Chapitre 4 - Placement de composants logiciels : Application à l'équilibrage d'application

«Toute parole, tout acte, toute pensée est un devenir» (André Levasseur)

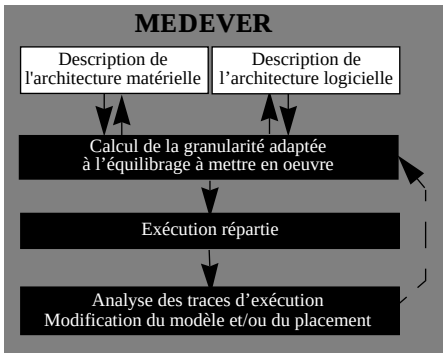
Résumé

Nous avons étudié l'équilibrage d'application en axant nos recherches sur l'un des aspects du prototypage, à savoir la génération de code réparti et son placement sur une architecture matérielle. Nous présentons donc dans un premier temps un état de l'art sur les méthodes de prototypage et sur ce que nous appelons le prototypage par raffinement. Puis, nous détaillons la Méthode d'Analyse et de Réalisation de Systèmes (MARS) et la manière dont nous y avons intégré notre méthode MEDEVER. Nous étudions ensuite la génération d'applications à partir de réseaux de Petri. Nous déduisons de cette étude que ce modèle réseau de Petri n'est pas toujours adapté à la génération automatique de code source. Nous présentons alors nos travaux en cours sur un nouvel outil de génération d'applications, utilisant un modèle de plus haut niveau que les réseaux de Petri (appelé H-COSTAM) et son articulation avec les langages HADEL et VODEL-D pour le calcul du placement des éléments du prototype généré.

Apports scientifiques

Nous montrons dans ce chapitre l'intérêt d'un modèle opérationnel semi-formel, nommé H-COSTAM, par rapport à un modèle formel (à base de réseaux de Petri) pour le prototypage d'applications. Nous constatons aussi que quel que soit le modèle utilisé, l'utilisation de VODEL-D permet de calculer le placement de composants logiciels d'un prototype généré automatiquement. Par contre, l'implémentation des gestionnaires, chargés au sein du prototype de contrôler l'exécution répartie, a de fortes influences sur les performances. Le choix de la meilleure architecture logicielle à générer n'est donc pas triviale et tire partie à la fois de la qualité sémantique des informations de haut niveau disponibles dans le modèle de spécification, du langage de génération et de la granularité des composants logiciels gérés.

Plan



Equilibrage d'application
Classification des méthodes de prototypage
Le prototypage et notre méthode MEDEVER
☞ Génération d'applications 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'applications 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

Mots clefs

Prototypage, génération de code, placement, H-COSTAM, H-TAGADA, réseaux de Petri, application répartie, spécification, plate-forme d'intégration.

L'équilibrage d'application donne, par rapport à l'équilibrage de charge, une plus grande importance à la phase de conception. Ainsi, on ne cherche pas à accélérer le plus possible le temps d'exécution d'une application, mais avant tout à concevoir, intégrer et valider des choix architecturaux et conceptuels. Ces choix architecturaux sont validés à priori sur un modèle formel ou semi-formel. Cette forme de validation n'est d'ailleurs pas incompatibles avec des techniques d'expérimentation concrètes («in visu») souvent recherchées par l'utilisateur (telles que des techniques de simulation par exemple). L'emploi de techniques de prototypage via de la génération de code est alors un moyen efficace pour passer semi-automatiquement d'une spécification à une implémentation.

Si on se place dans le cadre de la construction d'applications réparties (ou parallèles), on remarque qu'il n'existe toujours pas de méthodes miracles pour appréhender et résoudre leur complexité de mise en oeuvre. De nouveau, l'utilisation conjointe de méthodes de description de haut niveau et de techniques de prototypage visant à la production d'un squelette de code réparti sont des moyens d'appréhender cette complexité. L'expérimentation de politiques de placement diverses des éléments logiciels sur l'architecture matérielle est alors un moyen d'optimiser son application.

Dans ce cadre, nous pensons qu'il est dommageable de contraindre le rôle des spécifications à des fonctions de validation et de génération de code. En effet, dans une approche opérationnelle lorsque le concepteur conçoit son modèle, il a déjà plus ou moins une idée du découpage fonctionnel de son application. Par contre, il ne maîtrise pas toujours les techniques de répartition de l'architecture du prototype généré. Le générateur de code doit donc détecter et maximiser, autant que cela soit possible, le parallélisme d'une application.

Ces techniques d'optimisation sont, bien entendu, différentes en ce qui concerne les applications parallèles et réparties. Dans le premier cas, on fait en général appel à des compilateurs parallélisants adaptés à une machine parallèle donnée, alors que dans le second, on cherche plutôt à minimiser les coûts de communication et de partage des ressources de l'application. L'utilisation d'un langage tel que VODEL(-D) pour la description des architectures logicielles statiques (resp. dynamiques) est alors envisageable, à condition d'offrir des techniques de transformations entre le modèle opérationnel et VODEL(-D). Cette description couplée à une description de l'architecture matérielle avec le langage HADEL est alors une base solide pour le calcul d'algorithmes de placement.

Nous avons mis en oeuvre l'équilibrage d'application en axant nos recherches sur l'un des aspects du prototypage, à savoir la génération d'applications réparties et le calcul automatique de leur placement sur une architecture matérielle. Nous avons par conséquent exclu du cadre de nos investigations les approches basées sur la simulation.

La section 1 présente un état de l'art sur les méthodes de prototypage et sur ce que nous appelons le prototypage par raffinement. La section 2, décrit la Méthode d'Analyse et de Réalisation de Systèmes (MARS) [Estraillier & al. 92] et la manière dont nous y avons intégré notre méthode MEDEVER. La section 3 présente un état de l'art sur les techniques de génération d'applications réparties à partir de réseaux de Petri et nos travaux dans ce domaine. Nous découvrons alors que les réseaux de Petri ne sont pas totalement adapté à la génération automatique de code et au calcul de son placement. C'est pourquoi, dans la section 4, nous présentons nos travaux en cours sur la génération d'applications à partir d'un modèle de plus haut niveau que les réseaux de Petri (appelé H-COSTAM) et son articulation avec nos langages HADEL et VODEL-D. La section 5 reprend l'exemple d'une chaîne d'assemblage de carrosserie automobile, présenté dans [Kordon 92], en comparant les deux approches de la génération d'applications et le placement des composants logiciels qui en résulte.

1. Les méthodes de prototypage

Le développement de systèmes complexes répartis et/ou parallèles se fonde de plus en plus sur des techniques de prototypage pour déjouer les risques [Boehm 91] et éliminer les erreurs de spécification et de conception [Estraillier & al. 92] qui coûtent d'autant plus cher qu'elles sont détectées tard dans le cycle de développement [Vonk 92]. Le prototypage est donc utilisé [Hulaas 97] pour :

- déterminer la faisabilité d'un produit ;
- évaluer le coût de réalisation des développements ;
- établir et valider les besoins de clients ;
- rechercher des estimations des performances d'un système ;
- tester le système obtenu en utilisant le prototype comme un modèle de l'application finale.

Un prototype est, par définition, un modèle exécutable opérationnel de certains aspects du système [Luqi 89]. Il existe à ce jour de nombreuses techniques de prototypage, en fonction du type de prototype visé (un exécutable, du code intermédiaire, etc.) et du modèle utilisé pour spécifier le problème. Nous présentons dans la suite de ce paragraphe un bref état de l'art des classifications des techniques de prototypage. Puis nous introduisons une nouvelle technique de prototypage, le prototypage par raffinements, qui s'adapte particulièrement bien à la génération de code à partir d'un modèle formel ou semi-formel.

1.1. Première classification des techniques de prototypage

Une première classification des techniques de prototypage a été proposée par [Hallmann 91 & Asur 93] et met en exergue trois familles distinctes : le prototypage exploratoire, jetable et incrémental.

1.1.1. Le prototypage jetable

Le prototypage jetable (*Throw-away*) est une technique visant à produire un prototype qui sert à la validation des exigences de tout ou partie du système. Cette technique est surtout utilisée lorsque l'on désire une évaluation rapide et flexible d'un système et lorsque la stabilité et la performance des prototypes créés ne sont pas primordiales. Il existe deux catégories de prototypes jetables : les prototypes simulés (*Simulated*) et les prototypes autonomes (*Stand-alone*). Dans le premier cas, les prototypes sont exécutables uniquement dans l'environnement de développement utilisé (tel que Cabernet [Pezzè & al. 92]), alors que dans le second, les prototypes s'exécutent de manière autonome (SAO [Cassigneul 91] en est un exemple).

1.1.2. Le prototypage exploratoire

Le prototypage exploratoire (*Exploratory*) ou en cascade (*Waterfall*) [Vonk 92] & [Meinadier 94] est la technique la plus proche du développement traditionnel d'une application. Son seul but est de recueillir rapidement et à moindre coût des informations sur le système sans objectif prémédité de réutilisation. L'idée principale étant de conserver la flexibilité à la fois de l'architecture du système et des fonctions qu'il assuré. Les modules développés sont donc ajoutés les uns après les autres au système alors que les modules non implémentés sont simulés pour permettre d'effectuer des tests. Le prototype est donc amélioré au fur et à mesure des versions. Ce type de prototypage est utilisé pour la vérification des aspects fonctionnels (les services qu'offrent le système) ou architecturaux (l'assemblage donné des modules d'un système).

1.1.3. Le prototypage incrémental

Le prototypage incrémental (*Incremental*) qui est aussi appelé **évolutif** [Luqi 92] ou **cumulatif** [Kettani 95]. Cette technique consiste à produire un prototype et à l'enrichir constamment jusqu'à obtenir l'implémentation réelle finale du système. Le développement incrémental est parfois dit conduit par l'architecture, l'architecture étant considérée comme l'élément central autour duquel s'organise les autres activités du développement. La définition des besoins est donc dans un premier temps abrogée en faveur de l'architecture, qui doit être stabilisée le plus rapidement possible dans le cycle de développement. L'architecture du prototype du fait de son importance doit être robuste, extensible et performante. L'usage du prototype incrémental est considéré comme dangereux car la frontière entre le prototype et le système final est floue [Choppy 94].

1.2. Autre classification des techniques de prototypage

Une autre classification du prototypage différencie le **prototypage vertical du prototypage horizontal** [Budde & al. 84]. Dans le premier cas, seules certaines fonctions critiques sont implémentées dans le prototype, car elles sont considérées comme indépendantes du reste du système. Dans le second cas, toutes les fonctions du produit final sont présentes, mais sont implémentées à des niveaux différents, laissant de coté certains aspects qui ne sont pas primordiaux. Si les deux approches sont combinées dans un même prototype, on parle **de prototypage hétérogène**. Un prototype hétérogène est donc un modèle exécutable du système aux différentes parties spécifiées à différents niveaux d'abstractions [Leòn & al. 93]. Durant la durée de vie de ce prototype, ce mélange de niveau d'abstraction est susceptible d'évoluer [Zakhama 96].

Le prototypage mixte [Hulaas 97] est composé à la fois du prototypage hétérogène et du prototypage incrémental. Le principe du prototypage mixte est de produire un programme dans un langage impératif par compilation de spécifications formelles. Ces spécifications intermédiaires sont structurées de telle manière qu'elles puissent être progressivement remplacées par du code source écrit à la main (plus rapide et/ou plus proche des besoins réels). Les modules résultats sont alors qualifiés d'abstrait alors que les implémentation des utilisateurs sont qualifiées de concrètes. Le prototype peut donc être exécuté à n'importe quel moment dans le cycle de développement et le prototype est toujours dans un état minimal cohérent.

Enfin, **le prototypage rapide** [Luqi 89] est une classification transversale à celles présentées. Ce type de prototypage souligne le fait que le processus de construction et d'évaluation d'une série de prototypes est rapide. La rapidité du développement est, dans ce cas, synonyme de faible coût et d'accélération du rythme de production des versions du prototype. Un système de prototypage rapide doit se baser sur un langage de prototypage offrant des outils d'abstraction et de réutilisation et combiner une méthode de prototypage systématique à un ensemble d'outils intégrés d'aide au prototypage assisté par ordinateur [Kordon 92]. Or, il n'existe pas de formalisme universel qui réponde à l'ensemble des problèmes de génération de code. De plus, il est prouvé que le choix d'un formalisme adapté à un type de problème est délicat [Murphy & al. 89] et a des répercussions sur la qualité du résultat [Luqi 92].

Cette classification étant transversale, le prototypage rapide est mis en oeuvre soit à partir d'un modèle exécutable (comme dans ObjectGeode [Verilog 95] ou IPTES [Leòn & al. 93]), soit à partir d'une image logicielle générée automatiquement (comme dans CPN/Tagada [MARS 96] et SANDS/CO-OPN [Buchs & al. 92]).

1.3. Synthèse : le prototypage par raffinements

Le prototypage ne doit pas être réduit à la génération de code ou à la simulation du comportement d'un système. Il inclut aussi des techniques et des outils de modélisation, d'évaluation du modèle et du prototype. Bien plus qu'une technique de développement,

c'est une approche. Nous étendons donc la vision commune du prototypage par l'utilisation de deux entités distinctes : le modèle et le prototype (cf. Figure 34).

Un utilisateur travaillant sur la version N d'un modèle, l'évalue et raffine éventuellement les informations fournies par l'environnement utilisé pour le décrire. L'approche utilisée pour cette étape est formelle ou pas. De toutes les manières, l'utilisateur ne peut obtenir toutes les informations concernant l'exécution du système, car certains aspects sont uniquement expérimentés si le prototype fonctionne dans son environnement réel [Dolev & al. 94].

La génération de l'application fournit à l'utilisateur un prototype qui est l'image exacte du modèle prêt à fonctionner dans l'environnement d'exécution final. Ce prototype est composé de deux parties : le code généré dans un langage donné et le fichier de configuration indiquant le placement calculé des éléments du prototype. **C'est au niveau de la génération de l'application que s'insère nos travaux et notre méthode MEDEVER.** Le langage VODEL-D est alors utilisé pour décrire l'architecture logicielle et pour calculer le placement de l'application à créer.

L'application une fois créée est ensuite évaluée, modifiée et si le résultat n'est pas satisfaisant, raffinée jusqu'à obtenir une version N+1 plus proche des spécifications et corrigeant les erreurs de la version N.

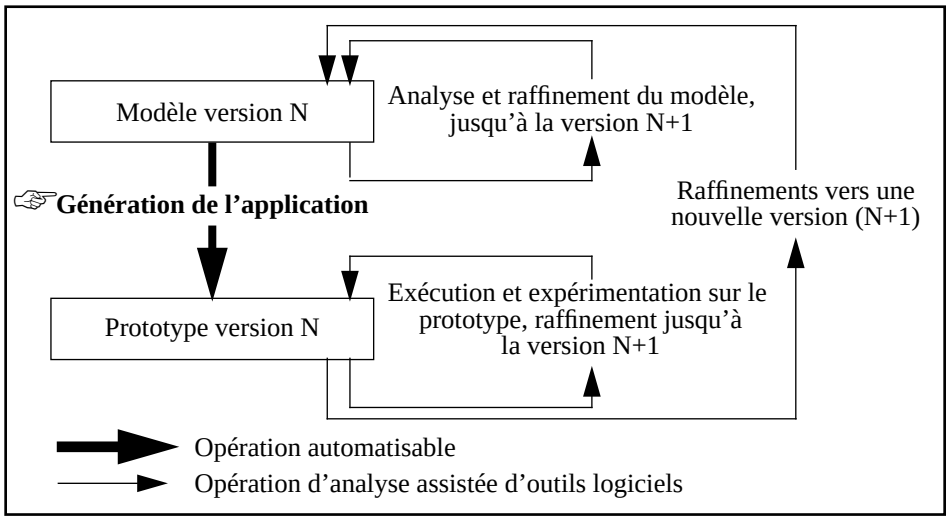


Figure 34 : Le prototypage par raffinements

Cette vision du prototypage est une extension du prototypage incrémental et renforce les différences entre la notion de modèle et de prototype. Nous nommons cette nouvelle vision du prototypage, **le prototypage par raffinements**. Le prototypage par raffinements s'appuie alors sur des techniques de validation et d'évaluation à deux niveaux (modèle et prototype), ce qui est particulièrement intéressant lorsque le modèle est décrit via des techniques formelles. C'est aussi un avantage, lorsqu'on désire calculer à priori le placement des éléments de l'application qui seront générés. On peut alors raffiner à la fois le modèle et le placement des entités logicielles qui en découlent.

Nous avons appliqué une démarche de prototypage par raffinements dans le cadre de la méthode MARS, qui utilise des modèles de descriptions formels et semi-formels.

2. Contexte de l'étude : la méthode MARS

La Méthode d'Analyse et de Réalisation de Systèmes (MARS) est basée sur l'association du formalisme réseaux de Petri avec des méthodes de structuration par le biais de forma-

lismes de haut niveau en vue de valider et vérifier des systèmes répartis [Estraillier & al. 92, Diagne & al. 96a].

La méthode MARS sépare le cycle de vie d'une application en trois niveaux (cf. Figure 35). Le premier niveau est utilisé pour une description du modèle via des formalismes de haut niveau (semi-formels). Ces formalismes sont utilisés pour décrire des systèmes répartis et parallèles complexes de manière modulaire. Le choix d'un formalisme à ce niveau est fortement dépendant du domaine d'application considérée [Attiogbé & al. 96]. Le second niveau est composé d'un formalisme pivot permettant la validation et la recherche de propriétés formelles sur le modèle. Enfin, le dernier niveau est constitué des programmes générés qui s'exécutent dans un environnement réel.

