**. On calcule la granularité des composants de l'architecture logicielle à placer sur l'architecture matérielle. Ce calcul est réa-

lisé en fonction des interactions entre composants et des contraintes imposées par l'utilisateur sur l'architecture logicielle. Bien entendu, le fait de privilégier l'équilibrage de charge ou d'application peut modifier la granularité et l'arrangement des composants logiciels. Puis, on effectue le placement statique de ces composants logiciels sur l'architecture matérielle cible. Ce placement est décrit dans un fichier de configuration et définit un modèle d'exécution.

3) **Exécution répartie**. La description et les évolutions des architectures logicielles (ie. de la granularité et de la localisation des composants qui la constitue) et matérielles sont donc considérées comme les deux flux d'informations parallèles à prendre en compte de manière dynamique pour assurer une exécution efficace de l'application. Reste alors à mettre en oeuvre un environnement et des outils susceptibles de travailler en temps réel sur ces flux. Dans le cas de l'équilibrage de charge, il existe déjà de nombreuses plate-formes de placement dynamiques, telles que Gatostar [Folliot & al. 95] ou UTOPIA [Zhou & al. 93]. En ce qui concerne l'équilibrage d'application, la gestion des évolutions dynamiques est généralement laissée à la charge du médiateur. Les composants logiciels interagissent alors par l'intermédiaire d'un bus logiciel ou d'un bus de message, comme dans OLAN [Bellissard & al. 96] ou dans Aster [Issarny & al. 96].

4) **Analyse et optimisation** : Les traces de l'exécution sont utilisées dans la dernière étape pour valider ou améliorer le placement lors d'une exécution future de l'application. Si une erreur architecturale importante est détectée, on doit pouvoir propager les modifications qu'elle entraîne du modèle d'exécution jusqu'au modèle initial. C'est d'ailleurs pourquoi, il est important d'offrir la traçabilité entre les phases de spécification, de conception et d'implémentation.

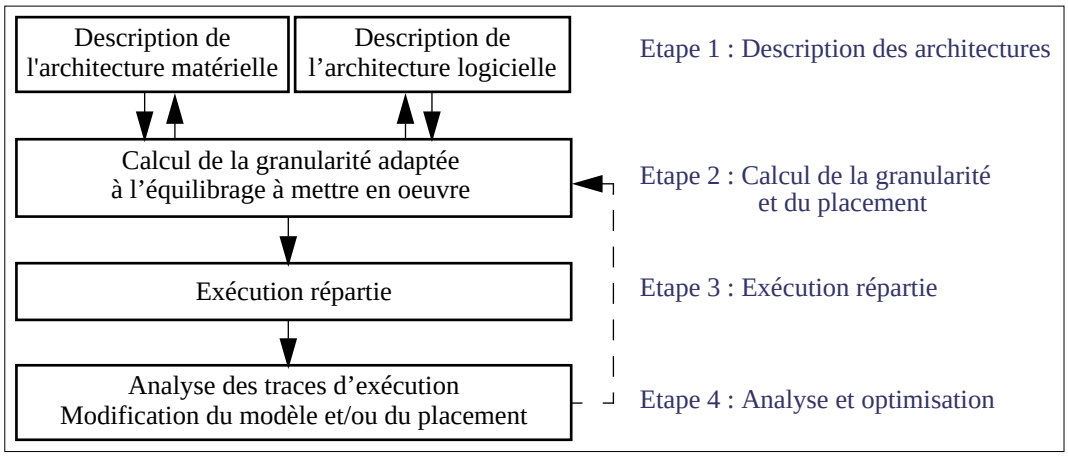


Figure 2 : La méthode MEDEVER

4. Organisation du mémoire

Pour aider le lecteur à mieux appréhender la structure de ce document, nous présentons son organisation chapitre par chapitre, mais aussi d'une manière plus condensée sous la forme d'un schéma unique. Sur chaque figure du schéma, nous indiquons aussi nos contributions.

