PROTOTYPAGE DE SYSTÈMES PARALLÈLES À PARTIR DE RÉSEAUX DE PETRI COLORÉS

APPLICATION AU LANGAGE ADA DANS UN ENVIRONNEMENT CENTRALISÉ OU RÉPARTI

Avertissement:
dans ce tirage, les indexs et table des matières sont hélas incorrects
(merci la compatibilité Microsoft;-)

Thèse de l'université Pierre & Marie Curie (Paris VI)

Spécialité : Informatique

Fabrice KORDON

Soutenue le 11 Mai 1992

devant le jury composé de:

M. **Jean-François PERROT**	*Président*
M. **Robert VALETTE**	*Rapporteur*
M. **Jean-Claude DERNIAME**	*Rapporteur*
M. **Claude GIRAULT**	
M. **Christian CARREZ**	
M. **Pascal ESTRAILLIER**	*Directeur*
M. **Bernard COUSIN**	

“Hardware is... Software will”

L’informaticien anonyme

A Alix
sans qui rien n'est...

A Monique, Claude, Margerite, Gaston, Hélène, Wolfgang
sans qui rien n'aurait été...

Et bien sûr, tous ceux sans qui rien ne serait réellement.

Je tiens à remercier vivement :

- à *Monsieur Claude Girault, Professeur à l´université Pierre et Marie Curie, pour m´avoir accueilli dans son équipe et soutenu dans les travaux dont le présent mémoire est l´aboutissement;*
- à *Monsieur Pascal Estraillier, Maître de Conférences à l´université Pierre et Marie Curie, qui a dirigé mes recherches et su si astucieusement détruire, en laissant systématiquement le mode d´emploi permettant une reconstruction ô combien plus résistante que la précédente;*
- à *Monsieur Jean-François Perrot, Professeur à l´université Pierre et Marie Curie, pour avoir accepté de présider ce jury;*
- à *Monsieur Robert Valette, Professeur à l´université Paul Sabatier, Toulouse, pour s´être intéréssé à ce travail et avoir accepté d´en être rapporteur;*
- à *Monsieur Jean-Claude Derniame, Professeur à l´université de Nancy, pour avoir accepté de juger le présent document;*
- à *Monsieur Christian Carrez, Professeur au Conservatoire National des Arts et Métiers, pour l´honneur qu´il me fait de participer à ce jury;*
- à *Monsieur Bernard Cousin, Maître de Conférences à l´université de Bordeau,x pour avoir été l´un des initiateurs de ce projet;*

- à *Christophe Boussant, Eric Cavillon, José Cheng, Michel Coriat, William El Kaïm, Pierre-Guillaume Raverdy, Pierre Sens pour leur aide précieuse dans la réalisation de certaines des idées développées dans ce document, avec une mention particulière pour l´impitoyable «passe syntaxique» effectuée par C. Boussant sur le présent mémoiez;*

- à *L´ensemble du personnel des laboratoires de l´Institut Blaise Pascal, et en particulier ceux de l´équipe Modélisation, Analyse et Réalisation de Systèmes du laboratoire MASI, pour le soutien, aussi cordial qu´efficace qu´ils ont su me procurer;*
- à *Le logiciel MACAO et son auteur, Jean-Luc Mounier, pour les dessins de graphes - réseaux de Petri et autres - contenu dans ce mémoire;*
- à *Claude Timsit, Martial Schmitt, Michel Angueloff, et les membres de l´équipe IOS, qui, les premiers, ont su me montrer que, parfois, «software can be» en me donnant le goût du métier que j´exerce aujourd´hui.*

Table des matières

INTRODUCTION GÉNÉRALE **1**

Objectifs....3

Plan de la thèse....6

Partie I
Le prototypage

Introduction à la partie I 11

CHAPITRE I : Prototypage de systèmes parallèles **15**

1. Introduction 17

1.1. Objectifs....18
1.1.1. Méthodologie générale....19
1.1.2. Mise en œuvre....20
1.2. Spécifier un système....22
1.3. Réaliser le système....24
1.4. Le prototypage....25
1.4.1. L'approche incrémentale....25
1.4.2. Le maquettage....26
1.4.3. L'approche par raffinements....27
1.5. Prototypage et méthodes de spécification....27

2. Quelques outils de prototypage à partir de formalismes autres que les réseaux de Petri 30

2.1. S.A.O....30
2.1.1. Les planches S.A.O.....30
2.1.2. L'outil S.A.O....31
2.1.3. Etude critique....32
2.2. PROTO....33
2.2.1. Les diagrammes PROTO....33
2.2.2. Construction et interprétation d'un modèle....34
2.2.3. Définition d'une architecture matérielle....35
2.2.4. Etude d'une application répartie sur une architecture multi-processeurs....36
2.2.5. Etude critique....36
2.3. Conclusion....37

3. Le prototypage à partir de réseaux de Petri 39

3.1. Les réseaux de Petri....39
3.1.1. Les réseaux de Petri Colorés....39
3.1.2. Les réseaux de Petri Bien Formés....41
3.2. Sémantique des réseaux de Petri....45
3.2.1. Règle de franchissement des transitions....45
3.2.2. La gestion des interblocages....45
3.2.3. Le degré de parallélisme....46
3.3. Différentes exécutions d'un réseau de Petri....47
3.3.1. Un exemple de modèle....48
3.3.2. Exécution centralisée....49
3.3.3. Exécution complètement décentralisée....50
3.4. Conclusion....52

4. Degré de parallélisme d'un système modélisé en réseau de Petri 54

4.1. Les travaux menés à l'université de Saragosse....55
4.2. Les travaux menés à l'université de Toulouse....57
4.3. Les versions d'étude du générateur de code Ada....58

4.4. Conclusion....61

5. Conclusion 62

5.1. L'apport des réseaux de Petri....62
5.2. Automatisation du prototypage 62
5.3. Du prototypage à la génération de code optimisé 63

CHAPITRE II : Interprétation sémantique d'un réseau de Petri 65

1. Introduction 67

2. Les G-objets 69

2.1. G-objet de type processus 69
2.1.1. Compteur ordinal d'un Processus....69
2.1.2. Mot d'état d'un Processus 70
2.1.3. Processus instancié 71
2.2. G-objet de type Actions....72
2.3. G-objet de type Etats_Processus 73
2.4. G-objet de type Ressources....74
2.5. Semi-flots et G-objets 75
2.5.1. Semi-flots d'un réseau de Petri Ordinaire....76
2.5.2. Le modèle structurel....77
2.5.3. Vers une interprétation des semi-flots....79
2.6. Synthèse....80

3. Algorithme de décomposition en processus 81

3.1. L'algorithme....81
3.1.1. La classe des modèles "prototypables"....81
3.1.2. Description de l'algorithme....84
3.2. Transformation d'un modèle en vue de le rendre "prototypable"....87
3.2.1. Processus instanciés dynamiques....88
3.2.2. Processus implicites....89
3.3. Vers une extension de l'algorithme de décomposition en processus....92
3.3.1. Interprétation de l'échec des propriétés 1 et 2....92
3.3.2. Vers une interprétation de l'échec de la propriété 3....97
3.3.3. Nouvelle classification des Actions 98

4. Synthèse 100

CHAPITRE III : Méthodologie de Prototypage 101

1. Introduction 103

2. Notion de composant externe 105

2.1. Abstraction d'un composant externe 105
2.1.1. Formalisation d'un composant externe....107
2.1.2. Description interne d'un composant externe 107
2.2. Les composants externes dans un modèle....110
2.3. La notion de service 111
2.4. Composants externes réversibles et non-réversibles 112

3. Les étapes du prototypage 115

3.1. Décomposition d'un modèle 116
3.2. Production du prototype....118

4. Conclusion 122

CHAPITRE IV : Structure du prototype généré 125

1. Introduction 127

1.1. Données en entrée de la phase de génération....128

1.2. Le prototype obtenu....129
1.3. Les différentes composantes du prototype....130

2. Les modules-processus 134

2.1. Problèmes liés à l'activation d'une Action....135
2.1.1. Evaluation d'une précondition....135
2.1.2. Réalisation du marquage postcondition....138
2.2. Etat_Processus....139
2.3. Actions....140
2.3.1. Actions simples....140
2.3.2. Actions simples gardées....140
2.3.3. Actions synchronisées....141
2.4. Comportement d'un Processus....141
2.5. Services requis....143
2.5.1. Services requis aux modules de service....144
2.5.2. Services requis au module de contrôle....149
2.5.3. Services requis au module des traitements....149
2.6. Services offerts....149
2.6.1. Services offerts aux modules de service....150
2.6.2. Services offerts au module de contrôle....150
2.7. Synthèse....151
2.7.1. Les interfaces du module....151
2.7.2. Génération du comportement....152

3. Le module de gestion des Ressources 154

3.1. Gestion des interfaces avec l'environnement....155
3.1.1. Production dans une Ressource externe en entrée....155
3.1.2. Consommation dans une Ressource externe liée à un composant externe non-réversible....156
3.1.3. Consommation dans une Ressource externe liée à un composant externe réversible....157
3.1.4. Traitement des événements positifs....158
3.2. Services offerts....160
3.2.1. Evaluation d'une précondition ressource....160
3.2.2. Production d'une postcondition ressource....163
3.2.3. La notion de condition....163
3.2.4. Initialisation et terminaison....165
3.3. Services requis....165
3.3.1. Services requis au module de contrôle....166
3.3.2. Services requis aux modules externes....166
3.3.3. Services requis aux modules clients....166
3.4. Synthèse des interfaces....167
3.5. Architectures possibles du module....168
3.5.1. Un seul serveur pour l'ensemble des Ressources....168
3.5.2. Un serveur par Ressource....169
3.5.3. Un serveur pour plusieurs Ressources....170

4. Le module de gestion des Actions synchronisées 172

4.1. Problèmes liés à l'activation d'une Action synchronisée....172
4.2. Services offerts....176
4.2.1. Services offerts aux modules-processus....176
4.2.2. Services offerts au module de contrôle....177
4.3. Services requis....177
4.3.1. Services requis aux modules-processus....177
4.3.2. Services requis au module de gestion des Ressources....178
4.3.3. Services requis au module des traitements....178
4.3.4. Services requis au module de contrôle....178
4.4. Synthèse des interfaces....178
4.5. Architecture du module....179

5. Le module de contrôle 182

5.1. Services offerts....182
5.1.1. Initialisation....183
5.1.2. Terminaison....184
5.2. Services requis....184

5.3. Synthèse des interfaces....185
5.4. Architectures possibles pour le module de contrôle....186
5.4.1. Contrôle effectué par un seul gardien....186
5.4.2. Contrôle effectué par plusieurs gardiens....187
6. Conclusion 189