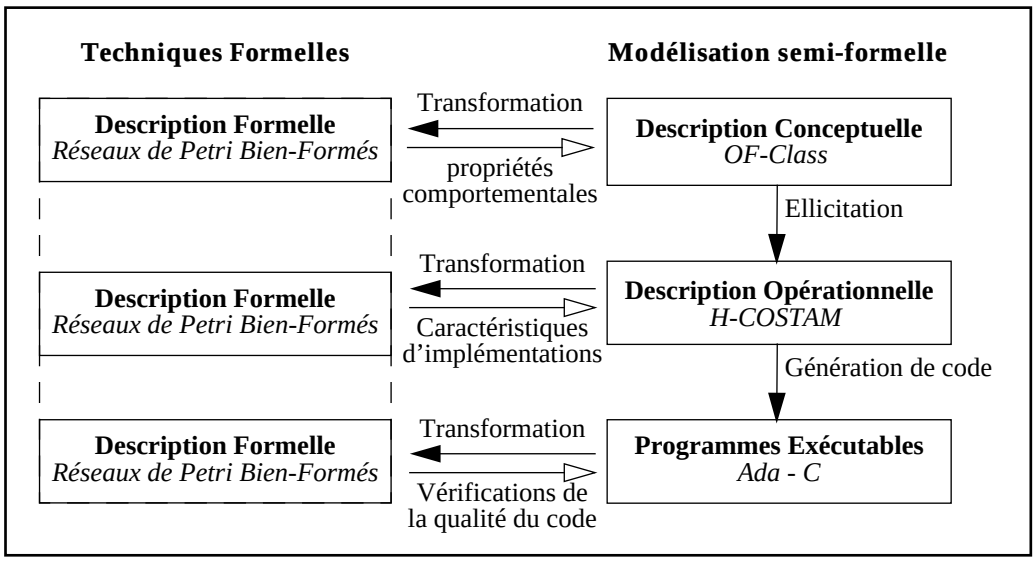


Figure 35 : La méthode Mars

2.1. Les trois formalismes de MARS

MARS étant une méthode dédiée à l'analyse, à la conception et la réalisation de systèmes répartis s'appuie sur un formalisme formel et sur deux formalismes semi-formels :

- *formalisme formel* : les réseaux de Petri Bien Formés [Chiola & al. 91 & Haddad 91], qui est une classe de réseaux de Petri colorés ;
- *formalisme semi-formel conceptuel* : le formalisme OF-CLASS (*Object Formalism Class*) [Diagne 97] est une approche semi-formelle qui intègre la description et la validation de réseaux de Petri Colorés [Jensen 92] avec une approche orientée-objet de spécification de systèmes répartis. Il fournit des informations concernant l'association des composants de l'application, la manière dont ils fonctionnent et dont on doit les utiliser.
- *formalisme semi-formel opérationnel* : le formalisme H-COSTAM (*Hierarchical COmmunicating State Machines*) [Kordon & al. 95] est un formalisme, basé sur des machines à états communiquantes, à partir desquelles on réalise la génération automatique de code optimisé.

Ces formalismes sont complémentaires car ils sont utilisés soit pour raffiner le modèle au niveau conceptuel (regroupement des services dans des modules), soit pour raffiner le modèle au niveau opérationnel (choix d'implémentation et d'optimisation du code). Quatre types d'opérations reliant ces formalismes existent (cf. Figure 35) :

1) le passage d'un modèle semi-formel (OF-Class et H-COSTAM) vers le modèle formel (réseaux de Petri). Chaque transformation est dédiée à la recherche de propriétés précises sur le modèle formel.

2) le passage d'une description conceptuelle (OF-CLASS) à une description opérationnelle (H-COSTAM). Cette transformation, appelée l'ellicitation [Diagne & al. 96b], est semi-automatique car elle implique des choix de la part du modélisateur lors de la transformation d'un modèle vers un autre.
3) le passage d'une description semi formelle opérationnelle à du code source. Cette transformation, appelée génération de code, est réalisée automatiquement.
4) le passage du code généré à une représentation réseau de Petri modélisant les appels de procédure et de fonctions du programme. Le réseau de Petri obtenu est ensuite utilisé pour la validation statique de propriétés du programme [Barkaoui & al. 92].

Les formalismes utilisés au niveau semi-formel sont donc munis de méthodes et d'outils de transformation en réseaux de Petri colorés. Ils sont nécessaires à la vérification des propriétés explicites ou au calcul des invariants structurels pouvant être utilisées pour améliorer l'implémentation du système cible (optimisation du code généré, directives de placement sur une architecture matérielle sous-jacente, etc.).

2.2. Génération d'applications et MEDEVER

Dans la méthode Mars, la génération d'applications produit du code exécutable et un fichier de configuration de placement. La phase de génération de code fabrique automatiquement le squelette d'une application à partir du niveau formel ou semi formel. le fichier de configuration est lui généré automatiquement ou à la main. Ce fichier de configuration indique au gestionnaire d'exécution le placement initial des composants logiciels de l'application. Le prototypage par raffinements est alors utilisé pour mettre au point le placement des différents éléments du prototype sur une architecture répartie, en fonction des propriétés du modèle et du prototype.

L'approche de prototypage par raffinement s'intègre parfaitement à la problématique de notre méthode MEDEVER. **Notre contribution se situe alors principalement au niveau de la description des architectures logicielles et matérielles et du calcul du placement de la première sur la seconde** (cf. Figure 36).

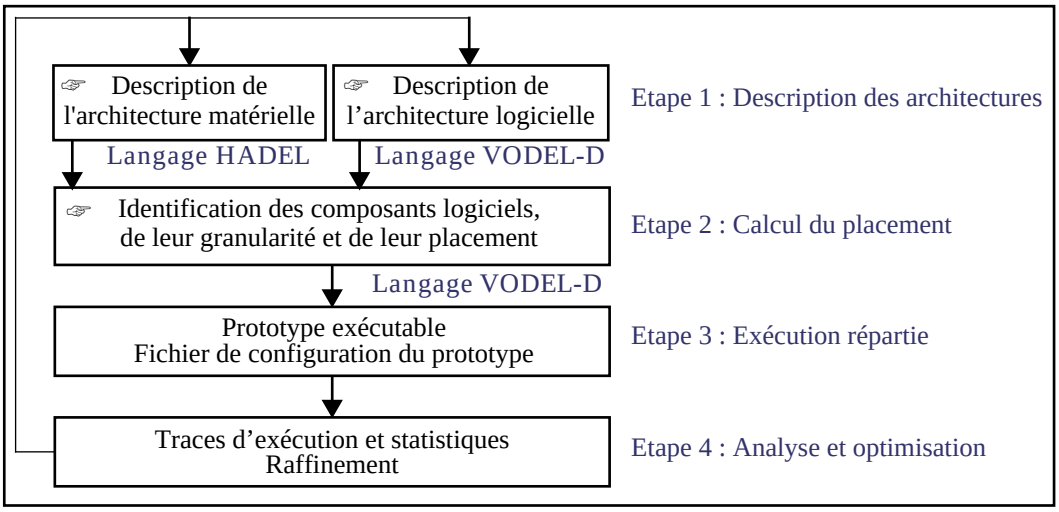


Figure 36 : La méthode MEDEVER adaptée à l'équilibrage d'application

L'équilibrage d'application, dans le cadre de MEDEVER est alors réalisé en tenant compte du niveau de spécification de l'application:

- si on utilise un modèle formel basé sur des réseaux de Petri, l'architecture logicielle du prototype généré et son placement sont issus du calcul d'invariants sur le modèle et d'un algorithme de décomposition du modèle en objets de génération

[Kordon 92].

- si on utilise le modèle semi-formel H-COSTAM, le calcul du placement du prototype est basé à la fois sur les spécifications de l'architecture logicielle (par transformation de H-COSTAM en VODEL-D) et matérielle d'exécution (décrite avec le langage HADEL). La méthode MEDEVER sert alors dans ce cas au placement et à la gestion d'objets H-COSTAM et des morceaux de code générés qui s'y rattachent.

Nous présentons dans la Section 3. les techniques de génération de code et de placement à partir d'un modèle formel (les réseaux de Petri colorés) et leur intégration dans MEDEVER. Puis dans la Section 4., nous étudions les techniques de génération de code et de placement à partir d'un modèle semi formel.

3. Génération d'applications à partir de réseaux de Petri

De nombreuses études ont été réalisées sur l'utilisation de méthodes formelles pour la description de systèmes répartis complexes [Murata 90, Shapiro 91]. L'intérêt des spécifications formelles est qu'elles offrent des techniques de validation du modèle avant son prototypage.

Nous commençons notre étude par un état de l'art des outils de génération de code à partir de réseaux de Petri. Puis, nous présentons nos travaux dans ce domaine et leur adéquation avec notre langage de description d'architecture logicielle VODEL-D.

3.1. Etat de l'art

Les réseaux de Petri, basés sur une théorie mathématique éprouvée, offrent des mécanismes de validation par calcul de propriétés formelles (telles que la recherche d'invariants, l'absence d'interblocage, etc.). Une fois le modèle validé le prototypage des modèles est alors possible directement à partir de classes distinctes de réseaux de Petri tels que les réseaux de petri P/T [Genrich 87], Colorés [Jensen 92], Temporisés [Felder & al. 93] et Algébriques [Reisig 91 et Gaudel 91].

Nous présentons une synthèse des caractéristiques principales des outils de génération de code à partir de réseaux de Petri. Cette étude issue de [Colom & al. 86 et Kordon 94a] que nous avons remise à jour et augmentée est résumée dans le Tableau 26.

Tableau 26: Comparaison d'outils de prototypage à partir de Réseau de Petri

Références	Classe de RDP	Implémentation matérielle	Génère	Technique d'Exécution	Approche	Prototypage
Silva & al. 82	P/T	Monoprocesseur	Assembleur	Interprété	Centralisé	Incrémental
Nelson & al. 83	P/T et arcs inhibiteurs	Monoprocesseur	PL/1 et PL/S	Compilé	Centralisé	Incrémental
Valette & al. 83	P/T	Monoprocesseur	Assembleur	Interprété	Centralisé + extensions distribution	Incrémental
Thuriot 85	Coloré	Monoprocesseur	Tâches Pascal	Interprété	Centralisé	Incrémental
Bruno & al. 86 et 95	Coloré (PROT-Nets)	monoprocesseur multiprocesseur	Ada	Compilé	Distribué (partiellement)	Incrémental
Colom & al. 86	Coloré	monoprocesseur multiprocesseur	Ada	Compilé	Centralisé, distribué & hybride	Incrémental
Murata & al. 86	Control-Nets	monoprocesseur	PL	Interprété	Centralisé	Incrémental

Tableau 26: Comparaison d'outils de prototypage à partir de Réseau de Petri

Références	Classe de RDP	Implémentation matérielle	Génère	Technique d'Exécution	Approche	Prototypage
Hauschildt 87	P/T	monoprocesseur & multitâche	C	Compilé	Distribué	Incrémental
Taubner 88	P/T	multiprocesseur (transputer)	OCCAM	Compilé	Distribué	Incrémental
Bréant 90	P/T	multiprocesseur (transputer)	OCCAM	Compilé	Hybride	Incrémental
Kordon & al. 90	P/T	monoprocesseur	Ada	Compilé	Hybride	Incrémental
Paludetto 91	P/T	monoprocesseur	Ada	Compilé	Centralisé	Incrémental
Kordon & al. 91a, 91b & 92	Coloré	monoprocesseur multiprocesseur	Ada	Compilé	Centralisé & Distribué	Incrémental
Buchs & al. 92 et 96	Coloré + hiérarchie	monoprocesseur	C++	Compilé	Centralisé	Incrémental
Pezzè & al. 92	Coloré	monoprocesseur	C	Compilé	Centralisé	Incrémental
Bréant & al. 93 et 94	Coloré	multiprocesseur	OCCAM	Compilé	Distribué	Incrémental
El Kaim & al. 94, Kordon & al. 95	Coloré	multiprocesseur	Ada	Compilé	Distribué	Incrémental
Lakos & al. 95a et Lakos 95b	Coloré + hiérarchie	monoprocesseur	C++	Compilé	Distribué	Incrémental
Zhakama 96	Temporisé et coloré	multiprocesseur	Ada 83 + intégration de programmes existant	Simulé	Distribué	Hétérogène
Hulaas 97	Algébrique	multiprocesseur	Ada 95	Simulé et compilé	Distribué	Mixte

A partir de l'étude résumée au Tableau 26, nous mettons en exergue trois approches successives de la génération de code à partir de réseaux de Petri : l'approche centralisée, l'approche totalement répartie et l'approche hybride.

3.1.1. L'approche centralisée

La première approche s'est efforcée de mettre en place un exécuteur de réseau de Petri centralisé. Cet exécuteur en fonction du marquage du réseau étudie pour chaque transition si les conditions de tir sont vérifiées (*firability*). Des filtres d'évaluation ont d'ailleurs été étudiés pour réduire le nombre de transitions testées, pour éviter des problèmes de performance lorsque le réseau de Petri croît en taille [Colom & al. 86, Murata & al. 86]. Néanmoins, cette approche ne préserve pas le parallélisme du modèle, car l'exécuteur du réseau de Petri est une tâche séquentielle.

L'approche centralisée est toujours d'actualité dans des environnements industriels flexibles (*Flexible Manufacturing Systems*), à la seule différence que le joueur de réseau de Petri sert d'ordonnanceur de tâches [Briz & al. 94]. Ainsi, une tâche est représentée par une action atomique acceptant des données en entrée et produisant des données en sortie. De telles actions sont bien entendu associées au tir des transitions. Le joueur de réseau de Petri est centralisé mais peut demander des exécutions de tâches sur des processeurs distants. Un tel ordonnanceur est difficile à répartir du fait de son besoin de maintenir un état global du système.

3.1.2. L'approche totalement répartie

La seconde approche, à l'inverse, a consisté à répartir totalement l'exécuteur du réseau de Petri [Hauschildt 87, Taubner 88]. Chaque place et chaque transition est donc implémentée par une tâche distincte. Cette approche préserve le parallélisme, mais lorsque le

réseau ou le nombre de jetons de couleurs distinctes croît, la gestion des conflits et les échanges d'informations ruinent les performances. Néanmoins, sur certaines architectures matérielles parallèles spécifiques, ce problème est atténué [Taubner 88], à condition de prendre en compte les spécificités de routage et de congestion de messages [Bréant 91 & Bréant & al. 92].

3.1.3. L'approche hybride

La dernière approche est une approche hybride, car mélangeant les qualités des deux premières. Elle est basée sur la décomposition d'un réseau de Petri en un ensemble de processus séquentiels communiquants regroupés de manière cohérente avant d'être répartis.

On cherche alors à adapter la granularité des composants logiciels du prototype dans le but de minimiser les conflits structurels et sémantiques du modèle. Les conflits structurels sont solvables par transformation automatique [Kordon 92, Peyre 93 et Bréant & al. 94] ou manuelle du réseau de Petri. Les conflits liés à la sémantique même du modèle sont, par contre, plus complexes à résoudre.

L'avantage de cette approche est d'offrir des implémentations efficaces, mais uniquement sur des réseaux de Petri décomposables en machines à états communiquantes.

3.1.4. Synthèse

La génération automatique de code à partir de réseaux de Petri n'est pas toujours réalisable. L'approche hybride permet néanmoins par des techniques de transformations de modifier la description du réseau de Petri, sans changer son comportement, pour que du code soit généré.

Ces transformations sont difficiles à réaliser de manière automatique sur des réseaux de Petri colorés. C'est pourquoi, une extension naturelle de l'approche hybride consiste à augmenter le formalisme réseau de Petri pour qu'ils contiennent des informations de plus haut niveau concernant la structuration du modèle [Sibertin-Blanc 94, Bastide 95 et Valk 95].

Nous présentons dans le chapitre suivant notre approche de la génération d'applications, qui complète la génération de code, ainsi que les liens avec notre méthode MEDEVER.

3.2. Notre approche de la génération d'applications

L'évaluation du parallélisme dans un modèle décrit avec des réseaux de Petri est important car les approches centralisées et totalement réparties de génération de code ont de faibles performances lorsque la taille du modèle croît. Une solution consiste alors à découper le réseau de Petri en un ensemble de composants à partir desquels on génère séparément du code (et donc des entités qui s'exécutent concurremment). Ce découpage est souvent réalisé dans le but de produire des machines à états séquentielles [Hack 74] à partir du modèle initial. En effet, ces dernières sont des candidates intéressantes pour un processus de génération de code, car elles sont implémentables par un programme séquentiel (un processus ou une tâche). **On adopte alors une approche hybride**.

De notre point de vue, une machine à états correspond formellement aux invariants de place d'un réseau de Petri (la réciproque n'étant pas toujours vraie) [Kordon & al. 91b]. Ainsi, pour réaliser une approche hybride, présentée au chapitre précédent, il est nécessaire de mettre au point une méthode pour sélectionner les invariants qui conduisent à des machines à états. Cette méthode présentée dans [Kordon 92], est hors du cadre de cette étude, néanmoins nous présentons dans la suite de ce chapitre ses points clefs.

Dans la suite de ce paragraphe, nous décrivons les différentes étapes de la génération de code. Nous étudions ensuite les composants du modèle intermédiaire, appelés objet de génération, à partir desquels est réalisée la génération de code. Les techniques employées pour calculer la décomposition d'un réseau de Petri en un ensemble d'objets de généra-

tion font, par contre, l'objet du paragraphe suivant. Enfin, nous concluons par un exemple et quelques remarques issues de nos expérimentations.

3.2.1. Les étapes de la génération de code

Le processus de génération de code [Kordon & al. 94b] est décomposé en trois étapes successives (cf. Figure 37) :

- **la phase d'identification** analyse syntaxiquement la description complète du modèle ;
- **la phase d'analyse** décompose le modèle en un ensemble d'objets de génération définis dans [Kordopn 92]. Les objets de génération sont donc une interprétation d'une spécification décrite avec des réseaux de Petri en un ensemble de machines à états communiquantes ;
- **la phase de génération** intègre les contraintes du langage de génération cible et la description des objets de génération pour produire un programme exécutable.

Les deux premières étapes sont dépendantes du formalisme utilisé. La phase de génération est par contre indépendante des formalismes de description utilisés, puisqu'elle se base sur une description intermédiaire constituée d'objets de génération. **C'est à ce niveau que l'intégration avec MEDEVER est réalisée**. En effet, à ce niveau on travaille directement sur une description de l'architecture logicielle à des fins de génération de code et de placement de composants logiciels.