Le chapitre1 est dédié à la recherche d'un langage de description d'architectures matérielles hybrides. Dans un premier temps, nous effectuons un état de l'art sur la description d'architectures matérielles dans les langages de programmation parallèle, les langages de configuration, les logiciels de simulation, les logiciels de répartition de charge et dans les logiciels d'administration de systèmes et de réseaux. Nous concluons qu'il n'existe pas, à notre connaissance, de langage adapté à la description d'architectu-

res matérielles hybrides et à la modélisation dynamique de ces architectures (changement de la charge des machines, taille de la mémoire libre, place disque). Nous proposons alors un langage de description d'architecture matérielle, nommé HADEL, correspondant au niveau de granularité géré dans notre méthode MEDEVER, pour réaliser le placement d'applications réparties et parallèles sur des architectures matérielles hybrides locales.

Dans le chapitre 2, nous étudions la notion d'architecture logicielle en nous focalisant sur les entités et les langages de description. Nous montrons alors que la prise en compte de la granularité est de première importance dès lors qu'on désire réaliser, au niveau architectural, l'équilibrage de charge et d'application pour des applications réparties et/ou parallèles. Pour étayer nos affirmations, nous réalisons un état de l'art sur les langages de description d'architectures logicielles de type client-serveur, constituées de composants logiciels interagissant. Ce travail a aussi confirmé que selon le point de vue adopté (système ou langage), les langages de description étudiés et leur intégration avec des environnements d'exécution, varient considérablement.

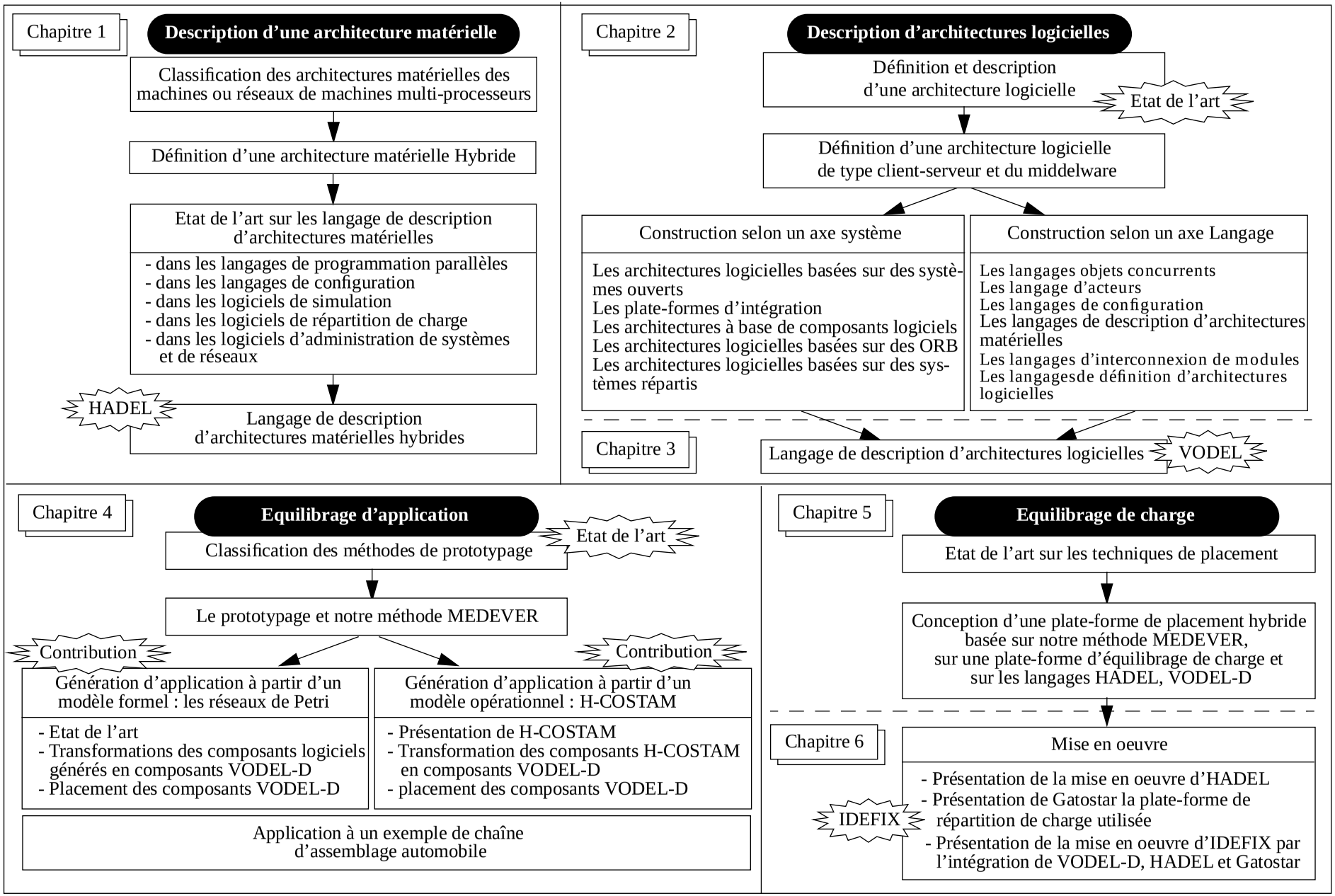
Suite à l'état de l'art du chapitre 2, nous proposons dans le chapitre 3 notre propre langage de description d'architecture logicielle, nommée VODEL-D. VODEL-D est le langage pivot de notre méthode MEDEVER, construit pour découpler l'architecture logicielle de définition, de celle d'exécution. On cherche ainsi à distinguer le modèle de l'architecture (les gabarits de conception) de leur implémentation (les gabarits de code), tout en gardant la traçabilité entre les deux modèles. Cette traçabilité est capitale, car lors de l'exécution de l'application, il est possible que des changements surviennent dans le placement des composants logiciels (en cas de migration des composants logiciels) ou dans la structure de contrôle de l'application (si les composants sont autonomes et «mutent»). Il est alors important de pouvoir répercuter ces modifications au niveau de l'architecture de définition.