Partie II
Application au langage Ada

Introduction à la partie II 193

CHAPITRE V : Caractéristiques du prototype Ada 197
1. Introduction 199
2. Modules fonctionnels et paquetages Ada 201
2.1. Modules internes....201
2.2. Les modules externes et le module de traitement....202
3. Les unités propres au prototype Ada 204
3.1. Evaluation du prototype....204
3.1.1. Rappel sur les exceptions en Ada....205
3.1.2. Trace du fonctionnement du prototype....206
3.1.3. Génération de la pile des appels....206
3.1.4. Le mode DEBUG....207
3.2. Le mécanisme de communication client-serveur....207
3.2.1. Motivation....208
3.2.2. Description de l'unité....209
3.2.3. Mise en œuvre....209
4. Les unités de "bas niveau" 212
4.1. Informations générales sur le prototype....212
4.2. La gestion des marques....213
4.2.1. Définition des classes de couleurs....213
4.2.2. Définition des domaines de couleurs....214
4.2.3. Le type marquage....215
5. Conclusion 216

CHAPITRE VI : le prototype centralisé 219
1. Introduction 221
2. Réalisation du module de gestion des Ressources 222
2.1. Description d'une condition....222
2.2. Description du résultat d'une précondition....224
2.3. Réalisation des requêtes....224
2.3.1. Description des Ressources....224
2.3.2. Traitement des requêtes....225
3. Réalisation des modules-processus 229
4. Réalisation du module de gestion des Actions synchronisées 232
5. Réalisation du module de contrôle 234
6. Conclusion 235

CHAPITRE VII : Le prototype réparti 237
1. Introduction 239

2. Les services de répartition 242
2.1. Choix du niveau de répartition 242
2.2. Traitement des unités réparties 243
2.3. Mise en œuvre de la répartition....245
2.4. Principes de conception....247
2.5. Définition d'une boîte à outils de répartition....252
3. Réalisation d'une application répartie 254
3.1. Principes de réalisation....254
3.1.1. Réception d'un message 254
3.1.2. Le contexte de l'application répartie....255
3.2. Architecture centralisée 256
3.2.1. Emission et routage des messages....256
3.2.2. Initialisation et terminaison 257
3.2.3. Adaptation d'une application déjà existante....260
3.2.4. Avantages et inconvénients 261
3.3. Architecture décentralisée....261
3.3.1. Emission et routage des messages....261
3.3.2. Initialisation et terminaison 262
3.3.3. Adaptation d'une application déjà existante....264
3.3.4. Avantages et inconvénients 265
4. Application au prototypage 266
4.1. Choix stratégiques 266
4.2. Première approche du prototype réparti 267
4.3. Seconde approche du prototype réparti....270
4.3.1. Le mode écran des TTR 270
4.3.2. La gestion délocalisée des Ressources....272
4.3.3. Services offerts par le nouveau gestionnaire des Ressources 281
5. Observations sur le placement 282
6. Expérimentation 287
6.1. Le modèle à prototyper....287
6.2. Les mesures....290
6.3. Quelques remarques 292
7. Conclusion 294

CHAPITRE VIII : L'outil de prototypage 297

1 Introduction 299
2 L'environnement logiciel AMI 301
2.1 La plate-forme de dialogue 301
2.2 L'interface utilisateur Macao 303
2.2.1 Le mode autonome....303
2.2.2 Le mode connecté 304
2.3 Les outils 305
2.4 Le formalisme de description utilisé....306
3 Le générateur de code 308
3.1 Architecture de l'outil 308
3.2 Hypothèses 309
3.3 Les phases d'Identification et de Caractérisation 312
3.3.1 Architecture....312
3.3.2 Identification des composants externes....313
3.3.3 Décomposition du modèle 317
3.4 La phase de Placement....317
3.5 La phase de Génération 318

3.6 La gestion des composants externes....319
3.6.1 Organisation des différentes bibliothèques....319
3.6.2 Organisation d'une bibliothèque....320
3.6.3 Création d'un composant....321

4 Conclusion 324

CONCLUSION GÉNÉRALE 325

Résumé....327
Apport méthodologique....327
Apport Formel....329
Apport Génie Logiciel....329
Apport technique....330
Perspectives....331

RÉFÉRENCES BIBLIOGRAPHIQUES 333

Annexes & Appendices

ANNEXE A : Les algorithmes types du prototype 347

Modèle de tâche déduit d'un Processus....349
Etat_Processus alternatif....349
Action simple....350
Action simple gardée....350
Action synchronisée....351
Structure d'un serveur (Ressources ou synchronisation)....352
Algorithme d'un gardien....353
Mise à jour d'une estampille locale à un serveur de Ressources....354
Production d'un marquage postcondition....355
Annulation d'une demande d'évaluation....356
Réveil des préconditions en attente d'un événement positif....357
Réception d'une demande de réactivation....358
Tenter l'envoi d'un jeton....359
Réception d'une demande de jeton....359
Réception d'un jeton R....359
Réception d'un jeton T....360
Evaluateur d'une précondition....360
Evaluation d'une précondition....361

ANNEXE B : Extraits du code généré pour un exemple 363

1. Le modèle....365
2. Exécution du prototype centralisé associé....367
3. Extraits du code généré....368

Code 1 : spécification de l'unité de gestion des traces....368
Code 2 : spécification de l'unité de gestion de la pile des appels de sous-programmes....369
Code 3 : exemple de fonction avec et sans mode DEBUG....369
Code 4 : spécification du paquetage de gestion des communications client-serveur....370
Code 5 : spécification de l'unité de description des données générales du prototype....371
Code 6 : spécification du paquetage de description des classes de couleurs....372
Code 7 : description des domaines de couleurs....373
Code 8 : spécification de l'unité décrivant le marquage du modèle....374
Code 9 : spécification de l'unité décrivant une condition....376
Code 10 : spécification de l'unité de gestion du résultat d'une consommation....380
Code 11 : description des ressources du modèle....382
Code 12 : spécification du gestionnaire des Ressources....383

Code 13 : spécification de l'unité chargée de l'évaluation des requêtes sur les Ressources....384
Code 14 : spécification de l'unité décrivant les opérations élémentaires sur les Ressources....385
Code 15 : spécification de l'unité permettant la manipulation du mot d'état de P1....385
Code 16 : spécification du paquetage décrivant le comportement de P1 386
Code 17 : modèle de tâche associé à P1 386
Code 18 : spécification de l'unité décrivant les Etats_Processus alternatifs de P1....388
Code 19 : spécification de l'unité décrivant les Actions de P1 389
Code 20 : code associé à l'Etat_Processus P1a....389
Code 21 : code associé à la transition Tcons1....390
Code 22 : spécification du paquetage réalisant l'Action synchronisée Tsync....392
Code 23 : spécification du module de contrôle....392
Code 24 : le programme principal....393

ANNEXE C : La description LANBADA 395

1. Introduction....397
2. Définitions....397
3. Description de l'ensemble du réseau de Petri 397
4. Description d'une classe de couleurs....398
5. Description d'un domaine de couleurs 399
6. Description des fonctions non prédéfinies du modèle....400
7. Description des ressources du modèle 401
8. Description d'un processus du modèle....403

APPENDICE D : Index des Définitions 409

APPENDICE E : Index des Figures 413

APPENDICE F : Index des Références 421

+

		Introduction générale
Chapitre I	:	*Prototypage de systèmes parallèles*
Chapitre II	:	*Interprétation sémantique d'un réseau de Petri*
Chapitre III	:	*Méthodologie de prototypage*
Chapitre IV	:	*Structure du prototype généré*
Chapitre V	:	*Caractéristiques du prototype Ada*
Chapitre VI	:	*Le prototype centralisé*
Chapitre VII	:	*Le prototype réparti*
Chapitre VIII	:	*CPN/TAGADA : L'outil de prototypage*
		Conclusion générale
		Bibliographie
		Annexes & Appendices

Introduction générale

Objectifs 3
Plan de la thèse.....6

OBJECTIFS

La conception de systèmes complexes implique le choix d'un ensemble de formalismes adaptés à la fois à l'expression de ces systèmes et à la maîtrise de leur élaboration. Le processus de modélisation intègre alors, sous forme d'optimisations progressives, l'ensemble des besoins identifiés par le cahier des charges. La plupart des méthodes de conception adoptent une approche à plusieurs niveaux d'abstraction qui vont d'une perception conceptuelle (fonctionnelle) jusqu'à la définition détaillée de l'implantation physique. Cette démarche permet de hiérarchiser les choix de conception et d'éviter, dans la mesure du possible, que des décisions du niveau conceptuel ne soient tributaires des choix préalables relevant de l'implantation physique. Par ailleurs, la réalisation d'un système est une opération coûteuse, longue et source d'erreurs et de déviations par rapport à la spécification initiale.

Dans ce cadre, nos recherches ont porté sur la définition et la mise en œuvre d'outils de génération automatique de prototypes exécutables destinés aux ingénieurs de conception. L'objectif visé est de leur permettre, non seulement de spécifier et de vérifier la correction de leur spécification, mais aussi d'automatiser la production d'un prototype conforme à leurs description. Une telle automatisation permet d'éviter la dérive entre la spécification et la réalisation d'un système.

En utilisant une approche délibérément formelle (basée sur les réseaux de Petri Colorés), on vise à renforcer les aspects liés à la vérification dans le cycle de vie, de conception et de réalisation de systèmes complexes. Il s'agit d'élaborer des méthodes d'analyse et de traduction de spécifications en déchargeant autant qu'il est possible le concepteur de la gestion explicite du modèle formel.

Plus précisément, notre objectif est d'effectuer l'analyse et la traduction d'un réseau de Petri afin de générer un prototype exécutable. Ce prototype regroupe les programmes sources composant le *squelette* du système. Ce squelette intègre le contrôle du comportement du système modélisé et invoque des traitements unitaires associés à la spécification. Certaines parties (traitements unitaires associés à l'application, interfaces avec le monde extérieur) ne peuvent être complètement définies à l'aide de la seule spécification formelle :

- Les traitements unitaires associés doivent être définis par le concepteur du modèle. Ils sont intégrés au prototype sous la forme d'appels de

procédures réalisés conformément aux contraintes de précédence définies dans le réseau de Petri.

- Afin de permettre une validation comportementale, il est nécessaire de relier la spécification d'un système à une *abstraction* du monde extérieur. De cette abstraction, nous pouvons déduire les interfaces du squelette avec l'environnement. Cependant, pour conserver certaines propriétés, il faut formaliser les relations entre la spécification du système et l'abstraction de l'environnement. Nous introduisons dans ce but la notion de *composant externe*. Naturellement, la génération ne porte que sur la spécification du système et pas sur son environnement.