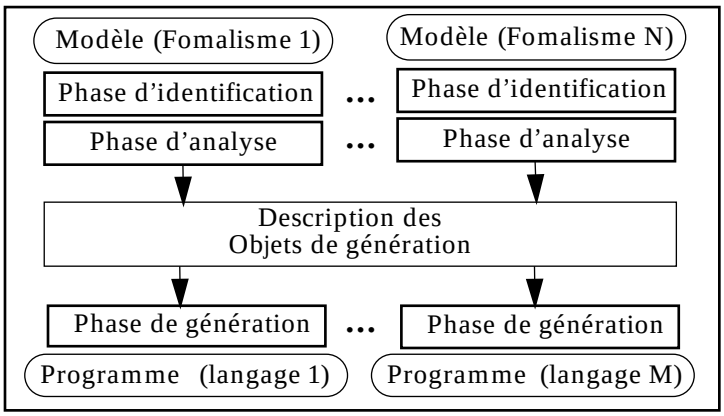


Figure 37 : Description des trois étapes de la génération de code

3.2.2. Présentation des objets de génération

L'approche adoptée pour réaliser un prototype à partir d'une spécification réseaux de Petri, repose sur les actions suivantes :

- analyse de la spécification du système que l'on réalise qui permet de générer le squelette de contrôle du prototype.
- interprétation de la représentation des activités concurrentes. A cette fin, nous introduisons les **objets de génération** (ou G-objets) qui constituent des types de composants logiciels.

Les objets de génération sont au nombre de quatre et sont déduits à partir de certains invariants du réseaux de Petri [Kordon 92, Peyre 93] :

1) *les Processus* sont constitués par des sous-ensembles du réseau de Petri assimilables à une machine à états [Hack 74]. **De tels sous-ensembles représentent des modèles de comportement séquentiels dans lesquels aucun parallélisme n'est possible**. Nous les interprétons comme des classes de Processus instantiables. Ainsi, : les processus sont munis de leur contexte unique et d'un comportement propre défini dans la classe de processus. Par abus de langage, nous appelons

désormais une classe de processus (un modèle de comportement séquentiel) Processus. Un processus instancié étant alors une instance du modèle de comportement défini par un Processus. Un processus instancié s'exécute en concurrence avec les autres. Le calcul du nombre de processus instanciés s'effectue à partir du marquage initial du modèle.

2) *une Ressource* est une place du réseau de Petri qui n'est contenue dans aucun Processus. Une ressource correspond à une zone de stockage de données accessible par un ou plusieurs Processus. Le type d'information transitant dans une Ressource est défini par son domaine de couleurs.

3) *les Actions* sont associées à toutes les transitions du modèle et correspondent aux traitements effectués par le système. Dans le modèle, une action correspond à un niveau de description atomique.Une Action peut également réaliser un mécanisme de synchronisation entre Processus. Les différents Processus du modèle sont des entités séquentielles concurrentes dont l'exécution est soumise à des contraintes liées aux Actions. Ces contraintes mettent en oeuvre une séquentialisation des Actions des différents processus instanciés. Elles s'expriment :

   - directement : l'exécution d'une Action implique plusieurs processus instanciés qui réalisent ainsi une synchronisation;
   - indirectement : l'exécution d'une Action est liée au contenu de Ressources, c'est-à-dire de zones de stockage contenant des données produites par d'autres processus instanciés (on effectue alors une communication asynchrone).

   Si le traitement est local au Processus (une seule contrainte directe), l'Action est dite simple. Dans le cas contraire (plusieurs contraintes directes), elle correspond à un rendez-vous entre plusieurs Processus, l'Action est alors synchronisée. Enfin, une Action, qu'elle soit simple ou synchronisée, est dite gardée si elle est soumise à des contraintes indirectes.

4) *les Etats_Processus* correspondent aux places du réseau de Petri qui relient les Actions d'un Processus entre elles. Dans un modèle validé, un Etat_Processus sans successeur est un état d'accueil pour le Processus. Il est associé à une terminaison du traitement. Un Etat_Processus ayant plusieurs successeurs est considéré comme un point de choix. Un Etat_Processus qui ne comporte pas de successeur, est dit de terminaison. S'il comporte un seul successeur, il est simple. Dans les autres cas, il est qualifié d'alternatif.

Du point de vue architectural, un processus est donc un regroupement d'objets de génération pour former des composants logiciels. Ces composants logiciels ont comme spécificité de représenter une machine à états séquentielle. On en déduit donc qu'il n'existe pas de parallélisme intra-composant. Par contre, le parallélisme inter-composant existe et comprend parfois des contraintes de synchronisation (les actions synchronisées). En ce qui concerne les ressources, leur placement vis à vis des processus a une influence sur les coûts de communication.

3.2.3. Placement des objets de générations via VODEL-D

La calcul du placement des composants logiciels du prototype passe non pas par la création d'un graphe d'objets de génération, mais par un graphe de composants logiciels communiquants. Cette nouvelle représentation du problème est complétée par l'évaluation des coûts d'exécution et de communication de ces modules.

A partir de ce graphe valué, on applique un algorithme de placement donnant la répartition des modules du prototype sur une architecture matérielle simple (un réseau de stations de travail par exemple). Une description de ces algorithmes est donnée dans [Coriat 92]. Pour passer d'un graphe d'objets de génération à un graphe de composants logiciels

communiquants décrit en VODEL, on utilise les transformations présentées au Tableau 27.

Tableau 27: Transformation des objets de génération vers VODEL-D

Objets de génération	Objets VODEL-D
Processus	DVSC Actif, centralisé ou Réplicable
Action Synchronisée	DVSC Actif, Centralisé
Ressource	DVSC Passif (stockage de données) ou canal (communication)

Un exemple de gestion d'opérations bancaires simples

L'exemple de la Figure 38, modélise l'incrémentation et la décrémentation d'unité de valeur dans le compte bancaire de clients.

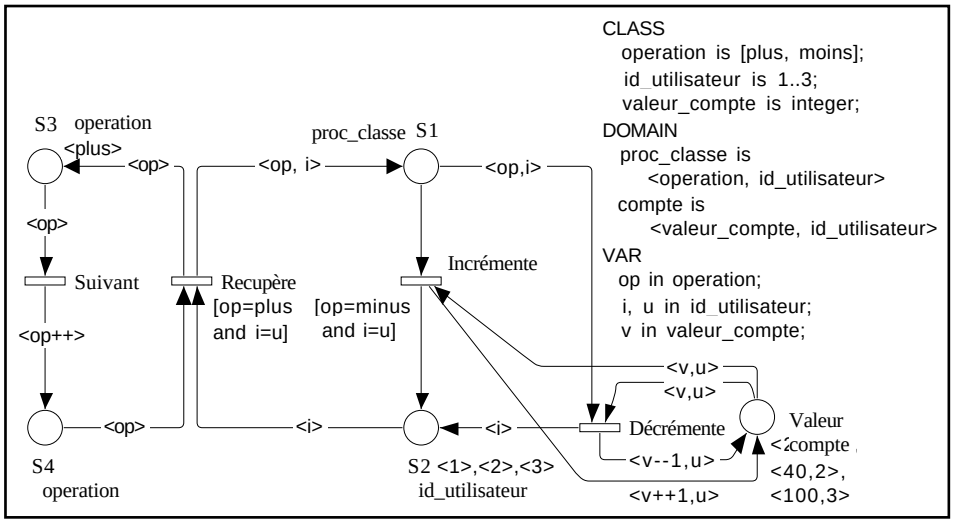


Figure 38 : Gestion de compte utilisateur dans une banque

Le générateur d'opération bancaire (places S3 et S4) génère alternativement des opérations bancaires (plus ou moins une unité de valeur). Le gestionnaire d'opération (places S1 et S2), quant à lui, récupère les opérations produites et les applique au compte bancaire d'un client.

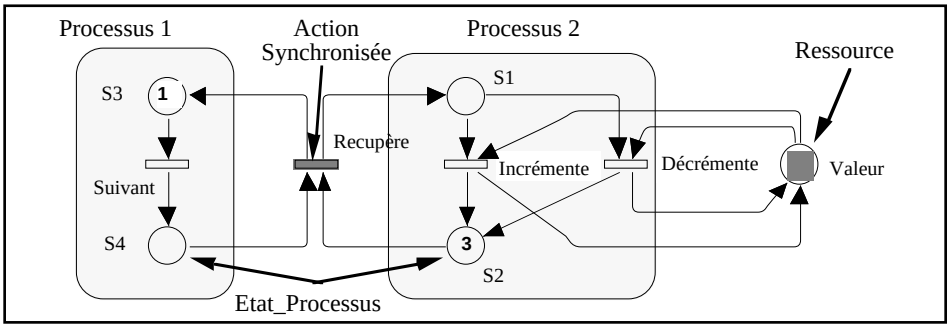


Figure 39 : Décomposition en objets de génération du modèle de la Figure 38

On en déduit les objets de génération suivants présentés sur la Figure 39 :

- le premier processus est composé des Etats_processus S3 et S4 et de l'action «Suivant» ;

- le second processus est composé des Etats_processus S1 et S2 et des actions Incrémente et Décrémente ;
- la place valeur est une ressource ;
- la transition Récupère est une Action Synchronisée, partagée par les deux processus du modèle.

La transformation des objets de générations en composants logiciels VODEL-D, est présentée au Tableau 28.

Tableau 28: Transformation des G-Objets en composants logiciels VODEL-D

Objet de Génération	Nom	Type	Evolution	Hiérarchie	Regroupement
Processus 1 (places S3 / S4)	DVSC 1	Actif	Fixe	Primitif	X
Processus 2 (places S1 / S2)	DVSC 2	Actif	Fixe	Primitif	X
Action Synchronisée (transition récupère)	DVSC 3	Actif	Fixe	Primitif	X
Ressource (place valeur)	DVSC 4	passif	Fixe	Primitif	X

Les processus sont modélisés par un DVSC actif, centralisé et fixe et l'action synchronisée est représentée comme un DVSC actif. Lors de l'exécution, le processus 1 sera instancié une fois et le processus 2, trois fois conformément au marquage initial. L'action synchronisée gère un rendez-vous entre toutes les instances des processus, elle ne peut donc être répliquée. En ce qui concerne la ressource valeur, on remarque que chacune de ces marques est liée à une instance particulière du processus (marque 1, 2 ou 3). On décide donc de répliquer la ressource valeur et de créer un lien d'attachement avec le processus 2. On en déduit alors les informations de placement présentées au Tableau 29.

Tableau 29: Description des informations nécessaires au placement

Nom	Type	Attachement	Répartition	Nombre d'instances
DVSC 1	Actif	X	Centralisé	1
DVSC 2	Actif	DVSC 4	Centralisé	3
DVSC 3	Actif	X	Centralisé	1
DVSC 4	passif	DVSC 2	Répliqué	comme DVSC 2

Tableau 30: Placement des DVSC sur les machines

Nombre de machines	Machine 1	Machine 2	Machine 3	Machine4	Machine5
1	1,2,3,4,5				
2	1, 2	3,4,5			
3	1	2	3,4,5		
4	1	2	3	4,5	
5	1	2	3	4	5

Le placement optimal des DVSC nécessite un nombre de sites au plus égal à la somme des instances de DVSC non liées, soit dans notre cas 5. Les propositions de placement des DVSC précédents sur une architecture matérielle allant de 1 à cinq machines sont décrits dans le Tableau 30. Les DVSC numérotés 3, 4 et 5 correspondent aux trois instances du couple (DVSC 3, réplique du DVSC 4). Lors du placement on sépare donc dans un premier temps les DVSC 1 et 2 des autres DVSC qui sont des répliques. Puis, au fur et à mesure que le nombre de machines devient suffisamment important, on sépare les répliques

3.2.4. Architecture du prototype généré

L'architecture du prototype est basée sur des gabarits de conception spécialisés, de telle sorte que chaque objet VODEL-D issu des objets de génération est pris en charge par un gestionnaire spécialisé :

- *gestionnaire de composants logiciels* : chaque processus possède son propre gestionnaire qui réalise le comportement de l'objet de génération processus. Un processus est instancié conformément au nombre de jetons présents dans les Etats_processus ;
- *gestionnaire de ressources* : Il gère l'accès aux objets de génération de type ressources. Il évalue les requêtes d'accès des clients (les processus), met à jour et maintient le marquage des places du réseau de Petri correspondantes aux ressources utilisées (addition, soustraction de jetons, exclusion mutuelle, absence d'interblocages, etc.) ;
- *gestionnaire de synchronisations* : Il prend en charge les communications synchrones dans le prototype. Il active les actions synchronisées quand tous les participants d'un rendez-vous sont présents et les réveille quand l'action a été réalisée ;
- *gestionnaire de prototype* : Il assure le contrôle de l'application : gestion de l'initialisation, de la terminaison et des communications entre les différents modules du prototype).

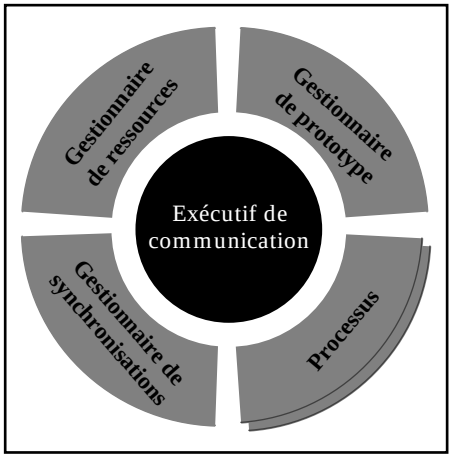


Figure 40 : Architecture d'un prototype généré par CPN/Tagada

L'architecture finale du prototype s'inspire de celle de l'ECMA [ECMA 93] (cf. Figure 40), notamment en ce qui concerne l'utilisation d'une couche de communication unique. L'architecture du prototype est donc générique et son implémentation est possible dans différents langages (les interfaces des différents gestionnaires et le contenu des processus ne changent pas).

Les algorithmes des gestionnaires et l'implémentation du noyau de communication utilisés sont par contre à adapter en fonction des langages et environnements d'exécution cibles. Par exemple, si le prototype est généré en langage Ada, le noyau de communica-

tion est fourni par l'environnement d'exécution. Dans le cas du langage C, ce noyau s'appuie sur les IPC [Stevens 90].

Les directives de placement ne sont pas insérées dans le code source, mais dans un fichier de configuration externe. Le placement des différents éléments du prototype est réalisé par configuration statique au démarrage du prototype et sans recompilation [Boussant & al. 92, Kordon 92].

3.2.5. Répartition automatique du prototype généré

La répartition du code généré est réalisée en deux temps. Dans un premier temps, on calcule les objets de génération à partir du modèle formel et leur transformation en objets VODEL-D. On dispose alors de l'architecture de définition. Puis, dans un second temps, on prend en compte à la fois les composants logiciels, les gestionnaires chargés de les gérer et une description sommaire de l'architecture matérielle cible. Ainsi, dans l'architecture d'exécution, on cherche à répliquer certains gestionnaires et certaines ressources dans le but de maximiser les performances.

Approches de répartition de l'architecture d'exécution

Pour répartir le prototype généré, deux approches ont été envisagées : l'approche «systématique» et l'approche «optimale». Dans l'approche systématique, on crée :

1) un gestionnaire de prototype centralisé ;
2) un gestionnaire de ressources centralisé ;
3) un gestionnaire de synchronisation par synchronisation.

L'inconvénient de l'approche systématique est qu'elle entraîne des goulots d'étranglement au niveau des accès aux gestionnaires centralisés. L'approche optimale offre par rapport à l'approche systématique une gestion délocalisée du serveur de prototype et de ressources. Ainsi, on décide de créer :

1) un gestionnaire de prototype par site ;
2) des gestionnaires de ressources répartis avec des ressources qui migrent en fonction des demandes. Pour gérer les conflits d'accès aux ressources, les demandes de ressources sont estampillées et des protocoles assurent l'équité et l'absence d'interblocage [Kordon 92] ;
3) un gestionnaire de synchronisation par synchronisation.

L'inconvénient de cette approche est qu'elle entraîne beaucoup de messages et donc un surcoût de gestion en ce qui concerne les ressources. La solution à ce problème est de tenter d'empêcher dés le départ la possibilité de migration des ressources. On a alors recours à une étude du réseau de Petri dans le but de trouver un partitionnement des ressources en prenant en compte le marquage coloré. On cherche alors à isoler des flots de contrôle selon des ensembles de couleurs distincts. Notons enfin, que le gestionnaire de synchronisation peut lui aussi être répliqué, dans le but de favoriser les synchronisations d'instances de processus faiblement couplés (on joue alors sur la répartition des couleurs des jetons transitant dans le réseau de Petri [Kordon 92]). Ainsi, lors d'une synchronisation de N processus, on place la réplique du serveur de synchronisations sur le même processeur que les N processus.

Modélisation des serveurs en fonction des approches avec VODEL-D

Le calcul du placement du prototype passe par la prise en compte des différents gestionnaires de l'architecture générique mis en place. C'est pourquoi dans le Tableau 32 et le Tableau 32, nous décrivons les objets VODEL-D associés pour chaque gestionnaire du prototype et en fonction des deux approches de répartition présentées précédemment.

Dans le cas de l'approche «systématique», tous les gestionnaires sont des DVSC actifs fixes, car un gestionnaire ne migre pas en cours d'exécution. En ce qui concerne le gestionnaire de synchronisation et de ressources, ils sont centralisés et regroupent les entités

qu'ils gèrent (respectivement les actions synchronisées et les ressources) et qui sont elles mêmes des DVSC, d'où l'attribut indissociable. Le gestionnaire de prototype est présent sur un site unique, celui où est effectué le lancement de l'exécution.

L'approche optimale entraîne une modification de la description du gestionnaire de ressources et de prototype. Les gestionnaires de ressources sont répartis et on partitionne les ressources en essayant de les placer sur le même site que les processus qui les consomment. En fonction des demandes, ces ressources sont susceptibles de migrer. En ce qui concerne le gestionnaire de prototype, il en existe un par site participant à l'exécution (d'où l'attribut Répliqué).