La méthode MEDEVER ayant été conçue pour réaliser à la fois de l'équilibrage de charge et de l'équilibrage d'application, nous consacrons donc un chapitre à chacun de ces points de vue. Nous focalisons en particulier notre intérêt sur l'apport de nos langages HADEL et VODEL-D et sur leur utilisation.

Notre première étude, dans le chapitre 4, se situe dans un cadre de génération automatique de code à partir de modèles de haut niveau (*point de vue langage)*. On crée alors des prototypes répartis ou parallèles autonomes dont le placement est soit calculé à partir du modèle, soit contrôlé par l'utilisateur. Nos études dans ce domaine ont été réalisées dans la cadre du projet TAGADA, qui vise à la génération et à la répartition automatique de code ADA et C. Cette génération est réalisée, comme le préconise la méthode MARS, soit à partir d'un modèle formel (les réseaux de Petri dans CPN/TAGADA [Kordon 92]), soit à partir d'un modèle semi-formel (en utilisant un formalisme dédié à la génération de code appelé H-TAGADA [Kordon & al. 95]). Il ressort de nos études que la génération de code et l'exécution répartie du prototype sont facilitées lorsqu'on utilise des modèles semi-formels adaptés (tels que H-COSTAM [Kordon & al. 95]).

Notre seconde étude, dans le chapitre 5, a porté sur l'utilisation d'une plate-forme système de placement dynamique *(point de vue système)*. Nous montrons alors que les langages HADEL et VODEL-D s'intègrent parfaitement à de telles plate-formes. VODEL-D met à la disposition de la plate-forme des informations supplémentaires à celles généralement prises en compte par les algorithmes d'équilibrage de charge. Il est donc possible d'étendre ces algorithmes pour prendre en compte des critères purement applicatifs. Ces critères améliorent le placement au démarrage de l'application et permettent de prévoir certaines de ses évolutions dynamiques. L'architecture matérielle support de l'exécution est définissable par l'utilisateur qui le désire, grâce au langage HADEL. Les choix réalisés sont alors pris en compte par la plate-forme, même s'ils s'opposent à ceux dictés par les algorithmes d'équilibrage de charge en vigueur. Nos travaux dans ce domaine se sont appuyés sur la plate-forme d'équilibrage dynamique de charge Gatostar [Folliot & al. 95], développée dans notre équipe.

Enfin, le chapitre 6 présente nos réalisations concernant la mise en oeuvre d'IDEFIX (*Integrated Development Environment with Flexible dIstributed eXecution*), qui s'appuie sur Gatostar. L'environnement qui en résulte, offre le placement statique et dynamique d'applications (décrites via VODEL-D), dans un environnement matériel hybride (décrits via HADEL). Pour faciliter la transparence d'accès et le travail multi-utilisateurs, nous avons implémenté IDEFIX au-dessus d'un gestionnaire de mémoire partagé. Les premières expérimentations montrent que ce choix est pertinent, même si le coût en terme de temps de communication est important.


Chapitre 1
Description d'une architecture matérielle
Classification des architectures matérielles des machines ou réseaux de machines multi-processeurs
Définition d'une architecture matérielle Hybride
Etat de l'art sur les langage de description d'architectures matérielles
- dans les langages de programmation parallèles
- dans les langages de configuration
- dans les logiciels de simulation
- dans les logiciels de répartition de charge
- dans les logiciels d'administration de systèmes et de réseaux
HADEL
Langage de description d'architectures matérielles hybrides
Chapitre 2
Description d'architectures logicielles
Définition et description d'une architecture logicielle
Etat de l'art
Définition d'une architecture logicielle de type client-serveur et du middelware
Construction selon un axe système
Les architectures logicielles basées sur des systèmes ouverts
Les plate-formes d'intégration
Les architectures à base de composants logiciels
Les architectures logicielles basées sur des ORB
Les architectures logicielles basées sur des systèmes répartis
Construction selon un axe Langage
Les langages objets concurrents
Les langage d'acteurs
Les langages de configuration
Les langages de description d'architectures matérielles
Les langages d'interconnexion de modules
Les langagesde définition d'architectures logicielles
Chapitre 3
Langage de description d'architectures logicielles
VODEL
Chapitre 4
Equilibrage d'application
Classification des méthodes de prototypage
Etat de l'art
Le prototypage et notre méthode MEDEVER
Contribution
Contribution
Génération d'application à partir d'un modèle formel : les réseaux de Petri
- Etat de l'art
- Transformations des composants logiciels générés en composants VODEL-D
- Placement des composants VODEL-D
Génération d'application à partir d'un modèle opérationnel : H-COSTAM
- Présentation de H-COSTAM
- Transformation des composants H-COSTAM en composants VODEL-D
- placement des composants VODEL-D
Application à un exemple de chaîne d'assemblage automobile
Chapitre 5
Equilibrage de charge
Etat de l'art sur les techniques de placement
Conception d'une plate-forme de placement hybride basée sur notre méthode MEDEVER, sur une plate-forme d'équilibrage de charge et sur les langages HADEL, VODEL-D
Chapitre 6
Mise en oeuvre
- Présentation de la mise en oeuvre d'HADEL
- Présentation de Gatostar la plate-forme de répartition de charge utilisée
IDEFIX
- Présentation de la mise en oeuvre d'IDEFIX par l'intégration de VODEL-D, HADEL et Gatostar

5. Références bibliographiques

[Atkins & al. 88] Atkins M. et Olsson R., «Performance of multi-tasking and synchronization mechanisms in the programming language SR», Software-Practice and Experience, Vol. 18, pp. 879-895, 1988.