Cette démarche présente les avantages suivants :

- le modèle est une spécification formelle regroupant l'ensemble des contraintes exprimées par le concepteur;
- le prototypage intervient après la validation du modèle, donc après élimination des erreurs de conception détectées par l'analyse;
- la détermination des processus, l'identification des ressources, la gestion des communications s'appuient sur les résultats de l'analyse structurelle et comportementale du modèle;
- le squelette permet de synthétiser clairement les interactions (synchronisation, échange des données) entre les activités concurrentes de l'application;
- les traitements unitaires associés (qui n'ont aucune interaction entre eux) constituent des éléments trivialement réutilisables.

Pour la spécification des applications, nous utilisons une catégorie de *réseaux de Petri Colorés* [Jensen 81] : les *réseaux de Petri bien formés* [Chiola 90, Dutheillet 91]. Ils correspondent simplement à une forme syntaxiquement contrainte des réseaux colorés.

Les réseaux de Petri ont démontré leur adaptation à l'analyse des systèmes parallèles [Ayache 85, Girault 87, Tu 90, Chehaibar 90, Vinci 90, Moitessier 91, Shapiro 91]. De plus, on observe actuellement une augmentation importante du nombre de résultats théoriques [Couvreur 90a, Couvreur 90b, Hubert 90, Dutheillet 91, Petrucci 91, Chehaibar 91, Haddad 91] permettant la démonstration de propriétés, tant structurelles que comportementales.

Nos travaux introduisent une méthode pour déterminer le processus de génération et pour établir une manière adéquate de modéliser. Dans ces conditions, une analyse particulière d'un modèle réseau de Petri suffit généralement pour retrouver la sémantique de l'application. Ce type d'analyse repose sur la théorie du calcul des flots qui met en évidence, sans développer le graphe des marquages accessibles, des combinaisons linéaires particulières permettant d'identifier les processus modélisés.

Le processus de prototypage que nous envisageons s'inscrit dans une méthodologie générale [Estraillier 91] donnée en Figure a. A partir de la description informelle de la solution proposée par le concepteur, une première étape permet d'obtenir une spécification semi-formelle du système montrant son organisation et ses interfaces [Kurtz 89, Brinkkemper 90, Brunet 91]. La

dérivation de ces spécifications génère ensuite un modèle formel du système incluant une représentation de son comportement dynamique sur laquelle sont appliquées des techniques formelles de validation qualitative et d'évaluation quantitative [Pomello 90, Van Glabbeck 90]. Finalement, un prototype peut être généré pour montrer au concepteur que les propriétés attendues du système sont vérifiées.

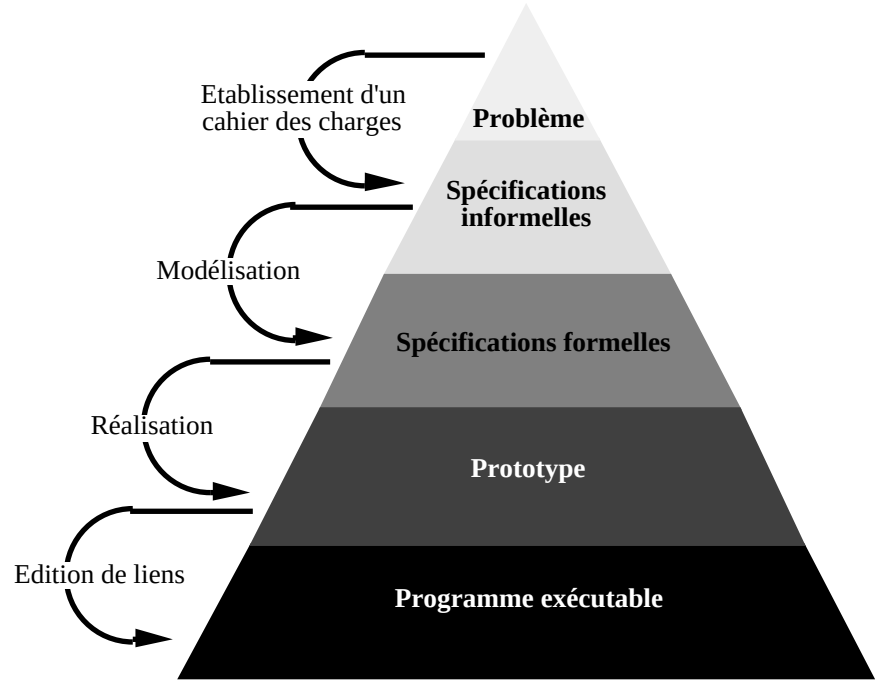


Figure a : Les différentes étapes de réalisation d'un système.

Pour l'étape de prototypage, nous avons étendu la théorie et développé les techniques assurant un niveau de vérification plus élevé. La difficulté vient alors de mettre en œuvre de manière coordonnée des mécanismes portant sur le contrôle des applications et sur le traitement des données tout en garantissant la cohérence vis-à-vis de la spécification.

Nous avons appliqué notre méthode de prototypage au langage Ada. Ce langage convient particulièrement, non seulement parce qu'il est procédural et intrinsèquement multi-tâches, mais aussi parce que sa structuration nous permet aisément de réaliser les fonctionnalités associées aux objets issus du modèle de spécification. En particulier, la notion d'*unité de programme* permet de diviser le prototype généré à partir de la décomposition du modèle en fonctionnalités types issues de son interprétation. Le mécanisme d'exception permet également d'élaborer des "sondes" fournissant des informations précieuses dans la phase de mise au point du prototype.

L'essor de la théorie et l'accroissement de ses champs d'application ont conduit à la mise en œuvre d'environnements logiciels permettant une modélisation aisée ainsi que la validation et une évaluation des modèles obtenus [Bernard 90, Chiola 91]. Il est alors possible de compléter de tels ateliers en y intégrant des outils d'implémentation et ainsi contribuer à la démonstration de l'apport des réseaux de Petri dans la conception d'applications complexes et parallèles.

Ainsi, nous avons concrétisé nos travaux dans la réalisation d'un outil de génération de code. Cet outil est intégré dans l'atelier de spécifications AMI. Il permet de générer du code Ada dans un environnement mono-processeur.

Notre contribution réside dans l'introduction de quatre idées novatrices :

- l'intégration progressive des contraintes liées à l'application, à son environnement et au langage de programmation;
- la formalisation de la relation entre la spécification d'un système et une représentation de son environnement;
- l'utilisation des propriétés produites par les étapes de validation et d'évaluation du modèle pour caractériser les objets;
- la définition exhaustive d'objets associés aux réseaux de Petri (les G-objets) utilisés pour l'implémentation.

PLAN DE LA THÈSE

Ce document est divisé en deux parties. La première traite du processus de prototypage en tant que méthode ainsi que des techniques qui y sont associées. Elle est constituée des quatre premiers chapitres. Nous y définissons une méthode suffisamment générale pour s'appliquer à différents langages.

La seconde partie applique nos principes au langage Ada. Nous décrivons deux types de prototypes Ada générés automatiquement : *centralisé* (exécution mono-processeur) ou *réparti* (exécution multi-processeurs). Enfin, nous présentons l'architecture et les caractéristiques de l'outil CPN/TAGADA qui assure la Traduction, l'Analyse et la Génération Automatique de code aDA.

Les Annexes détaillent des caractéristiques techniques liées à l'application de notre méthodologie au langage Ada. Les Appendices permettent d'indexer les différentes définitions, figures et références bibliographiques.

Le Chapitre I introduit nos objectifs et situe nos travaux par rapport aux études déjà réalisées dans ce domaine. En particulier, nous présentons les différentes techniques de prototypage à partir de réseaux de Petri, à contrôle centralisé (comparable à une technique de simulation) ou décentralisé (quasi-parallélisme ou parallélisme effectif).

Le Chapitre II décrit la démarche de caractérisation des différents objets pertinents pour la génération (G-Objets) à partir de l'analyse sémantique du réseau de Petri. Chaque ensemble de G-objets définit des fonctionnalités qui sont regroupées dans des modules. L'ensemble de ces modules compose le squelette du prototype.

La production du squelette de l'application repose sur l'analyse des invariants du réseau de Petri en vue de détecter des machines à états [Hack 74] communicantes. Le modèle est décomposé selon les types de G-Objets suivants :

- Un *Processus* est le résultat de l'interprétation d'un flot conservatif déterminé par l'analyse du modèle. Il regroupe plusieurs objets de type Etat_Processus et de type Action.

- Un *Etat_Processus* caractérise un état possible d'un Processus. Il est tiré d'une place du modèle.
- Une *Ressource* privée ou partagée (donnée mémoire, message, article de fichier, …) correspond à une place du modèle.
- Une *Action* décrit le traitement à exécuter. Elle correspond à une transition du modèle. Selon le nombre de processus qui l'exécutent simultanément, une action sera synchronisée ou simple. Elle sera également gardée si son exécution est conditionnée par l'état des Ressources.

Le Chapitre III est consacré à la description des étapes du prototypage. Chacune d'elles enrichit notre connaissance de la spécification et intègre des contraintes supplémentaires. En raison de la diversité des contraintes, nous avons divisé le processus en trois phases successives :

- Identification : le modèle d'une application intègre naturellement l'environnement dans lequel elle s'exécute. Ainsi, un modèle se décompose en deux sous-modèles : celui de l'application et celui de l'environnement, constitué de composants externes.
- Caractérisation : l'utilisation des invariants du réseau de Petri (en particulier les semi-flots) permet une décomposition en processus concurrents et G-Objets. La sémantique du modèle et les synchronisations internes sont étudiées. Un des problèmes majeurs a consisté à prendre en compte les conflits (verrou mortel, gestion des ressources partagées, …).
- Placement : l'intégration des contraintes liées à l'architecture nous permet d=allouer les différentes ressources du système, tant matérielles que logicielles, aux objets dégagés dans la phase précédente.
- Génération : nous tenons compte de la spécificité des objets caractérisés et des nouvelles contraintes introduites par le langage lui-même.

Le Chapitre IV décrit, à partir des fonctionnalités déduites des ensembles de G-objets, l'architecture modulaire du prototype pour des langages impératifs et autorisant la manipulation de tâches. Nous distinguons deux types de modules :

- les modules *externes* : décrivant les traitements unitaires associés et les services offerts par l'environnement du système;
- les modules *internes* : composant le squelette du prototype et assurant les fonctionnalités associées aux G-Objets (Processus, gestion des Ressources...) ou propre aux systèmes parallèles (gestion du contexte d'exécution).

Chaque module est précisément défini par les services qu'il offre et ceux qu'il requiert aux autres modules.