Tableau 31: Transformation des gestionnaires du prototype dans l'approche «systématique»

Gestionnaire du prototype	Type	Evolution	Hiérarchie	Regroupement	Répartition
Synchronisations	DVSC Actif	Fixe	Hiérarchie	Indissociable	Centralisé
Ressources	DVSC Actif	Fixe	Hiérarchie	Indissociable	Centralisé
Prototype	DVSC Actif	Fixe	Primitif	X	Centralisé

Tableau 32: Transformation des gestionnaires du prototype dans l'approche «optimale»

Gestionnaire du prototype	Type	Evolution	Hiérarchie	Regroupement	Répartition
Synchronisations	DVSC Actif	Fixe	Hiérarchie	Indissociable	Centralisé
Ressources	DVSC Actif	Fixe	Hiérarchie	Dissociable	Répliqué
Prototype	DVSC Actif	Fixe	Primitif	X	Répliqué

3.3. Conclusion

La génération d'applications à partir de réseaux de Petri est réalisée en prenant en compte des invariants structurels du modèle. A partir de ces invariants, on se ramène à des objets de génération de type processus, ressources et synchronisations. L'architecture de définition de l'architecture est donc calculée automatiquement. Le passage de l'architecture de définition à l'architecture d'exécution est réalisée de manière automatique en utilisant des gabarits de conception. Ainsi, chaque objet de génération est pris en charge par un gestionnaire spécialisé. L'exécution répartie du prototype ainsi créé est alors réalisée en précisant dans un fichier de configuration externe le placement et la réplication des gestionnaires et le partitionnement des objets de génération qu'ils gèrent. Toutes les étapes du processus de génération de code et de placement des composants logiciels sont résumées sur la Figure 41.

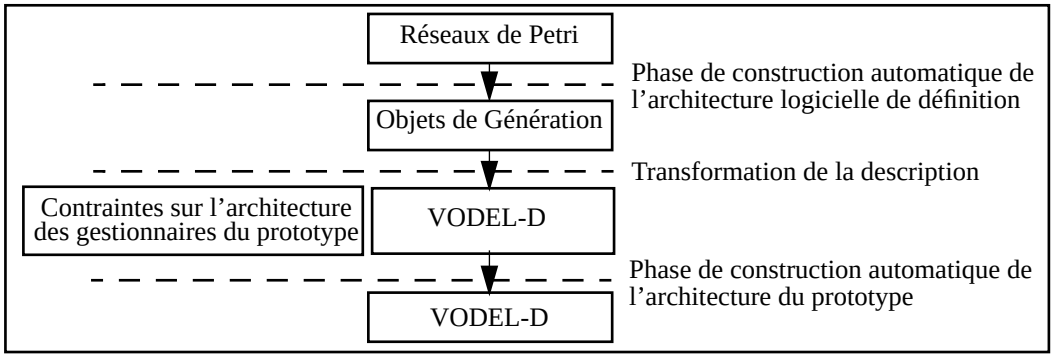


Figure 41 : Description des phases du calcul du placement lors de la génération d'une application

Un outil de génération de code, appelé CPN/Tagada, a été implémenté. Les prototypes générés par CPN/Tagada fonctionnent sur un ensemble de machines Unix connectées sur un réseau local. L'architecture générique du prototype mise au point s'est avérée robuste et adaptée à la génération de code.

Par contre, son implémentation s'est heurtée à des limitations du langage de génération choisi. Il a donc fallu faire face à des limitations de Ada 83 telles que :

- la difficulté de répartir des tâches Ada 83 (*Task*), car toutes les tâches d'un exécutable sont contenues dans un unique processus Unix ;
- à la gestion des rendez-vous synchrones distants.

Enfin, au niveau des algorithmes de placement, l'évaluation des coûts d'exécution et de communication des modules du prototype est difficile à partir d'une réseau de Petri coloré. Ainsi, par exemple, si le tir d'une transition est associé à une requête sur une base de données (coût de communication variable car dépendant de la requête et de la charge du système de bases de données), il est quasiment impossible de le détecter. Enfin, ces algorithmes ne prennent pas en compte tous les paramètres de puissance et de charge des machines disponibles et se basent sur une évaluation empirique des temps de production et de consommation d'une marque dans une ressource et du temps moyen d'exécution et de synchronisation des processus. La description de l'architecture matérielle ne sert qu'à indiquer les machines homogènes sur lesquelles le gestionnaire de prototype aiguillera les différents composants logiciels du prototype. Leur description est donc sommaire et contient principalement leur nom et leur adresse IP.

Pour combattre ces limitations, nous avons axé nos travaux dans deux directions. Dans un premier temps, nous avons travaillé à l'intégration des langages HADEL et VODEL et de la problématique de la génération de code [El kaim & al. 94] et du placement. Notre idée étant alors de renforcer l'importance des informations concernant l'architecture matérielle dans les algorithmes de placement. Puis, dans un second temps, prenant conscience que la génération de code réparti nécessitait un modèle sémantique plus riche (et donc de plus haut niveau) que les réseaux de Petri, nous avons travaillé à la mise au point de H-COSTAM [Kordon & al. 95]. Munis d'un modèle semi-formel restant compatible avec les réseaux de Petri et conçu pour la génération de code, il ne restait plus qu'à réécrire un nouveau générateur de code.

La section suivante présente les travaux actuels sur H-Tagada, notre nouveau générateur de code et sur les liens entre H-COSTAM et le langage VODEL.

4. Génération d'applications à partir de H-COSTAM

Nos observations basées sur des expérimentations avec CPN/Tagada, nous ont amenés à repenser, au sein de la méthode MARS, les techniques et les outils de prototypage dont nous disposions, afin d'offrir une plate-forme améliorée d'équilibrage d'application. Nous avons alors tenté :

1) de trouver un modèle opérationnel plus riche sémantiquement que les réseaux de Petri, mais sans perdre les capacités de validation du modèle réseau de Petri. L'idée étant d'encapsuler les réseaux de Petri dans une description de plus haut niveau. Nous avons été encouragés dans ce sens, par les résultats de travaux similaires sur Cab [Bruno & al. 95] et SANDS/CO-OPN [Hulaas 97 et Buchs & al. 96], basés sur des extensions du modèle réseau de Petri ;
2) d'offrir un modèle hiérarchique, pour exprimer des spécifications de grandes tailles lisibles, tout en améliorant le travail de modélisation ;
3) d'offrir des gabarits de conception concernant les canevas d'interaction entre composants logiciels. Notre volonté principale étant d'éviter les conflits sémantiques et

structurels compliquant la génération de code. Ces gabarits ont aussi été conçus pour optimiser la génération de code, la gestion des communications et des informations échangées.

4) d'offrir des gabarits de conception décrivant et paramétrant la création dynamique d'instances de composants logiciels. Ce besoin est directement issu de notre volonté de modélisation des applications client-serveur universelle et l'évolution dynamique de leur architecture. ces gabarits de conception s'appliquent sur des composants logiciels offrant un niveau de granularité similaire à celui des objets de génération ;

5) de prendre en compte la représentation de l'architecture matérielle sur laquelle serait réalisée l'exécution. A partir de cette description et de la description de l'architecture logicielle, il est possible de mettre au point des algorithmes de placement multi-critères ;

6) d'intégrer ces travaux dans le cadre de notre méthode MEDEVER pour offrir la traçabilité sur l'ensemble du cycle de développement (cf. Figure 42).

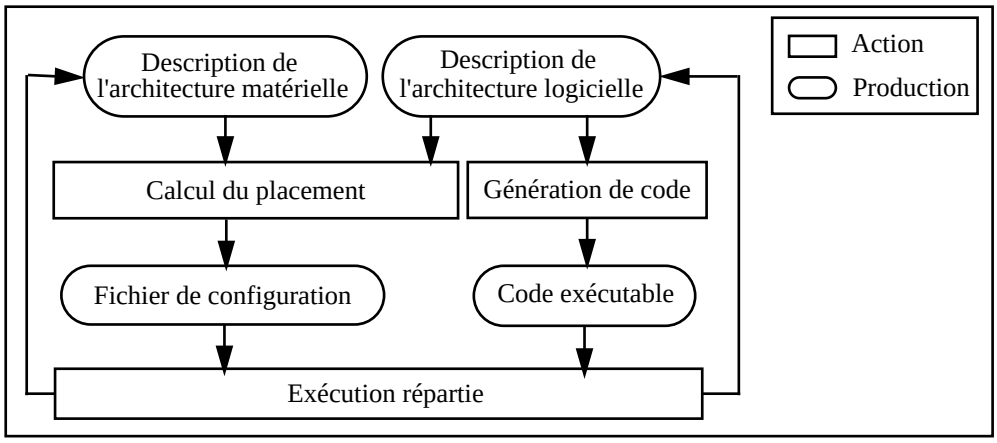


Figure 42 : Description des étapes de notre méthode d'équilibrage d'application

La prise en compte de tous ces besoins ont abouti à la création du modèle semi-formel H-COSTAM. H-COSTAM est adapté à la description opérationnelle des applications réparties. Du fait de l'utilisation de H-COSTAM, la structure du prototype générée a dû être modifiée et a donné naissance à un nouvel outil de génération de code appelé H-TAGADA (actuellement en cours de réalisation). La description de l'architecture matérielle est réalisée avec le langage HADEL et celle de l'architecture logicielle (déduite de H-COSTAM) avec notre langage VODEL-D.

Nous présentons donc dans la première section, le modèle H-COSTAM. Puis, nous décrivons les techniques de transformation automatique du modèle H-COSTAM en VODEL-D et nos premiers travaux sur les heuristiques de placement des prototypes générés.

4.1. Présentation de H-COSTAM

H-COSTAM (*Hierarchical COmmunicating STAte Machine*) est un formalisme dédié à la description opérationnelle de systèmes répartis dont les composants communiquent par échange de messages.

H-COSTAM fournit :

- une traçabilité via l'encapsulation des réseaux de Petri dans les entités H-COSTAM ;
- une gestion de la hiérarchie offrant des modèles lisibles et structurés ;

- des mécanismes de communication fortement typés (contenu des messages et comportement) ;
- la gestion de la généricité pour faciliter la paramétrisation et la réutilisation des composants principaux créés.

Une spécification H-COSTAM est composée de pages qui appartiennent à deux types de description : Macro et Micro. Une page de niveau Macro décrit les relations entre entités qui sont soit des sous systèmes (et qui contiennent des liens vers d'autres pages de niveau Macro) soit des machines à états (qui contiennent des liens vers d'autres pages de niveau Micro). Une page de niveau Micro décrit une machine à état et ses entités internes. Une page décrivant le comportement fonctionnel de l'environnement d'exécution est appelée page externe, alors qu'une page qui décrit le système est appelée une page interne. Lors de la génération de code, seules les informations contenues dans les pages internes sont utilisées, les pages externes représentant des modules logiciels externes déjà implémentés.

Dans la suite de ce paragraphe, nous présentons les composants élémentaires du modèle au niveau Micro, puis ceux au niveau Macro.

4.1.1. Le niveau Macro

Le niveau Macro a été conçu pour décrire la structure d'un système. Une page de niveau Macro correspond à un sous-système non séquentiel possédant ses propres interfaces (sauf pour la page Macro racine). Les composantes d'un tel sous-système sont soit des sous-systèmes (décrits par une autre page de niveau Macro), soit des processus séquentiels (décrits par une page de niveau Micro).

Les relations entre les composantes inclues dans la page sont exprimées au moyen de media de communication. On dénombre quatre types de media :

1) **les multi-rendez-vous** correspondent à une synchronisation entre N composantes. Cette synchronisation peut-être gardée par un prédicat et permet éventuellement aux participants d'échanger des messages.
2) **les liens** correspondent à un échange synchrone de données régi par un comportement spécifique. Les comportements actuellement mis en oeuvre sont FIFO (l'ordre des messages transmis est préservé) et LIFO (l'ordre des messages transmis est inversé) et RANDOM (les messages sont transmis de manière aléatoire). Les liens peuvent être connectés à un nombre arbitraire d'entités.
3) **les appels de procédure distants** (*RPC*) relient deux composants entre eux, l'un agissant comme un serveur et l'autre comme un client. Côté client, les RPC sont vus comme une action atomique et côté serveur, ils correspondent à deux liens FIFO (l'un transmettant les paramètres et l'autre les résultats de l'appel).
4) **les usines** (*factory*) véhiculent des messages particuliers provoquant la création de nouvelles instances d'un ou plusieurs processus. Les entrées d'une usine sont des actions de machines à états qui envoient des valeurs dans cette usine. Les sorties, quant à elles, sont reliées à des états initiaux de machines à états. Chaque envoi d'une valeur dans une usine par des entrées provoque la génération d'une machine à état avec des variables internes initialisées à partir de cette valeur et ceci pour chaque machine à états reliée à une des sorties de l'usine.

Un processus communique avec le monde extérieur par l'intermédiaire de media, qui définissent des protocoles d'interaction. Chaque medium de communication est fortement typé à la fois par le type de données qu'il véhicule et par son comportement. La hiérarchie mise en place avec les modules permet l'héritage de types et de constantes par des pages modules contenues dans une autre page module. Les types sont utilisés dans H-COSTAM pour définir le format des données qui traversent les media de communication (au niveau Macro et Micro) ou pour déclarer des variables dans le contexte du pro-

cessus (au niveau Micro). Les types sont définissables par l'usager dans la page où ils sont utilisés ou sont importés d'une entité qui contient la page.

La représentation graphique des différents constituants du formalisme graphique Macro, disponibles dans l'éditeur Macao [Macao 97], est donnée dans la Figure 43.

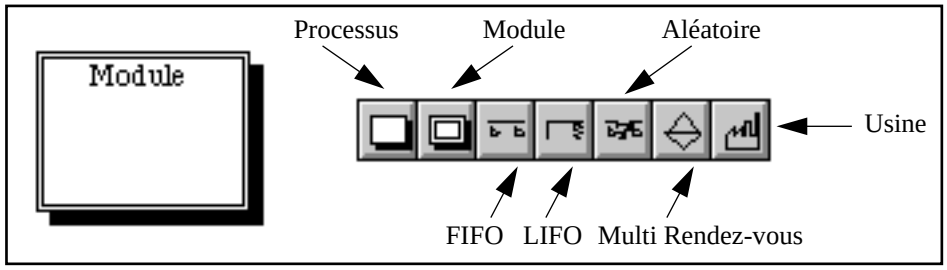


Figure 43 : Les composants élémentaires de H-COSTAM au niveau Macro

4.1.2. Le niveau Micro

Une page de niveau Micro décrit un composant élémentaire : le processus. Comme les sous-systèmes, un processus dialogue avec le monde extérieur par l'intermédiaire des media de communication.

Un processus représente un modèle de comportement instancié soit statiquement (instances définies dans la déclaration), soit dynamiquement (par le biais de messages provenant d'usines). Le contexte d'un processus est composé de variables propres à chacune des instances créées durant l'exécution du système. Les ressources privées, partagées par toutes les instances, sont définies au moyen de liens locaux.

Une page Micro contient un automate état-transition à la sémantique proche de celle des réseaux de Petri. Cet automate composé d'actions et d'états, décrit une machine à état correspondant au comportement statique du processus. Les actions peuvent être gardées par des prédicats portant sur des variables du contexte d'exécution et/ou de messages provenant des media. Les actions peuvent également produire des messages en direction de media d'interfaces ou modifier le contenu des variables de contexte au moyen d'opérateurs (identité, successeur, produit, etc.) dont le nombre est volontairement limité en fonction des contraintes définies dans les réseaux de Petri Bien Formés [Chiola & al. 91].

La représentation graphique des différents constituants du formalisme graphique Micro, disponibles dans l'éditeur Macao [Macao 97], est donnée dans la Figure 44.

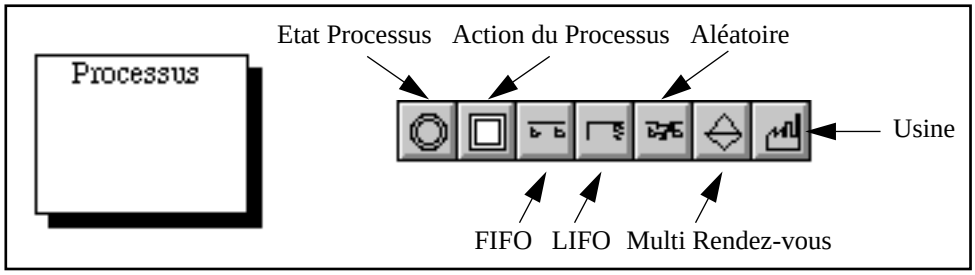


Figure 44 : Les composants élémentaires de H-COSTAM au niveau Micro

4.1.3. Un exemple de modèle H-COSTAM

Dans l'exemple de la figure 45, le processus «Générateur» envoie un message avec la valeur «valeur1» à l'usine. Cette dernière provoque alors la création d'une instance du processus «Généré» à la réception de ce message (avec sa variable interne initialisée à la valeur «Valeur1»).

Le processus «Générateur» envoie ensuite un message avec la valeur «valeur2» à l'usine qui va de nouveau créer une instance du processus «Généré» (mais cette fois ci avec sa variable interne initialisée à la valeur «valeur2»). Le processus «Générateur» se termine

en même temps que les deux instances du processus «Généré», grâce au multi-rendez-vous «Termine».

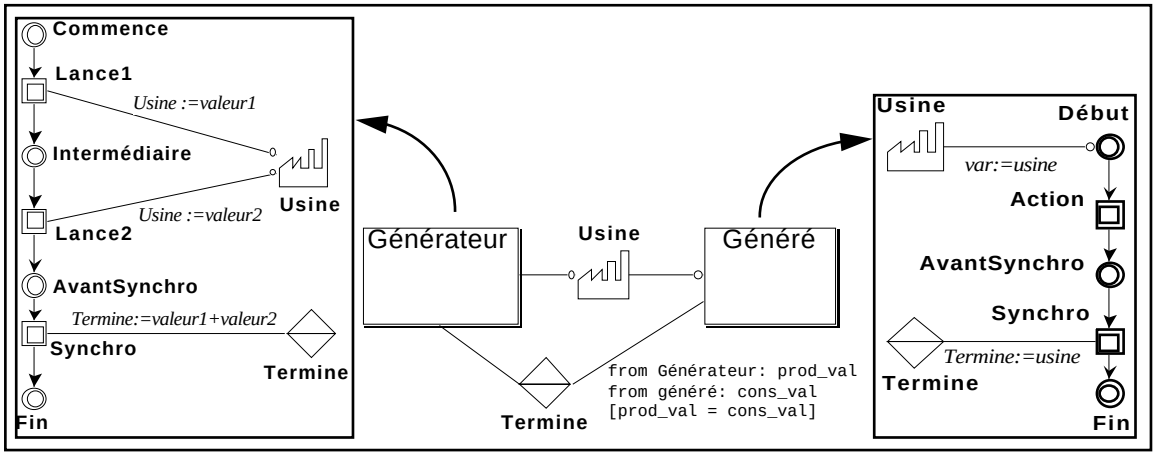


Figure 45 : Un exemple de création dynamique de processus en H-COSTAM

4.2. Placement des objets H-COSTAM

Nous avons appliqué notre méthode MEDEVER au placement d'objets H-COSTAM sur réseau local de stations de travail Unix homogènes.