[Azevedo & al. 94] G.D.F. Azevedo, H. Azevedo & H. Jino, «ProTR : A Tool for Real Time Systems Development», Proc. of the 5th International Workshop on Rapid System Prototyping, Grenoble, France, Juin 1994.

[Bellissard & al. 96] Bellissard L., BenAtallah S., Boyer F. & Riveill M., «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.

[Bernard & al. 96] G. Bernard & B. Folliot, «Caractéristiques générales du placement dynamique : synthèse et problématique», Actes de l'école thématique CNRS placement dynamique et répartition de charge, Presqu'île de Giens, pp. 3-22, Juillet 1996.

[Bréant & al. 94] F. Bréant & J.F. Peyre, «An Improved Massively Parallel Implementation of Colored Petri Nets Specification», IFIP-WG 10.3, Working Conference on Programming Environment for Massively Parallel Distributed System, Ascona, Switzerland, 1994.

[Caromel 93] D. Caromel, «Towards a method of Object Oriented Concurrent Programming», Communications of the ACM, Vol. 36 (9), pp. 90-102, September 1993.

[Chatonnay & al. 96] P. Chatonnay, B. Herrmann, L. Philippe, F. Bourdon, P. Bar & C. Jacquemot, «Placement dynamique dans les systèmes répartis à objets», Calculateurs Parallèles, Vol. 8 (1), Mars 1996, pp 11-30.

[Chiola & al. 91] G. Chiola, C. Dutheillet, G. Franceschni & S. Haddad, «On Well-Formed Coloured Nets and their Symbolic Reachability Graph», High Level Petri Nets: Theory and applications, K. Jensen and G. Rozenberg Eds., Springer Verlag 1991.

[Coad 92] P. Coad, «Object-Oriented Pattern», Communications of the ACM Vol. 35 (9), pp 152-159, 1992.

[Coulange 96] B. Coulange, «Réutilisation du logiciel», Collection MIPS, Masson Eds., 1996, 324 pages.

[Diagne & al. 96] A. Diagne & F. kordon, «A Multi-formalism Prototyping Approch from Conceptual Description to Implementation of Distributed Systems», In Proc. of the 7th IEEE Int. Workshop on Rapid System Prototyping (RSP'96), Greece, Porto Caras, June 1996, pp 102-107.

[Estraillier & al. 92] P. Estraillier & C. Girault, «Applying Petri Net Theory to the Modeling, Analysis and Prototyping of Distributed Systems», In Proceedings IEEE / SICE Int. Workshop on Emerging Technology for Factory Automation, Australia, August 1992.

[Folliot & al. 95] B. Folliot, P. sens & P.-G. Raverdy, «Plate-forme de Répartition de Charge et de Tolérance aux Fautes pour Applications Parallèles en Environnement Réparti», Calculateurs Parallèles, Vol. 7 (4), pp. 345-366, 1995.

[Fox & al. 91] G. Fox, S. Hiranandani, K. Kennedy, C.Koelbel, U. Kremer, C.-W. Tseng and M.-Y. Wu. «FORTRAN D Language Specification», Department of Computer Science, COMP TR 90079, 1991.

[Gamma & al. 94] E. Gamma, R. Helm, R. Johnson & J. Vlissides, «Design Patterns : Elements of Reusable Object-Oriented Software», Addison Wesley, 1994, 395 pages.

[Geib & al. 93] J. M. Geib, C. Gransart, C. Grenot, "Mixing object and activities in complex active objects", ECOOP'93, Workshop on Object Based Distributed Programming, Kaiserslautern, Germany, July 26-27, 1993.

[Geist & al. 93] A. Geist, J. Dongarra & R. Manchek, «PVM3 User's Guide and Reference Manual», Oak Ridge National Laboratory and University of Tenessee, May 1993.

[Hiranandani & al. 92] S. Hiranandani, K. Kennedy & C. Tseng, «Compiler Support from Machine Independant Parallel Programming in Fortran-D», Elsevier Science Eds., 1992.

[HPF 94] Koelbel, Loveman, Schreiber, Steele & Zosel, «The High Performance Fortran Handbook», MIT Press, 1994.