Nous décrivons, dans le Chapitre V, les caractéristiques propres au prototype Ada centralisé. Ces caractéristiques ne sont pas décrites dans le chapitre précédent mais permettent de mettre en œuvre l'architecture qui est développée. Ces caractéristiques sont indépendantes du type de prototype envisagé (centralisé ou réparti).

Dans le Chapitre VI, nous présentons, à partir de notre architecture modulaire, les principes permettant la génération d'un prototype Ada centralisé, s'exécutant sur une machine mono-processeur. Nous y définissons en langage Ada les traitements associés aux différents modules.

Dans le Chapitre VII, nous présentons des principes permettant la génération d'un prototype Ada réparti, s'exécutant sur un ensemble de machines mono-processeur connectées en réseau. Certains modules fonctionnels sont étendus en vue d'offrir de nouveaux services compatibles avec les problèmes posés.

Enfin, le Chapitre VIII est consacré à la description de l'outil de prototypage que nous avons réalisé. Nous y détaillons son architecture interne ainsi que l'environnement dans lequel il a été développé.

PARTIE I

LE PROTOTYPAGE

INTRODUCTION À LA PARTIE I

Cette première partie décrit notre méthodologie de prototypage de systèmes parallèles à partir d'une spécification formelle.

Dans un premier temps, nous replaçons nos travaux parmi les différentes interprétations du terme "prototypage". Notre méthodologie correspond à une approche par *raffinements* [Floyd 84, Hallmann 91].

Tant d'un point de vue économique que technique, le prototypage correspond à une problématique d'actualité. En effet, une étude effectuée par le Government Accounting Office sur la répartition des coûts de neuf projets de taille moyenne (environ 7.000.000 $) [GAO 79] fournit un résultat évocateur : entre ce qui n'est pas livré et ce qui est livré, mais non utilisé ou abandonné, 95% des sommes investies se volatilisent (Figure b).

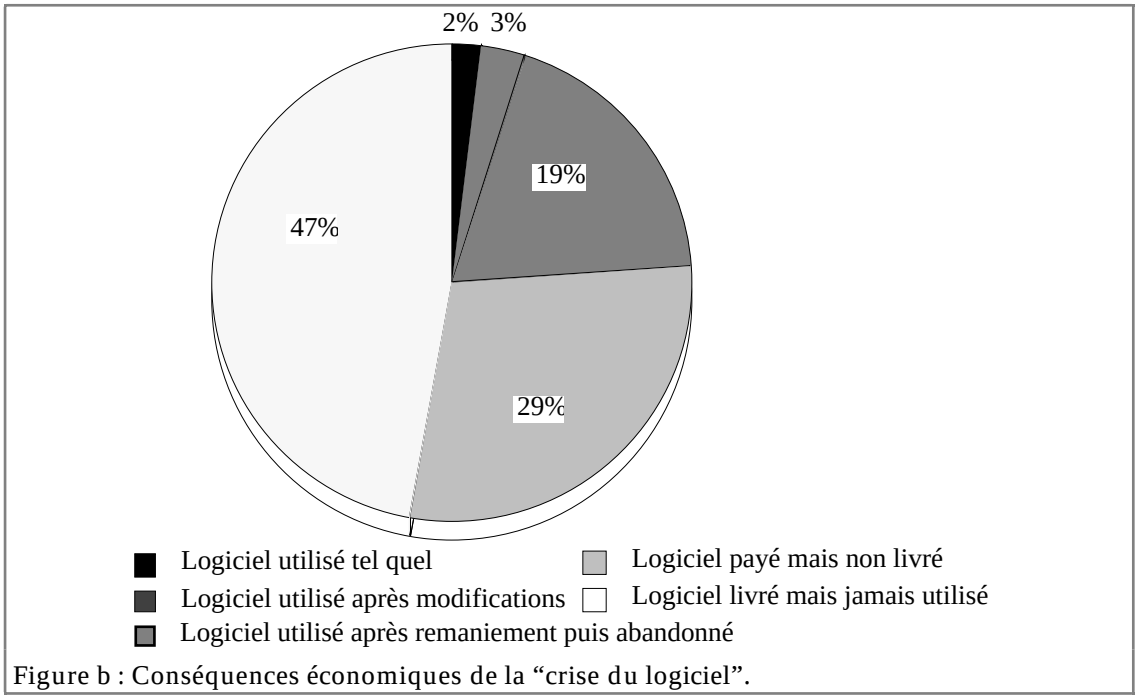


Figure b : Conséquences économiques de la "crise du logiciel".

D'autres études concernent les coûts de maintenance d'un système. [Lientz 78] diagnostique que 60 à 80% de l'effort total de maintenance est une conséquence directe des erreurs et ambiguïtés contenues dans la spécification du système. Le reste est consacré à l'adaptation du logiciel à un environnement évolutif.

Comme il est admis que la maintenance d'un logiciel couvre approximativement 60% des coûts du cycle de vie [Vonk 92], on considère qu'en l'absence de méthode, entre un tiers et la moitié des investissements dans un système sont dus à une mauvaise représentation des besoins réels.

D'après [Boehm 76, Boehm 81], plus une erreur est détectée tard, dans le cycle de vie du logiciel, plus sa correction est coûteuse (Figure c). La correction d'une erreur de conception est d'autant plus délicate et dangereuse qu'elle se situe à un stade avancé de la réalisation, ou de la maintenance d'un logiciel.

Vers la fin des années soixante, les constructeurs et utilisateurs de logiciels diagnostiquent la "crise du logiciel", caractérisée par l'impossibilité de contrôler le développement de projets de grande taille. Le problème avait deux origines :

- un manque de formalisation;
- un manque de méthode.

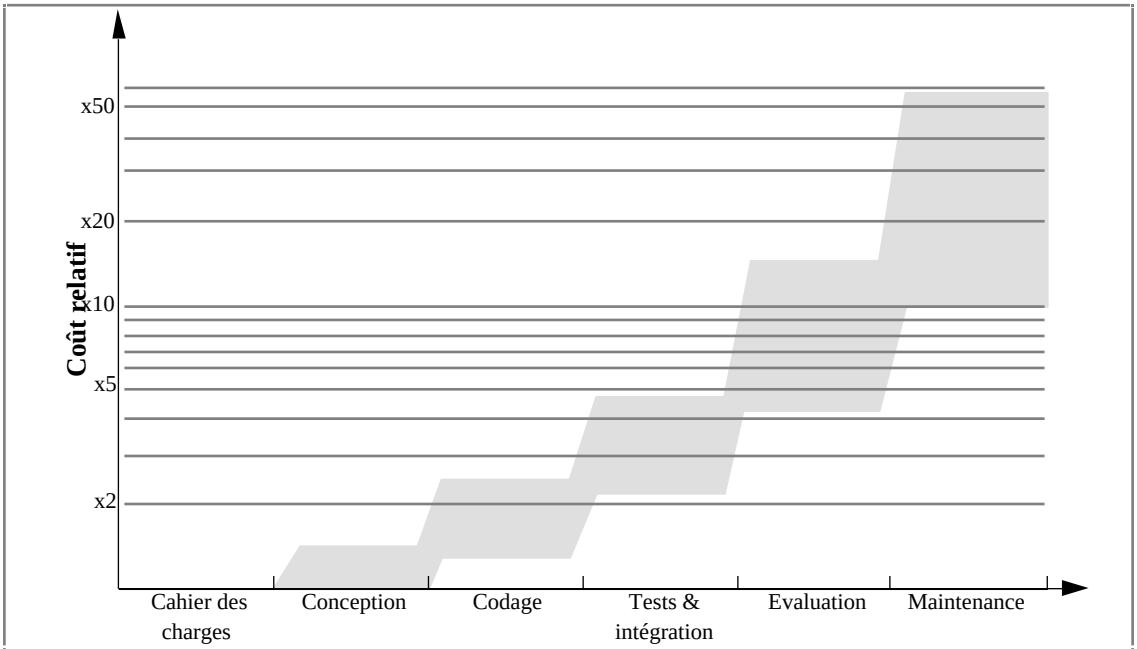


Figure c : Evolution du coût de correction d'une erreur durant les étapes du cycle de vie du logiciel.

Les efforts entrepris dans le domaine de la formalisation de l'écriture de programmes ont abouti à la création de langages de haut niveau comme Pascal [Jensen 78], Modula-2 [Wirth 83] ou Ada [Ada 83]. L'une des idées de base, dans ce domaine, est de confier le maximum de vérifications à un *compilateur*, le concepteur étant chargé des problèmes de conception plus généraux.

Par ailleurs, il n'existait aucune méthode de spécification et de conception. Peu à peu, des groupes de réflexion se sont organisés et le terme "Génie Logiciel" (Software Engineering) apparaît dès 1967.

D'une manière plus générale, une méthode de spécification est l'expression d'un cheminement par étapes permettant d'atteindre l'objectif en respectant des critères de qualité. Elle est obligatoirement basée sur un modèle de description hiérarchique permettant d'aller d'un besoin à la définition d'une solution [Booch 86].

Nous avons basé notre technique de prototypage sur l'apport successif d'informations permettant d'enrichir notre connaissance du système afin de caractériser des modules fonctionnels composant le prototype.

La génération d'un code réalisant un système spécifié à l'aide de réseaux de Petri est un thème de recherche déjà abordé par différentes équipes. L'étude des travaux existants [Silva 79, Silva 80, Silva 82, Valette 82, Nelson 83, Valette 83, Thuriot 85, Bruno 86, Colom 86, Murata 86, Hauschildt 87, Taubner 87, Paludetto 90, Paludetto 91] nous a permi de construire une stratégie permettant l'analyse sémantique de réseaux de Petri Colorés en vue de les interpréter en termes de modules fonctionnels. La définition de ces modules fonctionnels est suffisamment générale pour s'adapter à différents langages procéduraux multi-tâches sans se limiter à la moindre spécificité.

L'étude d'outils développés dans d'autres contextes [Burns 90, Burns 91, Cassigneul 91], ainsi que les différentes méthodes de modélisation envisagées dans l'aide à la réalisation d'un système [Moitessier 91, Coad 92, Vonk 92] nous ont amené à introduire la notion de composant externe.

La validation de la spécification d'un système implique la connaissance du comportement de son environnement. Ensuite, il faut pouvoir interfacer correctement cet environnement avec le prototype. Les composants externes permettent de formaliser les relations entre la spécification d'un système et la représentation de son environnement.

Le Chapitre I présente les travaux sur le prototypage de systèmes parallèles que nous avons étudiés pour définir notre méthodologie.