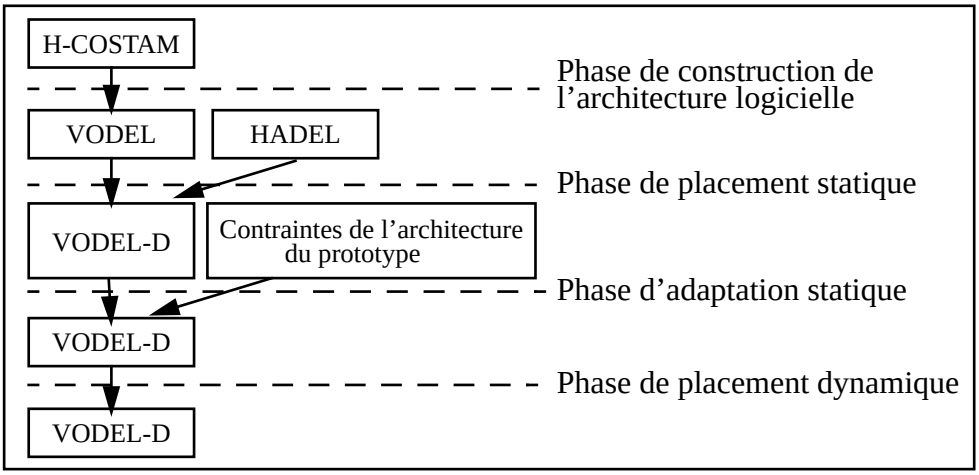


Figure 46 : Les différentes phases de placement

Le placement est alors réalisé en quatre phases (cf. figure 46) :

1) *phase de construction de l'architecture logicielle*. Le placement est d'abord calculé statiquement et à priori. Ce placement est réalisé sur le modèle opérationnel indépendamment du fait que le code ait été généré ou non. Ainsi, le modèle H-COSTAM est transformé en un modèle VODEL, décrivant l'architecture logicielle à placer.
2) *phase de placement statique*. Le modèle VODEL est combiné avec le modèle HADEL décrivant l'architecture matérielle pour calculer le placement statique des différentes entités logicielles de l'application. Enfin, la prise en compte des limitations du générateur de code et du prototype produit conduit à un lissage du placement et à un réajustement de la granularité des composants à placer.
3) *phase d'adaptation statique*. Si le prototype généré le permet, les évolutions dynamiques de l'application sont prises en compte par transformation du modèle VODEL en modèle VODEL-D. Cette transformation entraîne un re-calcul dynami-

que de la granularité et du placement des entités logicielles.

4) *phase d'adaptation dynamique*. L'exécution répartie de l'application est désormais contrôlée par l'exécutif qui réagit dynamiquement à tous les évènements susceptibles de modifier l'architecture logicielle du prototype (tels que la création de processus ou la migration de DVSC).

Nous détaillons chacune de ces quatre phases dans la suite de ce paragraphe.

4.2.1. Phase de construction : transformation de H-COSTAM en VODEL

Très peu d'efforts sont nécessaires pour transformer un modèle H-COSTAM en un modèle VODEL-D. Seules deux classes de VODEL-D ont dû être redéfinies :

- la classe processus a été redéfinie pour représenter une machine à états séquentielle composée de places et d'actions ;
- la classe usine n'existant pas a dû être créée et consiste en fait en une classe abstraite de création dynamique de processus.

Ainsi, les entités actives de H-COSTAM qui sont les processus, les media actifs et les usines ont été transformés en DVSC actifs dans VODEL-D. Les entités passives que sont les media passifs (les liens H-COSTAM) sont transformées en objets canal avec une politique de gestion positionnée à LIFO, FIFO ou générique dans VODEL-D.

La notion de page interne est représenté par la hiérarchie d'entités VODEL-D. Les pages externes sont transformées en DVSC actifs de type processus ou en DVSC passifs, suivant leur nature.

Les transformations effectuées pour passer d'un modèle H-COSTAM à un modèle VODEL-D sont résumées au Tableau 33.

Tableau 33: Transformation des objets H-COSTAM en objets VODEL-D

Objets H-COSTAM	**Objets VODEL-D**
Processus	DVSC actif de type processus
Media Passif : lien	Canal
Media actif : rendez-vous	DVSC actif de type rendez-vous
Usine	DVSC actif de type Usine
Page interne	Hiérarchie de DVSC
Page externe	DVSC actif de type processus ou passif

4.2.2. Phase de placement statique

Le nombre et la nature des paramètres pris en compte dans l'équilibrage d'application influent sur la qualité du choix du placement. Un grand nombre de paramètres améliore globalement la qualité de la décision, mais du fait que l'environnement est réparti, leur diffusion et leur traitement coûte cher. Il faut donc restreindre le nombre de paramètres utiles, les choisir et les combiner afin d'obtenir un indicateur significatif. Ces paramètres sont de deux types : statiques et dynamiques. Les paramètres statiques n'évoluent pas au cours de la vie du système et sont initialisés une fois pour toutes (par exemple la puissance d'une machine est un paramètre fixe). Les paramètres dynamiques sont récoltés et sont diffusés périodiquement aux services logiciels qui s'en servent.

Comme le calcul du placement dans cette phase est réalisé à priori, seules les informations statiques sont utiles. Concernant l'architecture matérielle, seule la liste des machines utilisées lors de l'exécution répartie est nécessaire. En effet, l'architecture matérielle considérée est composée de machines homogènes sur une réseau local de stations de travail.

La description de l'architecture logicielle est quant à elle directement issue du modèle H-COSTAM et est réalisée de deux manières complémentaires :

1) par transformation du modèle H-COSTAM en réseaux de Petri et application des algorithmes définis dans CPN/TAGADA ;
2) par transformation du modèle H-COSTAM en modèle VODEL-D. Le modèle ainsi obtenu est alors utilisé pour effectuer des regroupements hiérarchiques d'entités logicielles (DVSC) et pour caractériser les relations de communication (DVSL - DVGL) et de coopération (attraction ou répulsion) entre ces entités logicielles.

Des travaux sont actuellement en cours pour rechercher des heuristiques de transformation adaptées au placement statique dans le second cas. Nous présentons dans la suite quelques travaux préliminaires sur ce sujet.

Caractérisation de la granularité des DVSC lors du passage H-COSTAM-VODEL

La caractérisation de la granularité des DVSC à créer est réalisée par analyse des objets H-COSTAM. On cherche alors à évaluer pour l'équilibrage d'application :

- le nombre d'états et d'actions à franchir sur le chemin d'exécution le plus long dans un processus. On cherche ainsi à savoir si ce DVSC est plutôt orienté vers des fonctions de calcul, de communication ou de synchronisation ;
- les coûts de communication qu'il engendre, en recherchant les différents types de media auxquels il est connecté. Les media passifs étant généralement implémentés par des structures de données, ils sont potentiellement réplicables. Par contre, les media actifs représentant des points de synchronisation actifs (par multi-rendez-vous). C'est pourquoi on tente de minimiser leur influence en créant des relations d'attraction entre les DVSC qui les utilisent.

et pour l'équilibrage de charge :

- les coûts de création dynamique et de gestion de processus via des objets H-COSTAM «usine». Les usines sont gérées par le gestionnaire de prototype et le gestionnaire de processus «cible». L'intérêt des usines, est qu'il est possible de leur associer des règles de placement, évaluées dynamiquement. Le fait qu'elles soient déclarées dans le modèle, de manière statique, entraîne le choix priori de (ou des) algorithme(s) de placement dynamique. On pourrait ainsi choisir de créer tout nouveau processus sur la machine la moins chargée ou de regrouper les processus par groupes en fonction du contenu du message de création.

Quelques heuristiques de placement des DVSC

Nous présentons ci-dessous quelques heuristiques de placement structurelles (ie. basés sur le modèle de description de VODEL) :

1) un DVSC qui ne communique avec aucun autre est placé en dernier sur les machines les moins chargées ;
2) un DVSC réplicable qui communique avec d'autres DVSC, doit être instancié sur les différents sites ;
3) les DVSC passifs sont placés sur les Machines qui possèdent des ressources disques ;
4) chaque processus est un DVSC actif dont les instances sont réplicables sur des machines distinctes.

4.2.3. Phase d'adaptation

L'exécutif désigne tous les composants fixes qui vont collaborer avec le prototype lors de son exécution. L'architecture de cet exécutif est directement liée aux objets H-COSTAM qui sont utilisés pour la génération de code. Ainsi, l'exécutif est composé d'une bibliothèque d'exécution et de quatre gestionnaires d'exécution (cf. Figure 47).

La bibliothèque d'exécution fournit une interface entre le prototype et son environnement d'exécution. C'est à travers elle que toutes les actions du prototype sont exécutées. Cette bibliothèque est utilisée pour générer du code compact et indépendant de l'environnement d'exécution.

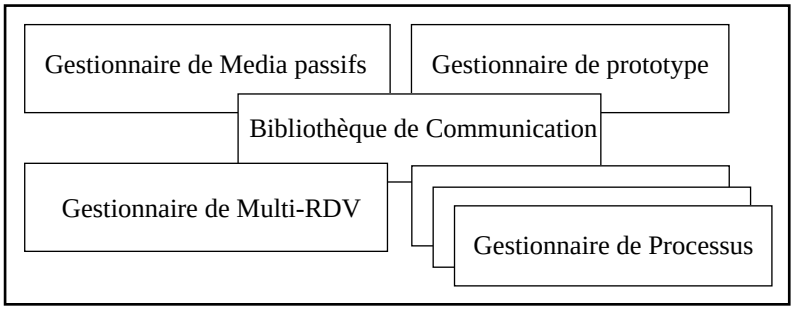


Figure 47 : Architecture d'un prototype généré par H-TAGADA

Les quatre gestionnaires d'objets H-COSTAM s'exécutent en même temps que le prototype et interagissent entre eux et avec le prototype pour effectuer des tâches externes à un processus du prototype. Les techniques que nous proposons pour la génération de code sont proches de celles précédemment implémentées dans CPN/Tagada. Néanmoins, les différents gestionnaires ont des fonctionnalités accrues :

- **le gestionnaire de prototype**. Comme dans CPN/Tagada, il est chargé du lancement de l'ensemble des entités et de la synchronisation globale des éléments du système. L'apport majeur de ce gestionnaire par rapport au précédent est sa capacité à gérer la création dynamique de processus ;
- **le gestionnaire de processus**. Il gère l'exécution, la création et la terminaison des processus. Chaque processus est implémenté par un gestionnaire de processus.
- **le gestionnaire de media passifs**. Il se charge de la gestion des messages au travers des objets H-COSTAM de type lien. Il doit implémenter les politiques d'ordonnancement de messages spécifiés dans le modèle (FIFO, LIFO et Random).
- **le gestionnaire de multi-rendez-vous**. Il gère les synchronisations entre les processus du modèle (création, vie et terminaison).

Le placement du prototype généré passe par la recherche de la répartition physique des différents gestionnaires. Nous étudions pour chacun d'entre eux leurs spécificités et les contraintes qu'ils induisent.

Le gestionnaire de media passif

Utilisables directement par envoi et réception de messages, ces media sont également utilisés pour évaluer des prédicats à partir de leur contenu afin de permettre une consommation conditionnelle des messages. Afin de simplifier la structure du gestionnaire, on impose que les prédicats étant partie prenante dans un prédicat soient placés sur la même machine.On définit alors la notion de groupe de media qui est le plus petit ensemble de media passifs qui ne partagent pas un seul prédicat. Le placement est alors réalisé par groupe de media passif.

Le gestionnaire de media actifs

Chaque action qui représente une synchronisation entre différents processus devient un DVSC actif. Le serveur de synchronisation doit gérer toutes les synchronisations. Suivant l'implémentation du serveur de synchronisation, elles sont :

- soit toutes regroupées dans un DVSC actif, indissociable et centralisé si le serveur est centralisé ;
- soit regroupées dans un DVSC actif, dissociable et réplicable s'il existe un parti-

tionnement possible du serveur de synchronisations en plusieurs serveurs indépendants fonctions des instances de processus gérées.

Le gestionnaire de processus

Chaque instance de processus créée dispose de son propre gestionnaire de processus et d'un contexte d'exécution qui lui est propre. Lorsqu'un processus est dans un état, son travail consiste à chercher quelle action il va exécuter ensuite. Ce choix est réalisé soit directement (s'il n'existe qu'une seule action successeur), soit en évaluant des prédicats à partir des variables internes du processus, soit en faisant évaluer des prédicats dépendant d'autres entités du prototype. Dans ce dernier cas, le processus envoie des requêtes aux gestionnaires de processus, de media passifs et actifs et réagit en fonction des réponses qu'il reçoit. Lorsqu'un processus réalise une action, son travail consiste à implémenter les tâches associées à cette action et à désigner l'état successeur de l'action.

Le placement des processus est liée principalement à trois critères : les coûts de communication, les coûts d'exécution et les coûts de gestion. La minimisation des coûts de communication (à travers les media passifs) et de synchronisation (dans les media actifs) est directement lié au placement et à l'implémentation des media actifs et passifs. Les coûts de gestion sont liés au nombre d'actions franchies sur le chemin parcourant le plus grand nombre d'états et aux évaluations de prédicats faisant intervenir des gestionnaires tiers ou non.

Le gestionnaire de communication

Le gestionnaire de communication est utilisé pour offrir des communications point à point entre le prototype et les différents gestionnaires. Au niveau local, les communications sont réalisées via les *IPC system V* pour les prototypes en C et via l'environnement d'exécution ADA pour les prototypes en Ada. Les communications distantes utilisent le protocole TCP/IP et les sockets pour le prototype en C et TCP/IP et l'environnement d'exécution Ada pour le prototype Ada.

Le gestionnaire de prototype

Le gestionnaire de prototype envoie des requêtes aux gestionnaires de processus pour la création statique (ie. qu'il lance au démarrage) et dynamique (ie. qu'il créé par envoi de messages à des usines) de processus. Il dialogue avec les gestionnaires de media passifs et actifs, via le gestionnaire de communication.

Lors du lancement de l'exécution répartie du prototype, le gestionnaire du prototype est lancé en premier et veille au lancement et à la synchronisation des entités locales. Puis, il s'assure que tous les prototypes sont bien tous présents et prêts au lancement de l'exécution répartie. Puis, une fois que tous les gestionnaires de prototype sont prêts, le gestionnaire qui se trouve sur la machine qui a servi au lancement de l'exécution répartie, donne le signal de début d'exécution.

Chaque gestionnaire de prototype peut conserver des traces d'exécution et échanger des informations de charge avec les autres gestionnaires de prototypes. La terminaison du prototype est détectée quand plus aucun processus ne s'exécute et qu'aucune demande de processus dynamique n'est en cours de traitement. Lors de l'exécution répartie, la terminaison répartie est réalisée par coopération des gestionnaire de processus.

Conclusion : le fichier de pondération du placement

Souvent le placement calculé est d'un grain trop fin par rapport aux capacité de l'outil de prototypage. En effet, si celui gère les synchronisations de manière centralisée, il est inutile de calculer la duplication optimale des synchronisations. C'est pourquoi, il est nécessaire de pondérer le placement du prototype par un fichier de configuration adapté. A l'heure actuelle, seules les indications concernant l'implémentation des différents gestionnaires de media (centralisé ou réparti) et la gestion des processus légers sont indiquées dans le fichier de configuration.

Sur chaque site s'exécute un gestionnaire de prototype, qui collabore avec le gestionnaire de prototype maître (canevas d'interaction de type maître-esclave). Les gestionnaires de media passifs sont répartis sur les machines en fonction des liens de communication qu'ils gèrent. Les instances de processus s'exécutent sur les sites sur lesquels ils ont été créés de manière statique ou dynamique. Enfin, le gestionnaire de multi-rendez-vous est réplicable en fonction des ensembles de processus qui se synchronisent. Si tous les ensemble de processus sont disjoints, il est possible de répliquer les gestionnaires de synchronisation, dans le cas contraire on tentera de minimiser les coûts de communication en rapprochant les gestionnaires des processus. Les différentes techniques de répartition des gestionnaires du prototype sont résumées sur la figure 48.

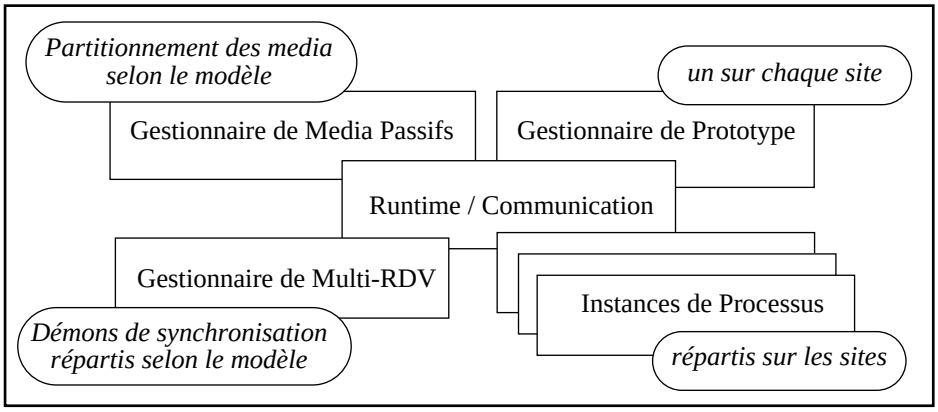


Figure 48 : Architecture répartie du prototype généré par H-TAGADA

4.2.4. Phase de placement dynamique

La prise en compte du placement dynamique dans le prototype généré est nécessaire du fait que :

- le placement statique est moins flexible que le placement dynamique ;
- l'intensité des communications et les coûts d'exécution des entités logicielles ne sont pas connus à l'avance ;
- les processus n'ont pas obligatoirement le même comportement d'une exécution à l'autre.