[Hürsch & al. 95] Walter Hürsch & Cristina Videira Lopes, «Separation of Concerns», Northeastern University Technical Report, NU-CCS-95-03, Boston, February 1995

[Inmos 88] Inmos, «Transputer Development System», Prentice Hall Eds., 1988.

[Issarny & al. 96] Issarny V. & Bidan C., «Aster: A Framework for Sound Customisation of Distributed Runtime Systems», Proceedings of the 16th International Conference on Distributed Computing Systems, May 1996.

[Kale & al. 93] L.V. Kale & S. Krishnan, «Charm++: A Portable Concurrent Object Oriented System Based on C++», Proceedings of the Conference on Object Oriented Programming Systems, Languages and Applications, September 1993.

[Kazman & al. 94] R. Kazman, L. Bass, G. Abowd & M. Webb, «SAAM: A Method for Analysing the Properties of Software Architectures», Proc. of the 16th Int. Conf. on Software Engineering, Sorrento (Italy), 1994.

[Khokar & al. 93] A.A. Khokar, V.K. Prasanna, M.E. Shaaban & C.-L. Wang, «Heterogeneous Computing: Challenges and Opportunities», IEEE Computer, pp. 18-27, June 1993.

[Kordon 92] F. Kordon, «Prototypage de systèmes parallèles à partir de réseaux de petri colorés, application au langage ADA dans un environnement centralisé ou réparti», PhD thesis, Université Pierre et Marie Curie, 1992.

[Kordon 94] F. Kordon, «Propososal for a Generic Prototyping Approach», Proc. IEEE Symposium on Emerging Technologies and factory Automation, Tokyo, Japan, pp. 396-403, November 1994.

[Kordon & al. 95] F. Kordon & W. El-Kaim, «H-COSTAM : A Hierarchical COmunicating State Machine for Generic Prototyping», In Proceedings of the 6th IEEE International Workshop on Rapid System Prototyping, Triangle Park Institute, June 1995, pp. 131-138.

[Luqi & al. 93] Luqi, M.Shing & J.Brockett, “Real-time scheduling for software prototyping”, 4th International Workshop on Rapid System Prototyping, Triangle Park Institute, June 1993, pp. 150-163.

[Magee & al. 92] J. Magee, N. Dulay & J. Kramer, «Structuring Parallel and Distributed Programs», International Workshop on Configurable Distributed System, London, pp 102-117, 25-27 March 1992.

[MPIF 93] MPI Forum, «Document for a Standard Message Passing Interface», Technical Report CS-93-214, University of Tennessee, Site : <http://www.mcs.anl.gov/mpi/mpi-report>, November 1993.

[Ng & al. 95] K. Ng, J. Kramer, J. Magee & N. Dulay, «The Software Architect’s Assistant - A visual environment for distributed programming», Proceedings of Hawaii International Conference on System Science (HICSS-28), January 1995.

[Rumbaugh & al. 95] J. Rumbaugh, M. Blaha, F. Eddy, W. Premerlani & W. Lorenson, «OMT - Modélisation et conception orientée objets», Masson Eds., 1995.

[Sa & al. 96] J. Sa, J.A. Keane & B.C. Warboys, «Software Process in a Concurrent, Formally-based Framework», Proceedings of the IEEE International Conference on Systems, Man and Cybernetics, Bejing, China, October 1996, pp. 1580-1585.

[Verilog 95] The Verilog Newsletter, No 8, January 1995.

[Ward & al. 85] P.T. Ward & S. J. Mellor, «Structured Development for Real time Systems», Vol. 1-3, Prentice Hall, Englewoods Cliffs, 1985.

[Zhou & al. 93] S. Zhou, Z. Zheng, J. Wang & P. Delisle, «UTOPIA: A load Sharing Facility for Large, Heterogeneous Distributed Computer Systems», Software Practice and Experience, Vol. 23 (12), December 1993, pp. 1305-1336.