Dans le Chapitre II, nous proposons, à partir d'un réseau de Petri Coloré, une interprétation sémantique adaptée au processus de prototypage. Nous définissons en particulier la notion de G-objets. Nous décrivons ensuite les algorithmes de décomposition d'une spécification exprimée à l'aide de réseaux de Petri en G-objets. Ces algorithmes utilisent une famille génératrice de semi-flots.

Le Chapitre III présente la notion de composant externe, permettant, à partir d'une spécification modulaire exprimée à l'aide de réseaux de Petri, de séparer la modélisation du système de la représentation de son environnement. Puis nous décrivons les quatre étapes de notre processus de prototypage.

Le Chapitre IV expose la structure d'un prototype généré suivant notre méthodologie. Nous ne faisons pas d'hypothèse sur l'architecture cible (nous envisageons deux types de prototypes : centralisé ou réparti). Les hypothèses relatives aux propriétés du langage cible sont faibles : il doit être procédural et autoriser la gestion du parallélisme, que ce soit de manière interne (gestion d'entité-tâches) ou externe (primitives de manipulation offertes par le système d'exploitation).

+

		Introduction générale
Chapitre I	:	*Prototypage de systèmes parallèles*
Chapitre II	:	*Interprétation sémantique d'un réseau de Petri*
Chapitre III	:	*Méthodologie de prototypage*
Chapitre IV	:	*Structure du prototype généré*
Chapitre V	:	*Caractéristiques du prototype Ada*
Chapitre VI	:	*Le prototype centralisé*
Chapitre VII	:	*Le prototype réparti*
Chapitre VIII	:	*CPN/TAGADA : L'outil de prototypage*
		Conclusion générale
		Bibliographie
		Annexes & Appendices

Chapitre I :

Prototypage de systèmes parallèles

1. Introduction 17
   - 1.1. Objectifs....18
   - 1.2. Spécifier un système....22
   - 1.3. Réaliser le système....24
   - 1.4. Le prototypage....25
   - 1.5. Prototypage et méthodes de spécification....27
2. Quelques outils de prototypage à partir de formalismes autres que les réseaux de Petri 30
   - 2.1. S.A.O....30
   - 2.2. PROTO....33
   - 2.3. Conclusion....37
3. Le prototypage à partir de réseaux de Petri 39
   - 3.1. Les réseaux de Petri....39
   - 3.2. Sémantique des réseaux de Petri....45
   - 3.3. Différentes exécutions d'un réseau de Petri....47
   - 3.4. Conclusion....52
4. Degré de parallélisme d'un système modélisé en réseau de Petri 54
   - 4.1. Les travaux menés à l'université de Saragosse....55
   - 4.2. Les travaux menés à l'université de Toulouse....57
   - 4.3. Les versions d'étude du générateur de code Ada....58
   - 4.4. Conclusion....61
5. Conclusion 62
   - 5.1. L'apport des réseaux de Petri....62
   - 5.2. Automatisation du prototypage....62
   - 5.3. Du prototypage à la génération de code optimisé....63

1. Introduction

La difficulté de mise en œuvre des systèmes complexes tient aux problèmes de spécification, d'organisation, de développement, d'intégration et de vérification d'ensembles d'actions nombreuses et compliquées. Ces problèmes sont aggravés pour les systèmes répartis par la difficulté de maîtriser dynamiquement les interactions de multitudes de programmes, de mécanismes et de personnes œuvrant en parallèle, communiquant, coopérant et partageant des ressources communes.

Les problèmes sous-jacents de communication (synchrone et asynchrone) sont d'autant plus difficiles à contrôler que les comportements des entités peuvent être indéterministes et qu'un fonctionnement réparti empêche toute vision globale.

L'analyse se complique considérablement si l'on exige que les erreurs, défaillances et pannes n'entraînent pas de conséquences désastreuses. Des vérifications sont nécessaires pour garantir que des situations indésirables ne peuvent se produire même dans des circonstances exceptionnelles. Pour diminuer les coûts et le temps de développement, il est donc nécessaire d'effectuer ces vérifications le plus tôt possible.

L'objectif de ce chapitre est d'introduire le processus de prototypage à partir d'une spécification formelle par la présentation de travaux caractéristiques portant sur les systèmes parallèles.

La Figure I.1 présente le cycle de vie du logiciel proposé par l'Association Française pour le Contrôle Industriel de Qualité [Jaulent 90].

Ce modèle montre bien que la démarche de spécification-conception est globalement descendante tandis que la phase de réalisation et de tests est globalement ascendante : il s'agit alors d'assembler les composantes pour obtenir les fonctionnalités requises.

L'axe horizontal représente les phases du projet et la durée de chacune d'elles. L'axe vertical représente le niveau d'abstraction pour l'application. Les phases de spécification et de conception conduisent à définir des niveaux de description de plus en plus détaillés. Pour une application logicielle, le codage est la forme la plus détaillée.

Les phases d'intégration, de test et de vérification permettent d'évaluer la conformité de la réalisation pour chaque niveau de la conception en commençant par les parties les plus élémentaires puis en remontant progressivement jusqu'au produit complet. Pour cela, en correspondance de chaque phase de la conception est associée une phase de test et d'évaluation de la conformité. Ainsi, les deux démarches descendante et ascendante sont complémentaires.

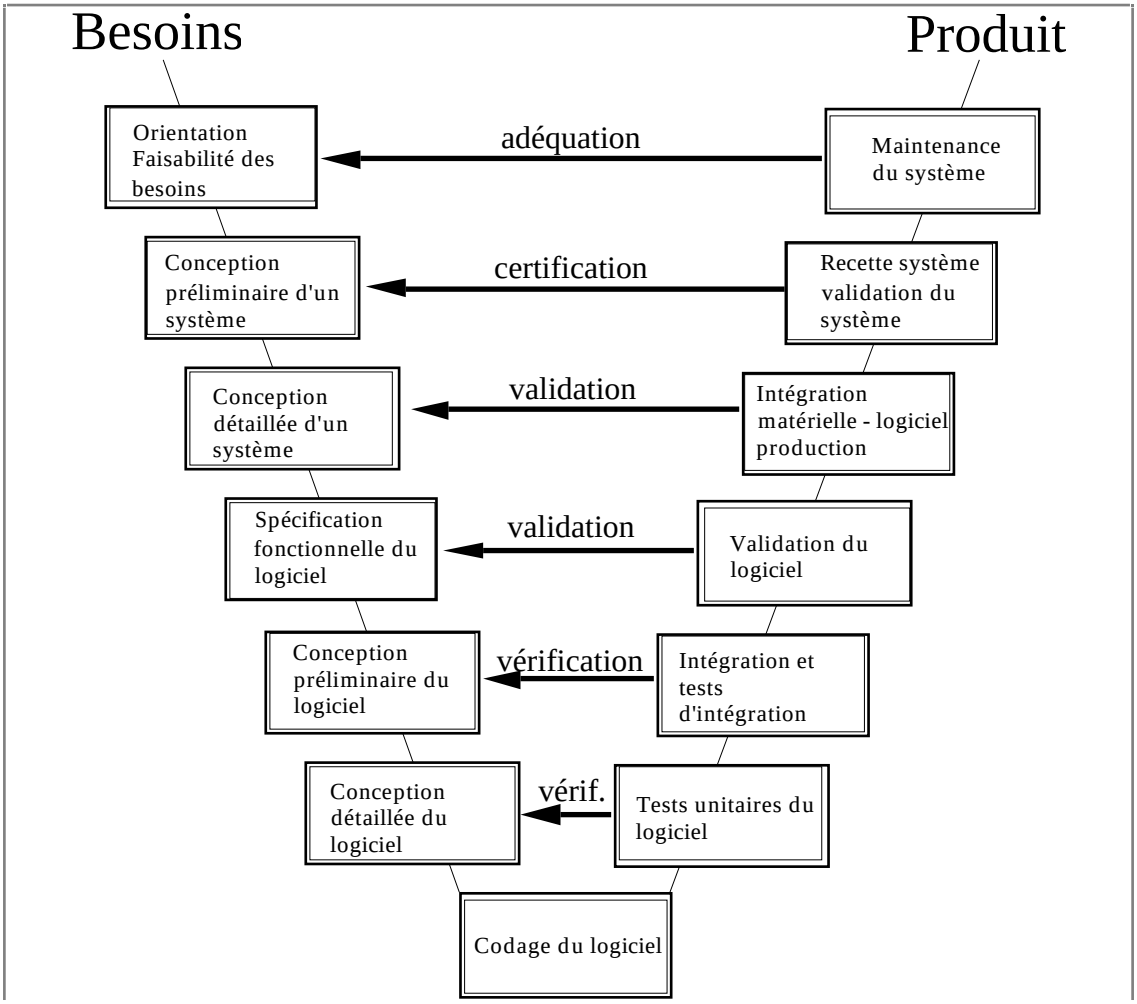


Figure I.1 : Le cycle de vie du logiciel (AFCIQ).

A partir de la spécification détaillée d'un logiciel, nous proposons d'effectuer le prototypage d'un *système.*

Définition I.1 : *Système*

Un système est caractérisé par un ensemble d'unités de programme réalisant une fonction complexe.
Nos travaux concernent les systèmes concurrents (mono-processeur ou multi-processeurs).

Après avoir introduit la problématique du prototypage et le contexte de cette thèse, nous présentons les problèmes posés par la spécification détaillée d'un système. Nous classons ces travaux dans deux groupes : ceux qui se basent sur un formalisme de type réseau de Petri et les autres. Ensuite, nous classons notre processus de *prototypage* parmi les différentes études dont nous avons eu connaissance.

1.1. OBJECTIFS

Nos travaux s'inscrivent dans le cadre du projet MARS (Modélisation, Analyse et Réalisation de Systèmes) dont l'objectif est de réaliser un atelier de conception de systèmes parallèles [Estraillier 91, Estraillier 92]. Le but est de

fournir aux ingénieurs des outils logiciels de conception de systèmes répartis afin de leur permettre, non seulement de spécifier (informellement au départ puis formellement ensuite) et de vérifier la correction de leur modèle, mais aussi de produire un prototype conforme à leurs spécifications.

Un tel atelier logiciel correspond à un réel besoin car la multiplication des techniques de spécification formelle et l'essor des techniques d'analyse rendent très difficile le partage de résultats entre outils. La plupart des applications actuelles n'utilisent pas de représentation intermédiaire commune et il n'y a pas de gestion globale des résultats sur une spécification analysée avec différents outils.

Les domaines d'application visés par le projet sont les systèmes distribués, les protocoles de communication, les applications bancaires…. Ces domaines d'application ont la particularité d'exprimer le parallélisme et l'indéterminisme des événements, ils sont donc difficiles à appréhender sans environnement logiciel.