De plus les processus gérés ayant une durée de vie assez longue, il devient intéressant de pouvoir les migrer (cf. [Nuttal 94] pour plus d'informations sur la migration de processus).

Rappelons que le prototype généré par H-TAGADA est composé de plusieurs exécutables identiques qui s'exécutent sur des machines homogènes. Au sein du prototype généré par H-Tagada, la gestion de l'équilibrage de charge dynamique sera déléguée au gestionnaire de prototype. En effet, seul ce dernier dispose d'une vision locale cohérente de l'exécution du prototype.

La charge locale est calculée en fonction :

- du nombre d'instances de processus qui ont été créées au démarrage du prototype et qui s'exécutent à un instant T ;
- du nombre de gestionnaires de synchronisations ;
- du nombre de processus créés dynamiquement par des usines ;
- du nombre de liens de communications gérés par le gestionnaire de media passif.

Les échanges d'informations se font entre les instances du gestionnaire de prototype. Ce dernier est chargé de collecter les charges et de les diffuser périodiquement. Lors de la création d'un nouveau processus, l'usine envoie un message de création de processus au

gestionnaire de prototype, qui se charge d'envoyer un message de création dynamique au gestionnaire des processus sur la machine appropriée. Le gestionnaire de prototype est donc chargé de la gestion de la charge et éventuellement de la migration des processus.

4.3. Conclusion

Le générateur de code H-Tagada est en cours d'implémentation et produira du code Ada 95 et du code C. Le prototype en Ada 95 aura un noyau de communication basé sur l'environnement d'exécution d'Ada 95 pour les communications locales et sur GLADE [ACT 97] pour le placement et la gestion des tâches et des partitions (au sens ADA 95).

La faisabilité de la traduction d'un modèle H-COSTAM en programme C a été expérimentée récemment [Diot 96]. Pour cela, nous avons généré «à la main» un prototype à partir d'un modèle H-COSTAM. Nos premiers résultats montrent que l'utilisation du modèle H-COSTAM simplifie la génération de code, la gestion des communications et du placement des composants générés. Ceci notamment grâce à la structuration des composants logiciels induite par le formalisme et à l'utilisation de liens de communications fortement typés.

5. Application à un exemple de chaîne d'assemblage automobile

Dans cette section, nous reprenons l'exemple présenté dans [Kordon 92], en comparant les approches de génération d'applications à partir de réseau de Petri et de H-COSTAM.

5.1. Le modèle initial

Considérons la chaîne d'assemblage d'une carrosserie automobile (Figure 49). Les caisses proviennent d'un autre atelier. Deux ailes avant (gauche et droite) sont tout d'abord soudées. Ensuite le hayon et le capot sont ajoutés. Enfin, les portières sont assemblées avant que la carrosserie complète ne parte en direction d'un autre atelier.

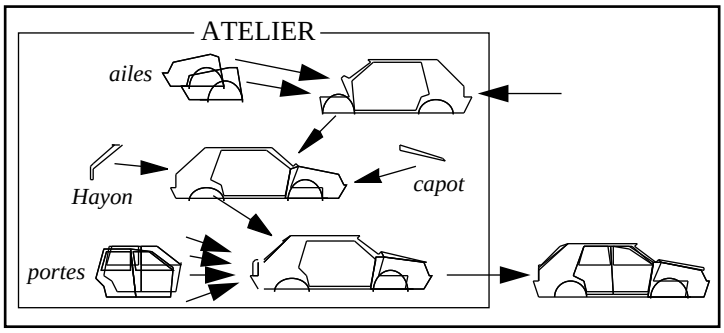


Figure 49 : Etapes de construction d'une carrosserie automobile

A part la caisse, qui provient d'une autre chaîne d'assemblage, les ailes avant, le coffre, le hayon et les quatre portières sont façonnées dans l'atelier lui-même :

- un robot produit les ailes avant, réalisant alternativement un coté gauche, puis un coté droit. En sortie de cet unité fonctionnelle, un aiguilleur se charge de séparer les deux types de pièces afin de les mettre dans des files d'attente correspondantes.
- le coffre et le hayon sont composés de deux pièces (partie interne et partie externe) qui, une fois produites par deux robots distincts, sont assemblées. Le robot produisant la partie interne dépose alors la pièce complète dans une file d'attente.
- une batterie de quatre robots produit les quatre types de portières (avant gauche, avant droit, arrière gauche et arrière droit). En sortie de cette unité de production,

un aiguilleur place les différents types de portières dans des files d'attentes différentes (une par type de portière).

5.2. Prototypage à partir du réseau de Petri

L'ensemble du système a été modélisé dans [Kordon 92] par le réseau de Petri de la Figure 50. Le modèle présenté, pour les marquages initiaux M1 et M2 définis sur la Figure 50, s'arrête automatiquement après production de 50 carrosseries, soit 50 capots et hayons, 100 ailes et 200 portières. Ce choix a été réalisé à des fins d'évaluation d'un modèle terminant son exécution dans un temps borné (son déroulement complet correspond à l'activation de 1819 transitions).

5.2.1. Description des objets de génération

Ce réseau de Petri est décomposable en 13 objets de génération processus, représentés en grisé sur la Figure 50. Les Processus *prod_p_e* et *prod_p_i* sont instanciés quatre fois. Les autres n'existent qu'en un seul exemplaire. Ce modèle comporte également 3 actions synchronisées (*assemble_h, assemble_c et assemble_p*) et 13 ressources (*ailes_avants, ailes_ag, ailes_ad, caisses, caisses_et_ailes, hayons, capots, caisse_sans_porte, p_ad, p_ag, p_dd, p_dg et portes*).

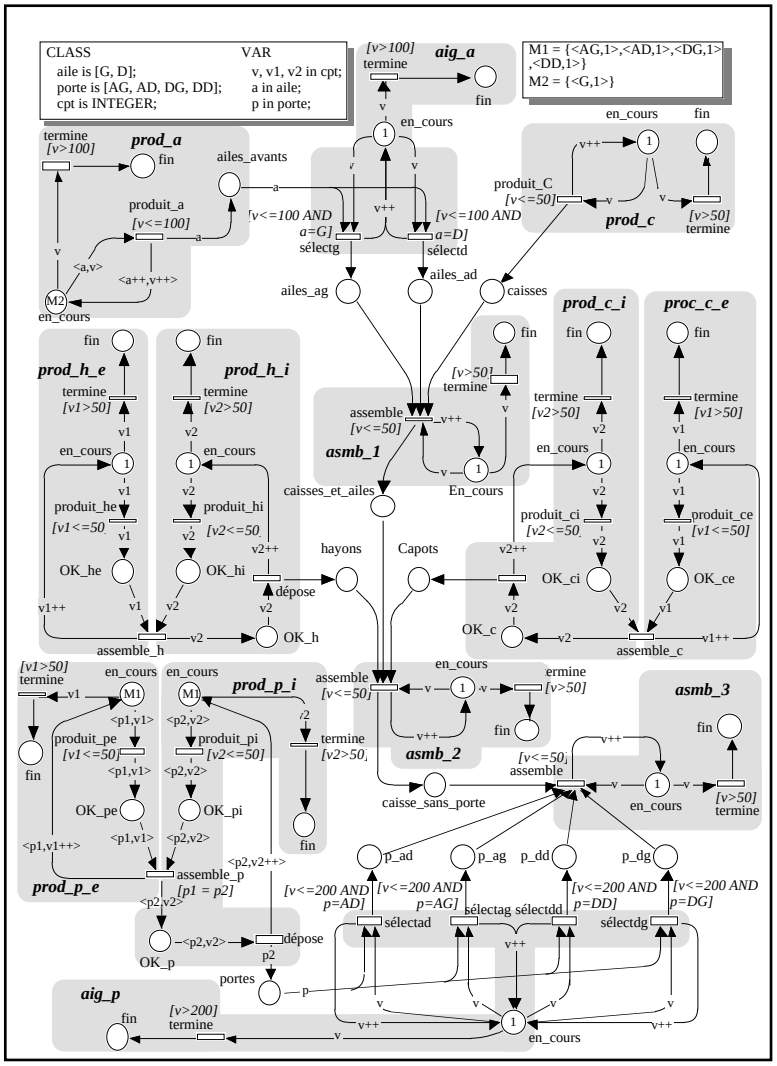


Figure 50 : Modélisation de l'atelier de montage avec des réseaux de Petri

5.2.2. Des objets de génération vers VODEL-D

Si on applique les procédures de transformation présentées dans la Section 3.2.3., on obtient le schéma de la Figure 51. Toutes les ressources ont été transformées en DVSC passifs et tous les processus et actions synchronisées en DVSC actifs.

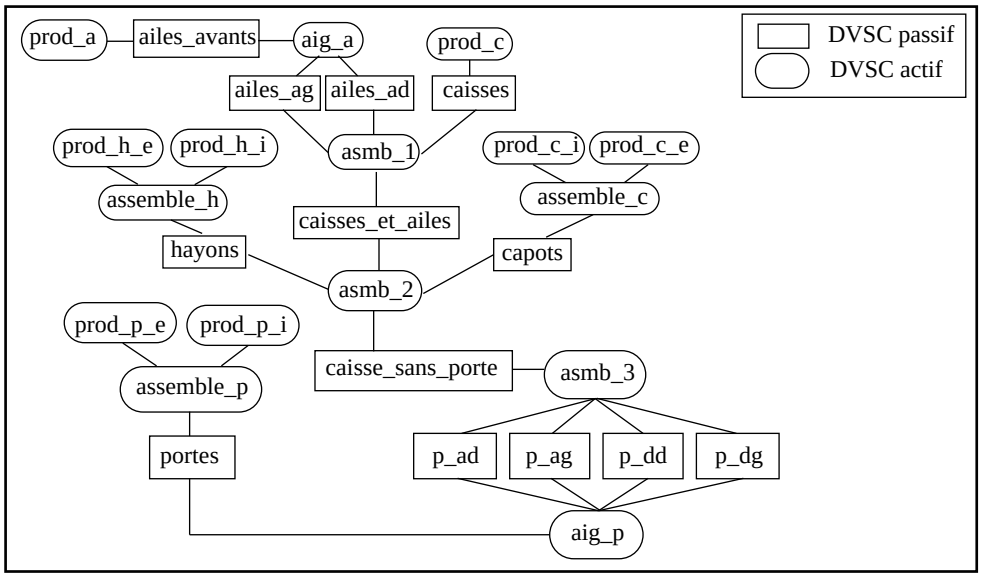


Figure 51 : Transformation des objets de génération en objets VODEL-D

5.2.3. Placement des composants logiciels

Il reste donc à regrouper et à placer les composants logiciels en fonction du nombre de machines disponibles. Une heuristique, comprenant trois règles de placement issues d'observations sur le code généré [Kordon 92], peut être utilisée ici :

1) dans un modèle producteur-consommateur impliquant des ressources, il est performant de placer les ressources sur le même site que les processus consommateurs ;
2) lorsque N processus se synchronisent via une action, il est préférable de placer le serveur de synchronisations associé sur le même site que les clients pouvant annuler des demandes de synchronisation ;
3) lorsque certaines parties d'un modèle peuvent être dépliées, il est intéressant de grouper des «instances» des parties ainsi caractérisées sur différents sites afin de maximiser le parallélisme induit au niveau du pliage. S'il n'existe pas d'interactions entre les différentes instances, certaines ressources et certaines actions synchronisées peuvent être dupliquées.

L'application de la règle de placement 2 entraîne le regroupement des DVSC actifs centralisés du modèle de la Figure 51, dans des DVSC actifs centralisés de niveau supérieur. Ainsi, on regroupe (cf. Figure 52):

- *assemble_h, prod_h_i et prod_h_e ;*
- *assemble_c, prod_c_i et prod_c_e ;*
- *assemble_p, prod_p_i et prod_p_e.*

L'application de la règle 3, sur l'action synchronisée assemble_p et sur les quatre instances des processus *prod_p_i* et *prod_p_e*, entraîne une duplication de l'action synchronisée. Ainsi pour chaque paire de processus *prod_p_i* et *prod_p_e*, initialisés avec le même marquage, on associe une action synchronisée. On construit alors quatre composants logiciels actifs dupliqués.

Enfin, l'application de la première règle entraîne la construction des DVSC suivants :

- *ailes_avants, aig_a ;*
- *ailes_ag, ailes_ad, caisses, asmb1 ;*
- *hayons, caisses_et_ailes, capots et asmb_2;*
- *caisses_sans_porte, asmb_3, p_ad, p_ag, p_dd, p_dg ;*
- *portes et aig_p.*

Enfin, pour minimiser les coûts de synchronisation, nous regroupons les DVSC *asmb_1*, *asmb_2*, *asmb_3* et *aig_p*. Bien entendu, en fonction des performances attendues et de l'architecture matérielle dont on dispose, on peut décider de ne pas les regrouper.

Le résultat des regroupements réalisés sont présentés graphiquement sur la figure 52 et en détail dans le Tableau 34.

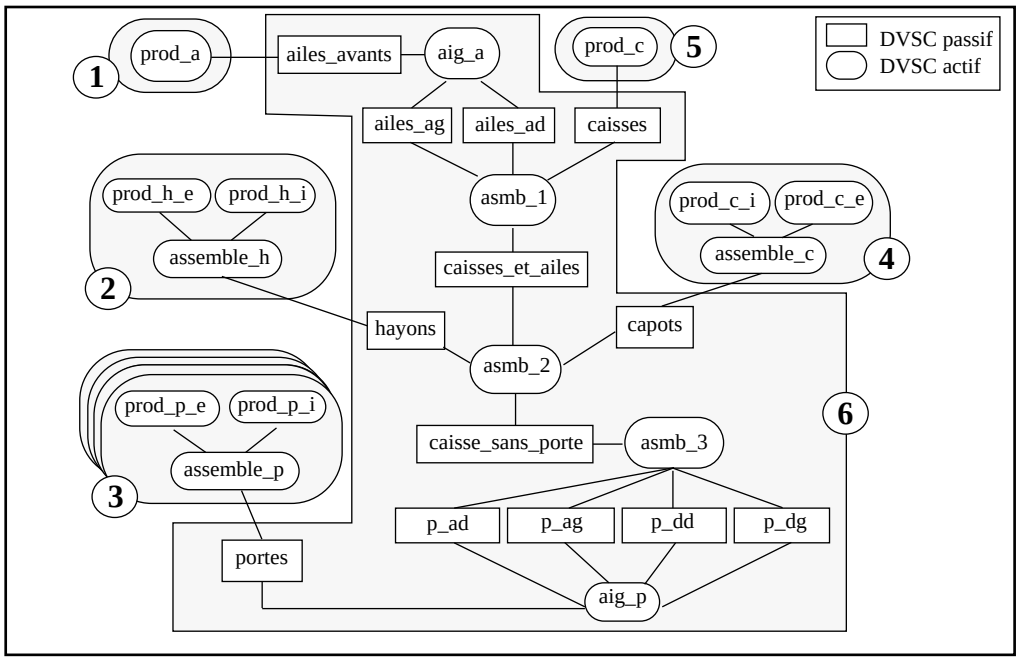


Figure 52 : Regroupement des DVSC avant placement.

Tableau 34: Les composants logiciels VODEL-D issus du calcul du placement

Nom	Contient	Type	Evolution	Hiérarchie (nombre de niveaux)	Regroupement	Répartition
Prod_ailes (1)	prod_a	Actif	Libre	Primitif	Indissociable	Centralisé
Prod_hayons (2)	prod_h_e, prod_h_i, assemble_h	Actif	Libre	Hiérarchique (1)	Indissociable	Centralisé
Prod_portes (3)	prod_p_e, prod_p_i, assemble_p	Actif	Libre	Hiérarchique (1)	**Dissociable**	**Réplicable**
Prod_capots (4)	prod_c_e, prod_c_i, assemble_c	Actif	Libre	Hiérarchique (1)	Indissociable	Centralisé
Prod_caisses (5)	prod_c	Actif	Libre	Primitif	Indissociable	Centralisé
Assemblage (6)	ailes_avants, aig_a, ailes_ag, ailes_ad, caisses, asmb1, asmb2, asmb3, caisses_et_ailes, caisse_sans_porte, p_ad, p_ag, p_dd, p_dg, aig_p, portes	Actif	**Fixe**	Hiérarchique (1)	Indissociable	Centralisé

On remarque que pratiquement tous les DVSC créés ont un seul niveau de hiérarchie. Ceci est principalement du au fait que l'architecture est déduite automatiquement des objets de génération qui sont des entités primitives déduites d'un réseau de Petri aplati. Tous les DVSC sont libres de se déplacer, sauf le DVSC Assemblage qui est fixe du fait du nombre d'entités passives et de synchronisations qu'il gère. Enfin, *Prod_porte* est un DVSC dissociable et peut être répliqué sur plusieurs machines. Il est alors décomposé en quatre DVSC actifs de création de portes (porte avant ou arrière, à droite ou à gauche du véhicule).

Un placement possible des DVSC en fonction du nombre de machines disponibles est présenté sur le Tableau 35. Le placement a été réalisé en appliquant les règles suivantes :

- les DVSC non primitifs qui contiennent une synchronisation entre des DVSC qu'ils agglomèrent se repoussent ;
- les DVSC qui sont primitifs s'attirent ;
- les DVSC réplicables sont répartis dés que le nombre de machines est suffisant.

Tableau 35: Placement des DVSC sur un nombre maximum de 10 machines

Nombre de machines	Machine 1	Machine 2	Machine 3	Machine 4	Machine 5	Machine 6	Machine 7	Machine 8	Machine 9
1	1, 2, 3, 4, 5, 6								
2	1, 2, 4, 5	3, 6							
3	1, 4	2, 5	3, 6						
4	1,4	2	5	3,6					
5	1,4	2	5	3	6				
6	1	4	2	5	3	6			
7	1	4	2	5	3	6 (ag, ad)	6 (dg, dd)		
8	1	4	2	5	3	6 (ag, ad)	6 (dg)	6 (dd)	
9	1	4	2	5	3	6 (ag)	6 (ad)	6 (dg)	6 (dd)

Les mesures effectuées sur le prototype généré automatiquement par l'outil CPN/ Tagada, montrent qu'au delà de cinq machines, le gain n'est plus optimal. Ceci est principalement dû aux coûts de communication des rendez-vous distants entre processus. On en déduit alors qu'il est difficile de détecter sémantiquement le placement idéal d'un modèle. Il faut donc prendre en compte l'architecture globale du prototype généré en tentant d'affiner le placement et les coûts d'utilisation des divers gestionnaires du prototype.

5.3. Modélisation avec H-COSTAM

La Figure 53 présente une version H-COSTAM (sans la définition des types de données, pour des raisons de lisibilité) du modèle d'assemblage de voiture présenté à la Figure 50.

L'utilisation de la hiérarchie facilite la lecture du modèle et met en valeur les composants logiciels H-COSTAM, tels que l'utilisateur les conçoit.

Nous avons volontairement mis en exergue sur la Figure 53, le sous ensemble du modèle décrivant la construction des hayons de la voiture. Ainsi, au niveau Macro, le module *Hayons* contient deux processus H-COSTAM pour la construction du hayon intérieur (*Hayons_Int*) et extérieur (*Hayons_Ext*), qui se donnent rendez-vous pour produire la

pièce finale. Enfin, au niveau Micro, la description des états et des actions du processus *Hayons_Ext* est très proche de celle du réseau de Petri du modèle de la Figure 50. Enfin, on remarque que les Ressources déduites du modèle réseau de Petri, sont modélisées par des media passifs (ici des liens de communication FIFO).

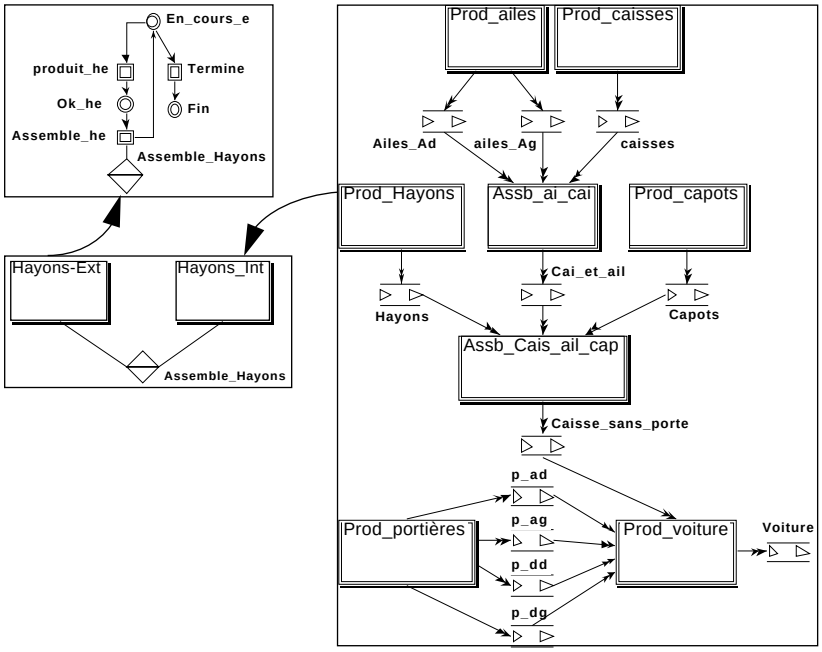


Figure 53 : Modèle d'assemblage d'une voiture en H-COSTAM

5.3.1. De H-COSTAM vers VODEL-D

Si on applique les règles de transformation définies dans le Tableau 33, on obtient le modèle de la Figure 54.

Il est intéressant de noter la pondération des DVGL (agglomérant plusieurs DVSL) liant les DVSC et indiquant qu'entre deux composants H-COSTAM, différents types de messages sont acheminés. Ainsi, entre *Prod_portières* et *Prod_voiture*, quatre liens de communication ont été définis (*p_ad, p_ag, p_dd, p_dg*), alors que dans le modèle VODEL-D, seul un lien a été défini, mais avec une pondération indiquant le nombre de liens (dans notre cas égal à 4).

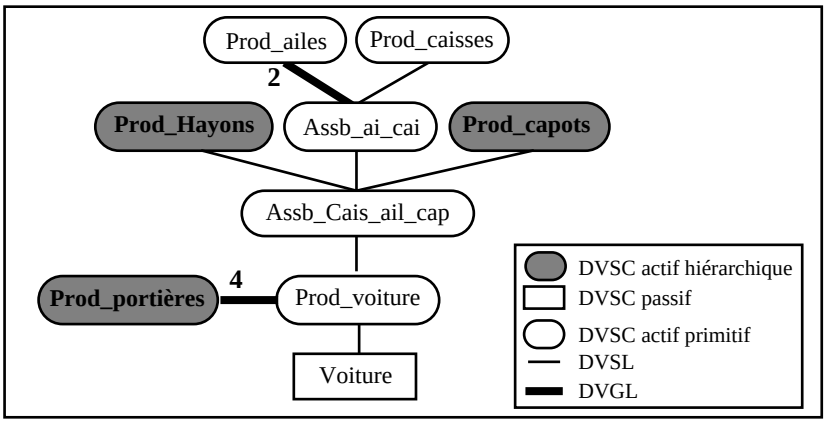


Figure 54 : Modèle déduit automatiquement de H-COSTAM

Notons enfin, que le lien FIFO allant de *Prod_voiture* à *Voiture* est transformé dans le modèle en DVSC passif.

5.3.2. Placement des composants logiciels au sens de l'équilibrage d'application

Si on prend en compte les pondérations entre DVSC, on en déduit le regroupement de :

- *Prod_portières* et *Prod_voiture* ;
- *Prod_ailes* et *Assb_ai_cai*.

Le DVSC passif *Voiture* est intégré dans *Prod_voiture*, car seul *Prod_voiture* est en relation avec *Voiture*.

Prod_Hayons et *Prod_Capots* sont d'un niveau de granularité satisfaisant, surtout si on prend en compte leurs synchronisations internes et le fait qu'ils ne sont pas des consommateurs (mais uniquement des producteurs).

Prod_caisses est intégré dans le DVSC contenant *Prod_ailes* et *Assb_ai_cai*, pour minimiser les coûts de synchronisations dans le composant *Assb_ai_cai*.

Enfin, le composant logiciels *Assb_Cais_ail_cap* dispose des liens communication avec tous les autres DVSC créés. Leur pondération étant équivalente, nous décidons de le rapprocher du DVSC consommateur (à savoir *Prod_portières, Prod_voiture, Voiture*).

La Figure 55 résume les choix de regroupement réalisé et le Tableau 36 les caractéristiques de chaque DVSC créé.

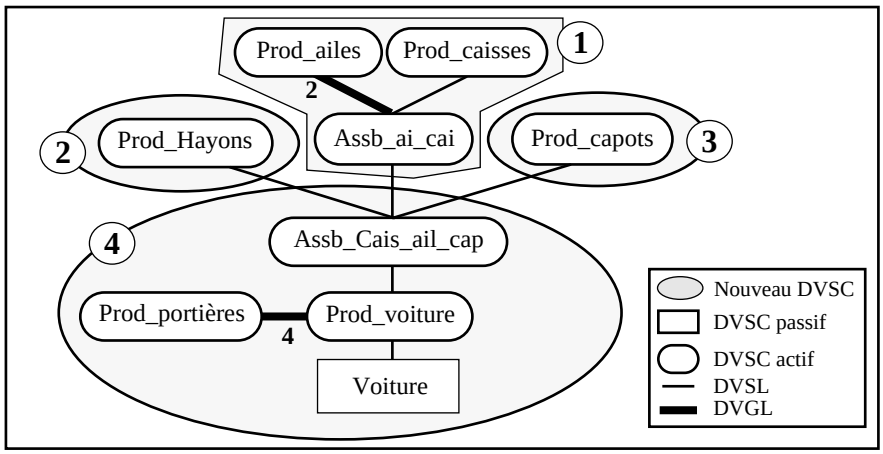


Figure 55 : Placement de composants logiciels issus d'un modèle H-COSTAM

Tableau 36: Les composants logiciels VODEL-D issus du calcul du placement

Nom	Contient	Type	Evolution	Hiérarchie (niveaux de hiérarchie)	Regroupement	Répartition
Prod_caisses_ailes (1)	Prod_ailes, Prod_caisses, Assb_ai_cai	Actif	libre	Hiérarchique (1)	Dissociable	Centralisé
Prod_hayons (2)	Prod_hayons	Actif	libre	Hiérarchique (2)	Indissociable	Centralisé
Prod_capots (3)	Prod_capots	Actif	libre	Hiérarchique (2)	Indissociable	Centralisé
Portiere_assemble (4)	Assb_Cais_ail_cap, Prod_portières, prod_voiture, Voiture	Actif	Fixe	Hiérarchique (2)	Indissociable	Centralisé

Le niveau de hiérarchie des DVSC présentés dans le Tableau 36 est de 2 (sauf pour le DVSC *Prod_caisses_ailes*, qui n'a qu'un niveau de hiérarchie). Ceci s'explique par le fait que ces DVSC contiennent des DVSC de niveau inférieur qui se synchronisent. Ce choix est directement issu de la volonté du concepteur. Parmi les quatre DVSC, seul le DVSC *Prod_caisses_ailes* est dissociable. En effet, il est possible de séparer *Prod_caisses* des autres DVSC inclus, si l'on désire augmenter le degré de parallélisme, tout en supportant le coût de communication avec *Assb_Cais_ail_cap*.

Un placement possible des DVSC en fonction du nombre de machines disponibles est présenté sur le Tableau 37.

Tableau 37: Placement des DVSC issus de H-COSTAM sur un nombre maximum de 5 machines

Nombre de machines	Machine 1	Machine 2	Machine 3	Machine 4	Machine 5
1	1, 2, 3, 4				
2	1, 2, 3	4			
3	1	2, 3	4		
4	1	2	3	4	
5	1 (Prod_caisses)	1 (Prod_ailes, Assb_ai_cai)	2	3	4

Le calcul du nombre de composants (ainsi que leur placement) à partir du modèle H-COSTAM conduit à une architecture logicielle plus proche du parallélisme réel. Le placement optimal calculé nécessite 5 machines, ce qui correspond aux mesures expérimentales.

5.4. Comparaison des deux approches

La différence majeure entre les deux approches est liée au rôle joué par l'utilisateur dans le processus de conception et de génération de l'application. Dans le premier cas, le résultat est soumis à la calculabilité de la décomposition du modèle réseau de Petri en machines à états communiquantes. Si ce calcul échoue, il faut modifier le modèle à la main car les procédures de transformations ne sont, dans le cas général, pas automatisables. Une fois les décompositions calculées, il reste à choisir celle qui sémantiquement correspond le mieux à l'approche qu'a le concepteur du problème. Dans le second cas, la conception et la structuration du modèle sont pilotées directement par l'utilisateur. Ainsi, grâce aux objets prédéfinis de H-COSTAM et à l'utilisation de la hiérarchie, le concepteur imprime sa vision de l'architecture et son découpage fonctionnel. Le placement est alors basé non plus sur un calcul structurel, mais sur une décomposition fabriquée de toute pièce par l'utilisateur.

Le second point à prendre en compte est lié à la sémantique de la communication entre deux entités logicielles. Lorsqu'un objet de génération processus utilise une ressource, cela équivaut en VODEL-D à un lien entre un DVSC actif (le processus) et un DVSC passif (la ressource). Notons néanmoins, que rien ne nous indique quelle est la sémantique de cette communication. On ne distingue donc pas un accès à une base de données, d'un envoi de message ou d'une écriture dans un tampon de données distant. Cela a des conséquences sur la précision des algorithmes de placement. Avec H-COSTAM, par contre, la sémantique des liens de communication est clairement définie. Un RPC, un rendez-vous ou une communication asynchrone ne sont pas représentés de la même manière. Cela facilite le calcul du placement par une meilleure compréhension du schéma et des coûts de communication. Par contre, la transformation de ces liens de communication en objets VODEL-D, n'est pas unique.

Ainsi par exemple, un lien de communication FIFO entre deux processus H-COSTAM est représentable de deux manières distinctes en VODEL - D (cf. Figure 56) :

1) par deux DVSC actifs reliés entre eux par un DVSL. On considère alors, en première heuristique, que la zone de stockage des données est située chez le consommateur.
2) par deux DVSC actifs reliés entre eux par l'intermédiaire d'un DVSC passif. L'uti-

lisation du DVSC passif est intéressante si l'on recherche un moyen de dupliquer les zones de stockage de données. Les deux DVSL correspondent alors à des envois de messages asynchrones ou communiquent en écrivant dans un fichier.

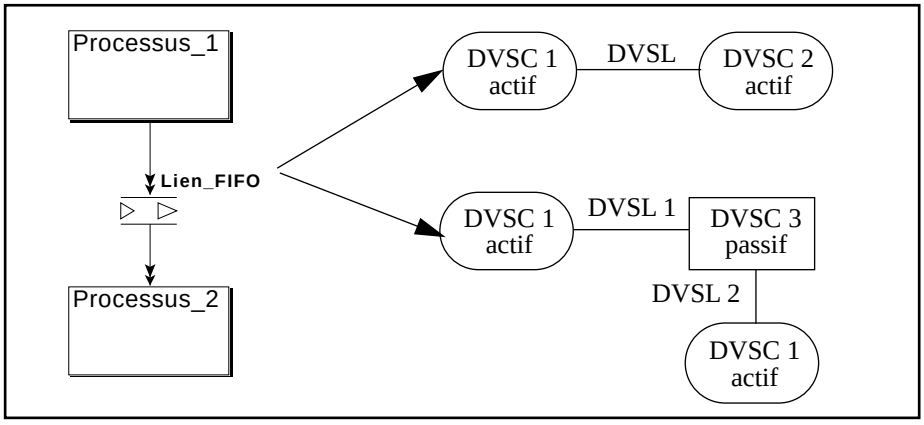


Figure 56 : Sémantique de communication entre processus H-COSTAM

En ce qui concerne l'aide au calcul du placement, dans un modèle réseau de Petri, il n'est pas possible de spécifier des liens d'attraction ou de répulsion entre composants logiciels, car on ne les connaît qu'après le calcul de la décomposition du modèle en machines à états. Par contre en H-COSTAM, comme le concepteur spécifie directement des composants logiciels hiérarchiques, il est possible de rajouter ce genre d'informations.

6. Conclusion

Nous avons mis en oeuvre l'équilibrage d'application dans un cadre de génération automatique d'applications réparties réalisée à partir de modèles formels ou opérationnels.

6.1. Génération d'applications à partir d'un modèle formel

La vérification d'invariants structurels sur un réseau de Petri [Kordon & al. 91b et Kordon 92] est utilisée pour identifier une décomposition d'un modèle en un ensemble de machines à états communiquantes. Cette décomposition donne naissance à des objets de génération (des processus, des ressources et des actions) [Kordon 92] utilisés pour produire automatiquement du code source.

Ces objets de génération forment une architecture logicielle de définition, qu'il est possible de décrire avec VODEL-D. Le passage des objets de génération aux DVSC et DVSL est réalisé lors du passage de l'architecture de définition à l'architecture d'exécution. Le prototype généré dispose alors de gestionnaires dont le rôle est de prendre en charge ces objets de génération correspondant à des composants logiciels.

La mise en oeuvre de la génération de code à partir de réseaux de Petri Biens Formés [Chiola & al. 91] a donné naissance à l'outil CPN/Tagada. Grâce à CPN/Tagada, nous avons pu tester l'exécution répartie des prototypes générés. **Nous en avons conclu que pour améliorer le placement, il est nécessaire de disposer d'informations de plus haut niveau que celles exprimables dans un réseau de Petri.**

De plus, la génération automatique de code à partir de réseaux de Petri est parfois limitée par des conflits sémantiques et structurels au niveau du modèle (lorsque la décomposition est impossible). La transformation du modèle, réalisée de manière automatique ou manuelle, est alors la seule solution possible pour résoudre ces conflits.

6.2. Génération d'applications à partir d'un modèle opérationnel

La génération d'applications (c'est à dire la génération du code du prototype et le calcul de son placement) à partir d'un modèle opérationnel semi-formel appelé H-COSTAM, a alors été étudiée. H-COSTAM a été conçu pour qu'il soit possible de passer d'un modèle H-COSTAM à un modèle réseau de Petri équivalent, via des transformations automatiques [Kordon & al. 95]. L'objectif étant également de capitaliser le travail précédemment réalisé dans la validation formelle des propriétés structurelle d'un modèle.

Pour réaliser le calcul du placement, nous transformons de manière automatique un modèle H-COSTAM en VODEL-D. Puis, à partir de la description de l'architecture logicielle et matérielle, nous calculons le placement des différents composants logiciels images du modèle H-COSTAM. Ce placement consiste, en fait, à paramétrer la réplication des différents gestionnaires du prototype et à leur indiquer les composants logiciels qu'ils gèrent. Du fait de la correspondance unique entre un modèle H-COSTAM et VODEL-D, la traçabilité est assurée et il est alors possible de remonter des informations après exécution directement jusqu'au modèle H-COSTAM.

Les différentes étapes de la méthode MEDEVER, dans le cadre de l'équilibrage d'application avec H-COSTAM, sont résumées sur la Figure 57. Les parties grisées sur la figure représentent les spécificités des étapes de la génération d'applications, par rapport à notre méthode générale MEDEVER. Elles correspondent à la phase de transformation du modèle H-COSTAM en VODEL-D, à la phase de génération de code et à la phase de production d'un fichier de configuration spécifique paramétrant le placement du prototype issu du générateur de code.