1.1.1. Méthodologie générale

Le premier aspect à prendre en compte lors de la réalisation d'un système (quel qu'il soit) est de définir les besoins de ses futurs utilisateurs.

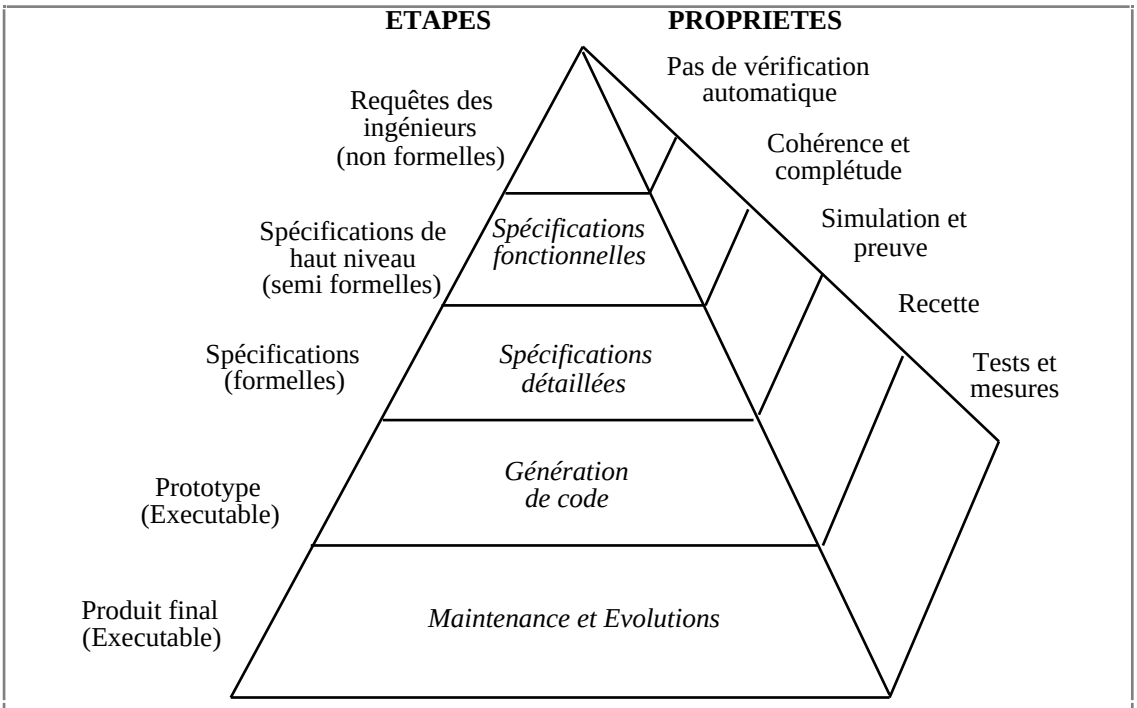


Figure I.2 : Etapes de réalisation d'un logiciel.

La conception implique le choix d'un ensemble de modèles adaptés au problème étudié. Le processus de modélisation intègre alors, sous forme d'optimisations progressives, l'ensemble des besoins identifiés. La plupart des méthodes de conception adoptent une approche à plusieurs niveaux d'abstraction qui vont d'une perception conceptuelle (fonctionnelle) jusqu'à la définition détaillée de son implantation physique. Cette démarche permet de hiérarchiser les choix de conception et d'éviter (dans la mesure du possible)

que des décisions du niveau conceptuel ne soient tributaires des choix préalables relevant de l'implantation physique.

Le métier d'ingénieur évolue vers la *formulation de l'énoncé* du problème (ou plutôt de la solution choisie) suivant un certain *formalisme* jusqu'à l'obtention d'une solution plutôt que vers la réalisation effective [Arthaud 89]. Il vaut mieux détecter le plus tôt possible qu'un énoncé paraît absurde (est faux). Ainsi, à chacun des niveaux correspondent des types de vérification différents.

La Figure I.2 présente les différentes étapes assurant le passage d'une spécification informelle à la génération de code.

Pour détecter le plus tôt possible les erreurs ou les inadéquations d'une spécification, il faut utiliser des formalismes adaptés aux domaines d'application et validables automatiquement. Lors des différentes étapes de développement d'un système, plusieurs formalismes sont souvent utilisés. Ils doivent permettre des vérifications avant de passer à l'étape suivante.

1.1.2. Mise en œuvre

Il est clair que les étapes de la méthodologie proposée doivent pouvoir être supportées par des outils logiciels. L'accroissement du nombre des logiciels amène à souhaiter l'existence de bases de connaissances intégrant de nombreuses descriptions, propriétés, versions, pour fournir un environnement adapté à la conception collective de systèmes.

Une *plate-forme commune d'outils logiciels* est nécessaire pour assurer l'interfaçage entre les formalismes, faciliter la manipulation des modèles appropriés à chaque étape et garantir la cohérence des analyses lors des modifications de modèles.

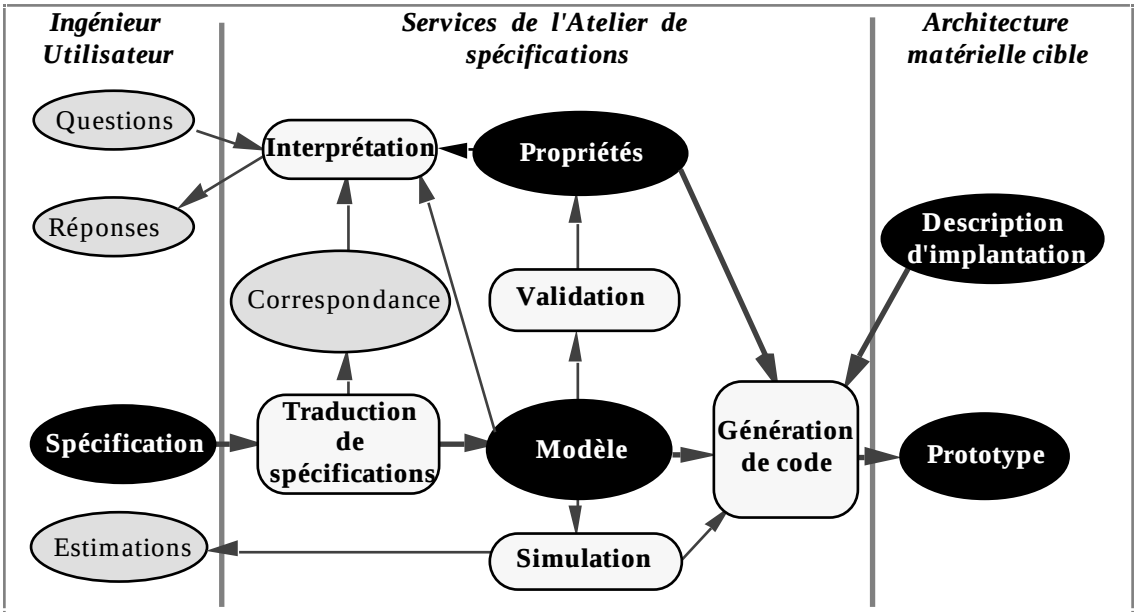


Figure I.3 : Interactions entre les différents outils de la plate-forme.

De tels outils de conception doivent en particulier :

- Permettre la conception de systèmes complexes. Le formalisme de spécification doit être structuré, modulaire et reposer sur une méthodologie bien définie. Il doit également être facilement compréhensible, et éventuellement dédié à un type d'application spécifique (systèmes de téléphonie, de télécommunications...).
- Assurer la traduction (semi-automatique) d'une spécification semi-formelle en spécification formelle, seul support d'une validation comportementale.
- Permettre l'analyse (éventuellement par étapes successives) du système spécifié jusqu'à l'obtention de la garantie de son bon fonctionnement. Un ensemble d'outils d'analyse doit fournir une validation formelle et interactive du système spécifié.
- Fournir les résultats d'analyse permettant la correction d'une spécification donnée. Ils doivent être suffisamment clairs et retranscrits dans le langage de spécification originel afin d'être utilisables, dès réception.
- Permettre éventuellement des vérifications spécifiques (évaluations de performance, résolution de problèmes d'ordonnancement...).
- Permettre l'animation automatique (ou quasi-automatique) de la spécification validée.
- Assurer le prototypage de la spécification validée par génération interactive de code.

La Figure I.3 présente succinctement les interactions entre les différents outils existants. La spécification initiale est d'abord traduite en un modèle qui joue un rôle de pivot pour toute la méthodologie. En effet, on peut en tirer des estimations par simulation, des propriétés par analyse, ou un prototype par génération de code. Pour répondre aux questions de l'ingénieur sur les propriétés de son modèle, il est essentiel de disposer de tables de correspondance entre la spécification et le modèle.

La Figure I.4 présente les étapes du cycle de vie couvertes par la méthodologie générale et par le prototypage. On remarque que l'approche formelle permet de déporter un effort de validation dès la conception, tandis que le passage d'un niveau à l'autre est assuré par des outils de traduction interactifs.

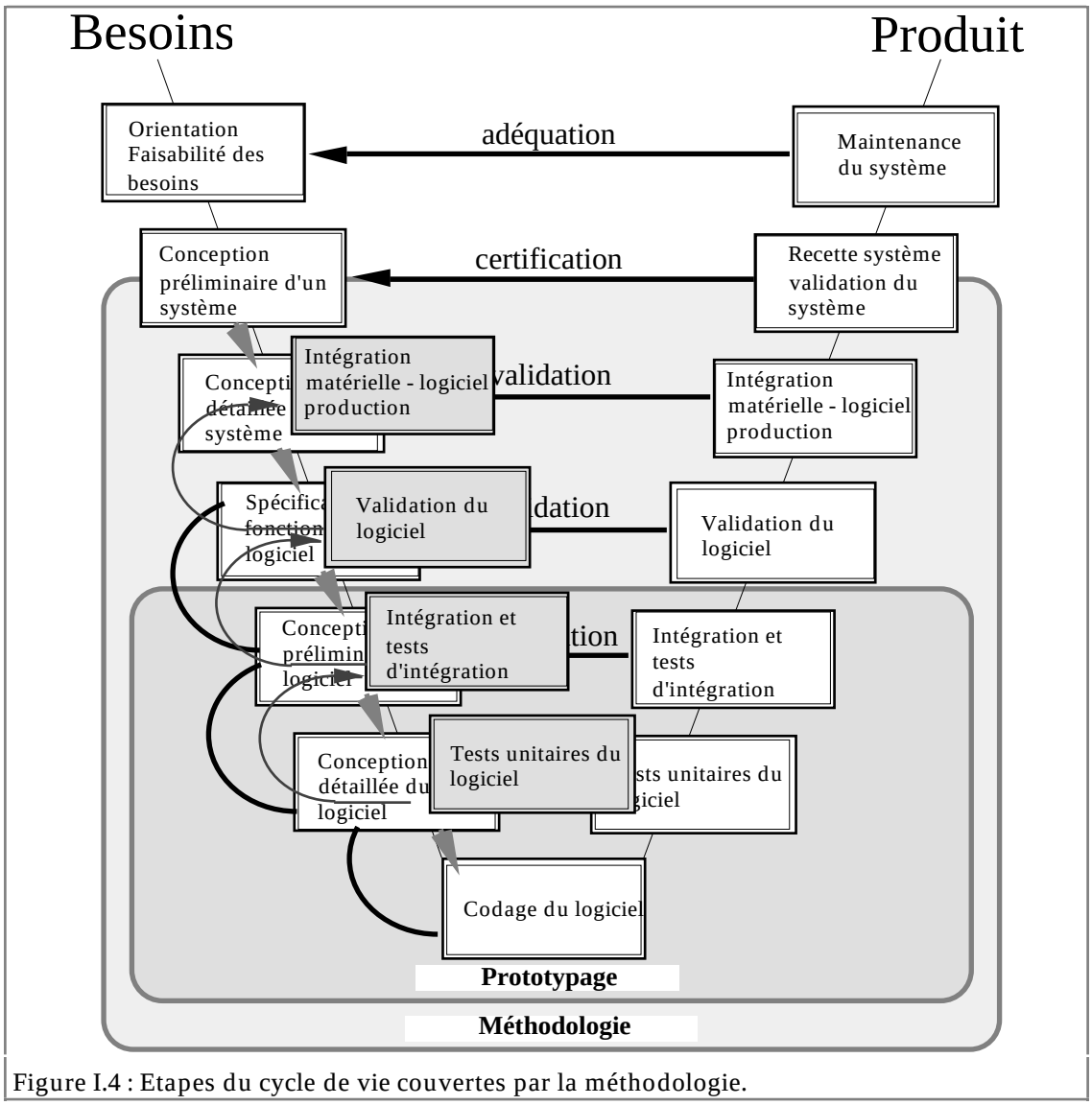


Figure I.4 : Etapes du cycle de vie couvertes par la méthodologie.

1.2. SPÉCIFIER UN SYSTÈME

La spécification d'un système est une représentation formelle de son comportement et des données qu'il manipule. Pour cela, il faut construire un *modèle*. Nous définissons deux types de modèles : fermés et ouverts.

Définition I.2 : ***Modèle fermé***

> Représentation formelle d'un système muni de la représentation abstraite de l'environnement dans lequel il s'exécute.

Un modèle fermé peut être vu comme une "boîte noire" contenant le système et son environnement. Si le formalisme utilisé pour énoncer un modèle l'autorise, il est possible de déduire des informations sur son comportement et de prouver qu'il apporte une solution correcte au problème.

Un modèle fermé permet ainsi la *validation* du modèle du système dans son environnement car il décrit le comportement du système en fonction des différentes interactions possibles avec son environnement.

Exemple I.1 : Considérons la réalisation d'un commutateur téléphonique gérant l'interconnexion d'abonnés. Le modèle fermé comporte :

- la spécification du commutateur (le système),
- la spécification des abonnés (l'environnement).

Il est impossible de valider le commutateur sans tenir compte des réactions possibles des abonnés.

Définition I.3 : *Modèle ouvert*

Représentation formelle d'un système sans son environnement. Un modèle ouvert doit :

- préciser les services requis, nécessaires à l'exécution du système;
- définir proprement les services offerts à l'environnement;
- expliciter les relations entre les différents types de services.

Un modèle ouvert est contenu dans un modèle fermé (Figure I.5). Le modèle ouvert décrit formellement un système. Les hypothèses et les propriétés attendues [ISO 7498] du système peuvent ainsi être validées formellement à l'aide du modèle fermé.

Les services requis et offerts [Bachatene 92] relient le modèle ouvert à l'environnement dans lequel il évolue. Ils définissent *la partie visible*. Si l'environnement, ou le système, sont décomposés en plusieurs modules, la définition et l'utilisation de *clauses de visibilité* [Ada 83, Booch 87, Coad 92] sont nécessaires.

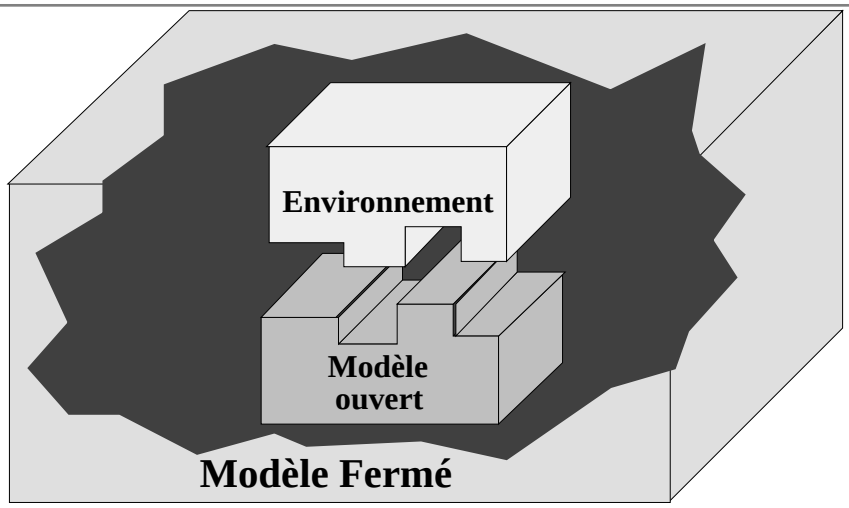


Figure I.5 : Décomposition d'un modèle fermé en un modèle ouvert plus la représentation de son environnement.

Les traitements permettant, à partir d'un ensemble de paramètres, de fournir un résultat, n'ont aucune interaction avec le monde extérieur : ils appartiennent à la *partie privée* du modèle. Si plusieurs traitements donnent un même résultat, ils sont parfaitement interchangeables.

Définition I.4 : *Partie visible et partie privée d'un module*

La partie visible d'un module est constituée de la déclaration des services offerts. Tout module ayant défini qu'il *voit* ces services, à l'aide de clauses de visibilité, peut ainsi en disposer.

La partie privée d'un module est constituée des fonctions internes au module permettant de réaliser les services offerts. De telles fonctions utilisent éventuellement des services requis (offerts par d'autres modules). Rien de ce qui est défini dans la partie privée d'un module n'est utilisable à l'extérieur.

Exemple I.2 : Considérons la réalisation d'un commutateur téléphonique (Exemple I.1). Le modèle ouvert correspond à la spécification du système, c'est-à-dire à la spécification du commutateur. Si l'on considère un objet abonné, le système doit voir les services qu'il offre (décrocher, raccrocher, composer un numéro...).

Lorsque le problème est complexe, le modèle est décomposé en sous-modèles plus simples, définissant de la sorte une *hiérarchie*.

Le modèle est un *énoncé* du problème, il peut être partagé dans le cadre d'une réalisation collective. Il constitue donc un outil pour les personnes qui auront à le réaliser. Nous verrons, dans le Chapitre IV, les techniques que nous avons développées afin d'automatiser le processus de réalisation.

1.3. RÉALISER LE SYSTÈME

Une fois le modèle fermé spécifié puis validé, il faut réaliser le système associé. Ceci n'est pas une simple compilation : en effet, la sémantique du système peut donner lieu à plusieurs interprétations possibles [Murphy 89]. Il faut choisir la plus efficace en fonction de critères qui auront été définis, et, par conséquent, *interpréter* le modèle. Cette tâche est compliquée par le fait que, sur des projets complexes, ceux qui réalisent un système ne sont pas forcément ceux qui l'ont modélisé.

La difficulté de l'interprétation dépend naturellement de la précision du formalisme dans lequel est exprimé le modèle.

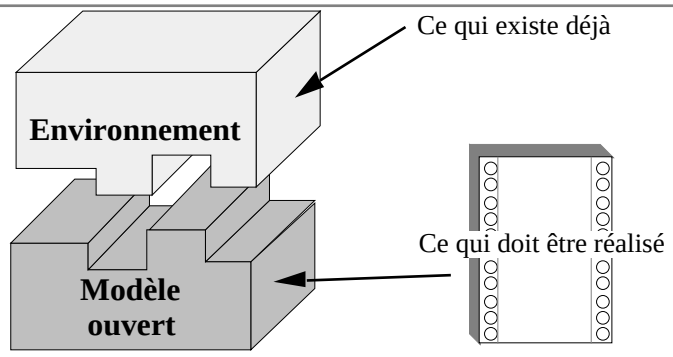


Figure I.6 : Les différentes composantes d'un modèle.

Puisque, par définition, un système est intégré dans un environnement avec lequel il interagit, lors de la réalisation du système, l'application doit être *interfacée* avec son environnement (Figure I.6).

Exemple I.3 : Il faut réaliser le commutateur téléphonique évoqué dans Exemple I.1 : le modèle ouvert correspondant à la description du commutateur téléphonique doit être réalisé et interfacé avec un ensemble d'abonnés.

L'interprétation d'une spécification en vue d'écrire l'application associée peut être décomposée en deux étapes :

- dans un premier temps, il faut repérer le sous-modèle correspondant à l'application en elle-même et "supprimer logiquement" tout ce qui concerne l'environnement;
- ensuite, il faut extraire de la spécification les informations permettant de réaliser le système.

1.4. LE PROTOTYPAGE

Cette section présente les différentes formes de prototypage. Chaque forme correspond à une utilisation différente, en fonction des objectifs initiaux, du prototype obtenu [Dollas 90]. Certaines formes de prototypage sont incompatibles avec certains domaines d'application.

[Hallmann 91] propose une classification intéressante des méthodes de prototypage :

- l'approche incrémentale;
- le maquettage : nous différencions les maquettes simulées des maquettes autonomes, c'est-à-dire indépendantes de l'environnement dans lequel elles ont été créées;
- l'approche par raffinements.

1.4.1. L'approche incrémentale

L'approche incrémentale [Hallmann 91] s'apparente au développement traditionnel d'une application. Etape par étape, des modules d'une application sont ajoutés à une architecture globale conçue auparavant. Les modules non implémentés à une étape donnée sont simulés afin de permettre le test de l'ensemble de l'application.

Cette approche présente un gros défaut : elle n'est réellement intéressante que si l'architecture du prototype est robuste. En effet, les effets de bord liés à l'ajout d'un nouveau module sont inévitables, surtout dans le domaine des applications parallèles.