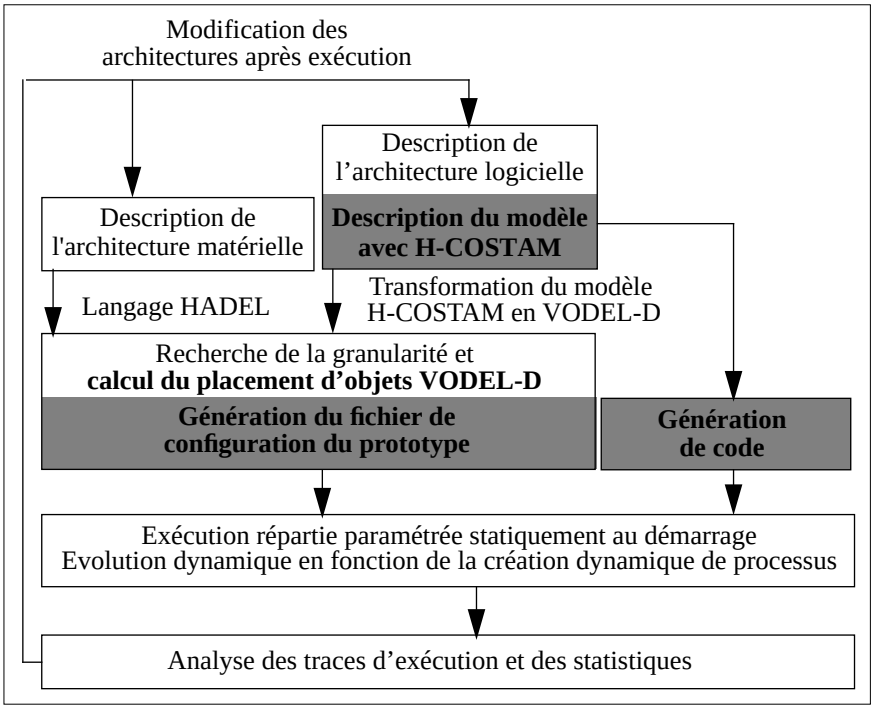


Figure 57 : La méthode MEDEVER adaptée à l'équilibrage d'application

6.3. Mise en oeuvre et perspectives

Les formalismes H-COSTAM et réseau de Petri sont tous deux intégrés dans l'environnement CPN-AMI 2.0 [Mars 96], qui implémente la méthode MARS (Méthode d'Analyse et de Réalisation de Systèmes). L'implémentation de H-Tagada, notre nouveau générateur de code à partir d'un modèle H-COSTAM, est actuellement en cours.

La recherche d'algorithmes de placement du code généré par H-Tagada est basée sur une représentation intermédiaire du problème. On modélise alors l'application via les graphes de DVSC et DVSL issus de VODEL-D et l'architecture matérielle par le graphe de H-Machines et H-liens issu de HADEL. Le placement peut être calculé par des algorithmes de partitionnement de graphe ou de correspondance d'un graphe de composants logiciels sur un graphe de machines. Ces algorithmes existent dans la littérature [Billionnet & al. 89 et Norman & al. 93], nos travaux futurs porteront donc sur l'intégration de tels algorithmes dans H-Tagada.

Enfin, H-COSTAM étant un modèle opérationnel dédié à la génération d'applications, **nous pensons qu'il serait judicieux d'inclure directement dans H-COSTAM certains attributs définis dans les DVSC de VODEL-D et qui sont utilisés pour le placement.** Ces attributs, cachés ou non à l'utilisateur, faciliteraient la prise en compte des contraintes architecturales du modèle et permettraient l'optimisation du code du prototype généré et le calcul direct (sans repasser par VODEL-D) du placement de ses composants.

7. Références bibliographiques

[ACT 97] ACT-Europe, «GLADE User Manual», <http://www.act-europe.fr/glade_guide.html>

[Asur 93] S.Asur & S.Hufnagel, «Taxonomy of Rapid-Prototyping Methods and Tools», Proceedings of the 4th "International Workshop on Rapid System Prototyping", Research Triangle Park Institute, N.Kanopoulos Eds., North Carolina, USA, pp. 43-56, June 1993.

[Attiogbé & al. 96] C. Attiogbé, H. Habrias & A. Vailly, «Exercices de styles», Institut de Recherche en Informatique de Nantes, Rapport de Recherche IRIN, RR-139, Octobre 1996.

[Barkaoui & al. 92] K. Barkaoui, C. Dubois & T. Demarche, «Validation statique des programmes multitâches ADA par l'analyse structurelle des réseaux de Petri», 2nde Conférence Maghrébine en Génie Logiciel et Intelligence Artificielle, Tunis, 1992.

[Bastide 95] R. Bastide, «Approaches in Unifying Petri Net and the Object-oriented Approach», Proceedings of the 1st Workshop on Object-oriented Programming and Models of Concurrency, Torino, Italy, 1995.

[Billionnet & al. 89] A. Billionnet, M.-C. Costa & A. Sutter, «Les problèmes de placement dans les systèmes distribués», Technique et Science Informatiques, Vol. 8 (4), pp. 307-337, 1989.

[Boehm 91] B.W. Boehm, «Software Risk Management : Principles and Practices», IEEE Software, pp. 32-41, January 1991.

[Boussant & al. 92] C. Boussant, E. Cavillon & F. Kordon, «Tepacap : transformation et repartition d'une application codée automatiquement en Ada sous Unix», Rapport MASI 92/2, Juin 1992.

[Bréant 90] F. Bréant, «Tapioca: Occam Rapid Prototyping from Petri Nets», Proceedings of the Vth Jerusalem Conference on Information Technology, Jerusalem, Israel, 1990.

[Bréant 91] F. Bréant, «Rapid Prototyping from Petri Net on a Loosely Coupled Parallel Architecture», Applications of Transputer, IOS Press, Vol. 3, 1991.

[Bréant & al. 92] F. Bréant & E. Paviot-Adet, «OCCAM Prototyping from Hierarchical Petri Nets», Proceedings of the International Conference Transputers'92, Senan, France, 1992.

[Bréant & al. 93] F. Bréant & J.F. Peyre, «A New Prototyping Method of Massively Parallel Applications Using Colored Petri Nets», Transputer Research and Applications, IO Press, Vancouver, Canada, Vol. 6, pp. 83-98, 1993.

[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.

[Briz & al. 94] J.L. Briz & J.-M. Colom, «Implementation of Weigthed Place/Transistion Systems Based on Linear enabling Functions», Advances in Petri Nets, Lecture Notes in Computer Science, No 815, pp. 99-118, 1994.

[Bruno & al. 86] G. Bruno & G. Marchetto, «Process Translatable Petri Nets for the Rapid Prototyping of Control Systems», IEEE Transactions on Software Engineering, Vol. 12(2), February 1986.

[Bruno & al. 95] G. Bruno, A. Castella, R. Agarwal & M.P. Pescarmona, «Cab: an Environment for Developping Concurrent Application», Proceedings of the 16th International Conference on Application and Theory of Petri Nets, Torino, Italy, 1995.

[Buchs & al. 92] D. Buchs, J. Flumet & P. Racloz, «Producing Prototypes from CO-OPN Specifications», 3rd International Workshop on Rapid System Prototyping, Research Triangle Park, North Carolina, USA, pp. 77-93, 1992.

[Buchs & al. 96] D. Buchs, P. Racloz, M. Buffo, J. Flumet & E. Urland, «Deriving Parallel Programs Using SANDS Tools», Transputer Communication Journal, Vol. 3 (1), pp. 22-32, 1996.

[Budde & al. 84] R. Budde, K. Kuhlenkamp, L. Mathiassen & H. Zûllighoven, «Approaches to prototyping», Springer Verlag Eds, Berlin, 1984.

[Cassigneul 91] V. Cassigneul, «SAO Presentation», Aerospatiale Technical Report, No 463.097/91, Toulouse, 1991.

[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.

[Choppy 94] C. Choppy, Rapport de recherche LRI No 891.

[Colom & al. 86] J.M. Colom, M. Silva & J.L. Villarroel, «On software implementation of Petri Nets and colored Petri Nets using high-level concurrent languages», 7th Workshop on Application and Theory of Petri Nets, pp. 207-220, 1986.

[Coriat 92] M. Coriat, «Placement d'un prototype Ada déduit d'un réseau de Petri sur une architecture distribuée : application au placement des G-Objets», Rapport de DEA, Laboratoire MASI, 1992.

[Diagne & al. 96a] A. Diagne & P. Estraillier, «Formal Specification and Design of Distributed Systems», In Proc. of the fisrt IFIP Int. Workshop on Formal Methods for Open Object-based Distributed Systems (FMOODS'96), France, Paris, Mars 1996.

[Diagne & al. 96b] 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.

[Diagne 97] A. Diagne, «Une approche Multi-Formalismes de Spécification de Systèmes Répartis : Transformation de Composants Modulaires en Réseaux de Petri», Thèse de doctorat de l'Université Pierre et Marie Curie, Laboratoire MASI, Paris, Mai 1997.

[Diot 96] J. Diot, «Génération de programmes parallèles en langage C à partir d'un modèle de machines à états communicantes», Rapport de DEA, Laboratoire MASI, Paris, 1996.

[Dolev & al. 94] D. Dolev, R. Strong & E. Wimmers, «Experiences with RAPID prototypes», Proceedings of the Vth IEEE International Workshop on Rapid System Prototyping, Grenoble, France, June 20-23, pp. 62-72, 1994.

[ECMA 93] ECMA, «A Reference Model for Frameworks of Stoftware Engineerings Environments», ECMA Report, No TR/55 (version 3), NIST Report Number, April 1993.

[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.

[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.

[Felder & al. 93] M. felder, C. Ghezzi et M. Pezzé, «High Level Timed Petri Nets as a Kernel for Executable Specification», Real Time Systems, Vol 5 (2/3), pp. 235-248, May 1993.

[Foughali & al. 94] K.Foughali & B.Folliot, «TOPICS-SE : a tool-based open platform for integration of control in software environments», Information and Software Technology, Vol 36 (7), 1994.

[Gaudel 91] M.C. Gaudel, «Algebraic Specifications», Chapter 22 in "Software Engineer's Reference Book", John Mc Dermid Eds., Butterworth, 1991.

[Genrich 87] H.J. Genrich, «Predicate/Transition Nets», in «Petri Nets : central model and their property», Lectures Notes in Computer Sciences, No 254, pp. 207-247, Springer Verlag, 1987.

[Hack 74] M. Hack, «Extended State-Machine Allocatable Nets (ESMA), an extension of free Choice Petri Net results», MIT, project MAC, Computation Structures Group, Memo 78-1, 1974.

[Haddad 91] S. Haddad, «A Reduction Theory for Coloured Net», K. Jensen & G. Rozenberg Eds., High Level Petri Net, Theory and Application, Springer Verlag, 1991.

[Hallmann 91] M. Hallmann, «A Process Model for Prototyping», Software Engineering & its Applications, Toulouse, 9-13 Décembre 1991.

[Hauschildt 87] D. Hauschildt, «A Petri Net Implementation», Fachbereich Informatik, Universität Hamburg, Hamburg, February 1987.

[Hulaas 97] G.J. Hulaas, «An Incremental Prototyping Methododlogy for Distributed Systems Based on Formal Specifications», PhD thesis, Ecole Polytechnique Fédérale de Lausanne, Avril 1997.

[Jensen 92] K. Jensen, «Coloured Petri Nets. Basic concepts, analysis method and practical use (vol 1)», EATC monographs on Theoretical Computer Science, Springer Verlag 1992.

[Kettani 95] N. Kettani, «Iterative Development as a New Solution for the Software Cost Problem», Proceedings of the 8th International Conference on Software Engineering and its Applications, Paris, France, pp. 741-748, November 1995.

[Kordon & al. 90] F. Kordon, P. Estraillier & R. Card, «Rapid Ada Prototyping: Principles and Example of a Complex Application», Proceedings of the 9th International IEEE Phoenix Conference on Computers and Communications", Phoenix, USA, pp. 453-460, 1990.

[Kordon & al. 91a] F. Kordon & P. Estraillier, «Complex Systems rapid Prototyping and Environment Abstraction», Proceedings of the 2nd International Workshop on Rapid System Prototyping" N.Kanopoulos Eds, IEEE comp Soc Press, Triangle Park Institute, pp. 34-46, June 1991.

[Kordon & al. 91b] F. Kordon & J.F. Peyre, «Process Decomposition for Rapid Prototyping of Parallel systems», 6th International Symposium on Computer and Information Science, Kener, Antalya, Turkie, October 1991.

[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 94a] F. Kordon, «Formal techniques based on nets, object orientation and reusability for rapid prototyping of complex systems», Proc. of the IFP-WG 10.3 Working Conference on Programming Environments for Massively Parallel Distributed Systems, Ascona, Switzerland, April 1994.

[Kordon & al. 94b] F. Kordon & W. El-Kaim, «CPN/TAGADA User Guide, in «The CPN-AMI Environment Version 1.3», Technical Report, Laboratoire MASI, Université Pierre et Marie Curie, Paris, France, 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.

[Lakos & al. 95a] C. Lakos & C. Keen, «An Open Software Engineering Environment Based on Object Petri Nets», Technical Report, Dept of Computer Science, University of Tasmania, Australia, May 1995.

[Lakos 95b] C. Lakos, «From Colored Petri Nets to Object Petri Nets», .Proceedings of the 16th International Conference on Application and Theory of Petri Nets, Torino, Italy, LNCS No 935, pp. 278-297, June 1995.

[Leòn & al. 93] G. Leòn, N. Zakhama, J.C. Duenas & al., «The IPTES Environment: Support for Incremental Heterogeneous and Distributed Prototyping», Real Time Systems, Vol 5 (2/3), pp. 153-171, May 1993.

[Luqi 89] Luqi, «Software Evolution Through Rapid Prototyping», IEEE Software, Vol. 22 (5), pp. 13-25, May 1989.

[Luqi 92] Luqi, «Computer Aided System Prototyping», 3rd International Workshop on Rapid System Prototyping, Triangle Park Institute, pp. 50-57, June 1992.

[Macao 97] <http://www-src.lip6.fr/logiciels/macao>.

[MARS 96] MARS-Team, «The CPN-AMI Version 2.0», Laboratoire MASI, Institut Blaise Pascal, Université Pierre & Marie Curie, 4 place Jussieu, 75252Paris Cedex 05.

[Meinadier 94] J.P. Meinadier, «Du génie logiciel au génie système (vers l'atelier d'ingénierie système en informatique)», Résultat de l'enquête 1994 sur l'état du génie logiciel en France, AFCET, groupe de travail génie Logiciel, Octobre 1994.

[Murata & al. 86] T. Murata, N. Komoda & K. Matsumoto, «A Petri Nets based Factory Automation controller for flexible and maintainable control specification», IEEE Transactions on Industrial Electronics, Vol. IE-33 (1), February 1986.

[Murata 90] T. Murata, «Petri nets : Properties, Analysis and Applications», Proceedings of the IEEE, Vol 6 (1), pp 39-50, January 1990.

[Murphy & al. 89] S.C. Murphy, P. Gunningberg & J.P.J. Kelly, «Implementing protocols with Multiple Specifications : Experiences with Estelle, LOTOS and SDL», 9th IFIP WG 6.1 International Symposium on Protocol Specifications, Testing and Verification, Enschede, The Netherlands, June 1989.

[Nelson & al. 83] R.A. Nelson, L. M. Haibt & P. B. Sheridan, «Casting Petri Nets into Programs», IEEE Transaction on Software Engineering, Vol. SE-9, September 1983.

[Norman & al. 93] M.G. Norman & P. Thanisch, «Models of Machines and Computation for Mapping in Multicomputers», ACM Computing Surveys, Vol. 25(3), pp. 263-302, 1993.

[Nuttal 94] M. Nuttal, «A Brief Survey of Systems Providing Process or Object Migration Facility», Operating Systems Revue, Vol. 28 (4), pp. 64-80, October 1994.

[Paludetto 91] M. Paludetto, «Sur la commande de procédés industriels : une méthodologie basée objet et réseaux de Petri», PhD Thesis, Université paul Sabatier, Toulouse, France, Dec. 1991.

[Peyre 93] J-F.Peyre, «Résolution paramétrée de systèmes linéaires ; applications au calcul d'invariants positifs dans les réseaux colorés, à la validation formelle et à la génération de code», PhD thesis, Université P.&M.Curie, 4 place Jussieu, 75252 Paris Cedex 05, France, March 1993.

[Pezzè & al. 92] M.Pezzè & C.Ghezzi, «Cabernet : a customizable Environment for the Specification and Analysis of Real-Time Systems», Report of the Dipartimento di Elettronica e dell'Informazione Politecnico di Milano, Piazza Leonardo da Vinci 32, 20133 Milano, September 1992.

[Reisig 91] W. Reisig, Petri Nets and Algebraic Specification, Theoretical Computer Science, No 80, pp. 1-34, 1991.

[Shapiro 91] R.M. Shapiro, «Validation of a VLSI chip using hierarchical coloured Petri Nets», Microelectronics and Reliability, Special issue on Petri Nets, Pergamon Press, 1991.

[Sibertin-Blanc 94] C. Sibertin-Blanc, «Cooperative Nets», Proceedings of the 15th International Conference on Application and Theory of Petri Nets, Zaragoza, Spain, 1994.

[Silva & al. 82] M. Silva & S. Velilla, «Programmable logic controllers and Petri nets», International Symposium IFAC-IFIP on Software Computer Control, Madrid, September 1982.

[Stevens 90] W.R. Stevens, «Unix Network programming», Prentice Hall Eds., 1990.

[Taubner 88] D. Taubner, «On the implementation of Petri Nets», Advances in Petri Nets, Lecture Notes in Computer Science, No 340, pp. 318-340, 1988.

[Thuriot 85] E. Thuriot, «Le synchroniseur coloré : une approche pour la mise en oeuvre des systèmes multitâches», Thèse de docteur-ingénieur, Université Paul Sabatier, Toulouse, Février 1985.

[Valette & al. 83] R. Valette, M. Courvoisier, J.M. Bigou & J. Alburkerque, "A Petri net based programmable logic controller", 1st International Conference on Computer Application in Production and Engineering, Amsterdam, April 1983.

[Valk 95] R. Valk, «Petri nets as Dynamical Objects», Proceedings of the 1st Workshop on Object-oriented Programming and Models of Concurrency, Torino, Italy, 1995.

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

[Vonk 92] R. Vonk, «Prototypage : l'utilisation efficace de la technologie CASE», Masson & Prentice Hall Eds., Paris, 1992.

[Zakhama 96] N. Zakhama, «Le prototypage hétérogène formel de systèmes distribués, une perspective pour le développement continu», PhD thesis, Université Pierre et Marue Curie, Paris, Juin 1996.