Nous ne considérons pas une telle approche comme une technique de prototypage au sens où nous l'entendons. Il s'agit de méthodes de développement d'une application. Le prototype ainsi réalisé ne repose pas sur une modélisation : il *est* le modèle.

1.4.2. Le maquettage

Une "maquette" ne possède pas toutes les fonctionnalités du système. Sa réalisation ne lui permet pas de se substituer au système, pour des raisons de fiabilité par exemple.

Le prototype se réduit à une réalisation permettant de faire des études de faisabilité. La maquette apporte des informations dans les domaines suivants :

- La spécification du problème : il se peut que, lors de la résolution du problème, ou pendant l'établissement du cahier des charges, certains points aient été négligés ou sous-estimés. L'élaboration d'une maquette permet alors de corriger le modèle, afin que la réalisation finale ne souffre pas des mêmes "erreurs de jeunesse".
- Le coût du projet : il est possible, à partir de l'expérience obtenue dans la réalisation du prototype, d'obtenir un ordre de grandeur du coût global du projet.

Le prototype est ici considéré comme un produit "jetable", il n'est pas prévu de récupérer les éléments de la maquette. Dans le domaine du matériel, c'est quasiment la seule approche envisagée : avant de concevoir un système, un prototype est réalisé dans une technologie peu coûteuse mais plus lente et plus volumineuse. On y trouve parfois un mélange de logiciel et de matériel.

Diverses études présentent deux types de maquettes : simulées ou autonomes.

Maquette simulée

Le prototype peut être vu comme une simulation d'un modèle fonctionnant dans l'environnement où elle a été conçue. Si l'on possède les outils de simulation adéquats, cette approche du maquettage reste peu coûteuse.

Dans cette catégorie, [Lintulampi 90] propose un outil de simulation et de validation pour des systèmes logiciels temps réel. Les travaux présentés dans [El Fallah 89] visent également la validation. [Royals 90] propose d'insérer, à un catalogue de fonctions prédéfinies intégrées au niveau matériel (multiplexage, adressage...), la simulation logicielle des composants en cours de test. La rapidité des fonctions de base permet de compenser la lenteur de la simulation.

Maquette autonome

Le prototype peut également évoluer en dehors de l'environnement dans lequel il a été créé.

Dans cette catégorie, [Petersen 90] propose une méthode basée sur la simulation, puis une implémentation rapide à base de PLA. La simulation permet d'effectuer des tests unitaires. Le pseudo-composant obtenu sous forme de PLA peut être intégré à un environnement réaliste en vue d'effectuer des tests d'intégration.

Une telle approche est plus coûteuse puisqu'elle implique un travail qui n'est pas forcément réutilisable (recherche des propriétés, création d'outils de mesure...). Elle n'est donc envisagée que dans des projets de grande envergure, avant de continuer, ou dans des projets à long terme.

1.4.3. L'approche par raffinements

Dans l'approche par raffinements (aussi appelée "prototypage en cascade"), le système est conçu de façon que de nouveaux concepts, tant architecturaux que

comportementaux, puissent être introduits : rien ne doit être fixé a priori. Le procédé permet d'obtenir des versions par raffinements successifs [Floyd 84, Vonk 92].

Dans le cadre d'une telle approche, [Budde 84] propose quatre étapes :

- choisir un prototypage "vertical" ou "horizontal". Ils sont définis comme suit :
  - prototypage "vertical" : certaines fonctions sont choisies et réalisées pour la version finale,
  - prototypage "horizontal" : les fonctions sont implémentées quel que soit le niveau de détail;
- construire et préparer le prototype;
- évaluer le prototype;
- décider si le prototype est suffisamment fiable et complet pour être une version finale ou s'il faut réappliquer un cycle complet.

Une telle méthode suppose des outils de modélisation, de réalisation et d'évaluation importants. Lorsque le processus est terminé, le prototype devient la "version 1" d'un produit. Sa réalisation doit être beaucoup plus soigneuse.

Pour des raisons technologiques évidentes, une telle approche est surtout réalisable dans le domaine du logiciel. Les contraintes de fiabilité dans la réalisation du prototype sont beaucoup plus fortes que dans le cas d'une "maquette". Une telle approche a été abordée avec succès dans le projet S.A.O. [Cassigneul 91].

Les travaux que nous présentons se situent dans cette catégorie. L'objectif est de permettre à un utilisateur d'obtenir un programme parallèle exploitable, par raffinements successifs de ses spécifications. Le développement, dans ce cas, porte sur les spécifications [Estraillier 91].

1.5. PROTOTYPAGE ET MÉTHODES DE SPÉCIFICATION

Apparemment simple, la notion de modèle cache un grand nombre de concepts et de sous-entendus. Elle suppose tout d'abord que le système qui doit être modélisé a été bien compris par ceux qui devront le réaliser. Les systèmes développés à l'heure actuelle deviennent de plus en plus complexes : ils sont alors décomposés en sous-problèmes plus faciles à appréhender.

Avec l'émergence des techniques de Génie Logiciel, permettant de réaliser de façon plus sûre des systèmes sans cesse plus complexes, sont apparues un grand nombre de techniques de représentation.

Tout modèle est exprimé dans un formalisme particulier. La formalisation des spécifications est indispensable pour :

- La communication : l'utilisation d'un standard, compréhensible par plusieurs individus, constitue une raison suffisante pour créer des règles de représentation d'un problème;

- La gestion d'un projet : il s'agit de formaliser le développement. Cela permet de développer des systèmes d'aide à la production de logiciel [Case 86].

Un grand nombre de méthodes, basées sur des principes différents, ont été élaborées. Chacune d'elles donne de bons résultats dans l'étude de certains aspects d'une problématique [Coad 92] :

- l'approche fonctionnelle s'intéresse à la sélection des étapes (ou des sous-étapes) d'un traitement, jusqu'à ce que les actions soient considérées comme atomiques;
- l'approche "DataFlow" se concentre sur le cheminement des données entre les différentes unités fonctionnelles d'un système [De Marco 78];
- l'approche "modélisation de l'information" se focalise sur la description des entités du système à l'aide d'attributs ainsi que sur leurs associations [Loomis 87].

A l'heure actuelle, la majorité des formalismes tendent vers une approche orientée objets. La décomposition fonctionnelle "pure" montre très rapidement ses limites lorsque les systèmes deviennent importants, ou sont sujets à évolution. La méthode DataFlow "pure" devient inextricable lorsque le nombre d'événements à traiter croît.

La nécessité d'introduire une hiérarchie est évidente à partir du moment ou l'on considère des systèmes complexes. Les formalismes basés sur une approche orientée objets [Booch 86, Meyer 88, HOOD 89] sont tous hiérarchiques. Ils sont basés sur la notion de *classes*, instanciées en *objets*. Chaque *classe* est un modèle de comportement caractérisé par des *méthodes*. Il est possible de dériver des classes existantes (*héritage*) et d'ajouter des fonctionnalités (*enrichissement*).

Les formalismes doivent permettre d'obtenir des informations, qu'elles soient d'ordre syntaxique (complétude et cohérence) ou sémantique (comportementales). Si seules des informations d'ordre syntaxiques peuvent être obtenues, la validation comportementale ne peut être que *quantitative* : elle est obtenue par simulation du modèle. Sur la base d'informations d'ordre sémantique, une validation *qualitative* (formelle) est possible.

Les méthodes les plus au point, actuellement, sont toutes de type approche orientée objets [Arbaoui 91, Coad 92]. Leur principale faiblesse repose cependant sur l'absence de validation formelle, contrairement aux spécifications algébriques [Gaudel 91] ou aux réseaux de Petri [Brams 82, Arnold 92].

L'association f'une méthodologie orientée objets aux réseaux de Petri [Di Giovanni 90, Paludetto 90, Paludetto 91], comme l'intégration de la hiérarchie [Hubert 90] sont actuellement des thèmes de recherche. Cela permet de cumuler les avantages des deux méthodes.

2. Quelques outils de prototypage à partir de FORMALISMES AUTRES QUE LES RÉSEAUX DE PETRI

Cette section présente brièvement deux travaux concernant le prototypage à partir de modèles.

Le premier a été développé à l'Aérospatiale [Cassigneul 91]. Il est opérationnel et l'impact de son utilisation dans des projets industriels a pu être évalué. Les prototypes produits sont réutilisables dans le développement de logiciels plus importants. C'est une illustration intéressante de l'approche incrémentale.

L'autre fait l'objet d'un développement aux ROME Laboratories, dans le cadre d'un contrat avec le Department of Defense américain [Burns 90, Burns 91]. C'est un prototypage de type "maquettage", la maquette étant fortement liée à l'environnement dans lequel elle a été conçue. Cependant, les possibilités offertes au concepteur d'une application parallèle, en particulier dans la description d'architectures cibles, nous ont paru fort intéressantes.

2.1. S.A.O.

S.A.O. (Spécification Assistée par Ordinateur) a été mis au point à l'Aérospatiale dans le cadre du développement de produits embarqués.

Des travaux prospectifs avaient été menés pendant la mise au point de l'Airbus A320 (spécification et travail sur l'un des calculateurs de bord). Devant le succès de cette approche, le développement et l'extension de l'outil ont été poursuivis. Le processus est actuellement appliqué avec succès dans la réalisation de l'A340, en particulier pour la mise au point du calculateur des commandes de vol (traduction des ordres du pilote aux commandes, pilotage automatique). Il s'agit d'applications critiques : un degré de fiabilité élevé est exigé.

Dans un premier temps, nous décrivons brièvement la représentation d'une spécification. Nous détaillons ensuite la méthodologie utilisée avant d'étudier l'impact de l'utilisation de S.A.O.

2.1.1. Les planches S.A.O.

Le formalisme d'expression des modèles est utilisé depuis longtemps par les automaticiens et équipementiers travaillant avec l'Aérospatiale.

Il s'agit d'une représentation non ambiguë composée d'un ensemble de planches. Il existe deux types de planches :

- Les planches décrivant une fonction : elles sont composées de connecteurs de type "composant", reliés par des arcs (Figure I.7);
- Les planches décrivant un ordre de succession et la fréquence d'utilisation des planches : elles indiquent l'ordonnancement des planches entre elles. Si le système ne comporte ni contraintes

d'ordonnancement, ni contraintes temps-réel, ces planches ne figurent pas dans sa description.

En outre, chaque planche contient des informations de gestion permettant de l'identifier. Il s'agit de sa date, du numéro de la version, de l'auteur ainsi que des références propres à l'outil de gestion d'un modèle. La spécification d'un calculateur de bord est divisée de façon hiérarchique : elle est contenue dans un livre, divisé en chapitres composés de plusieurs planches.

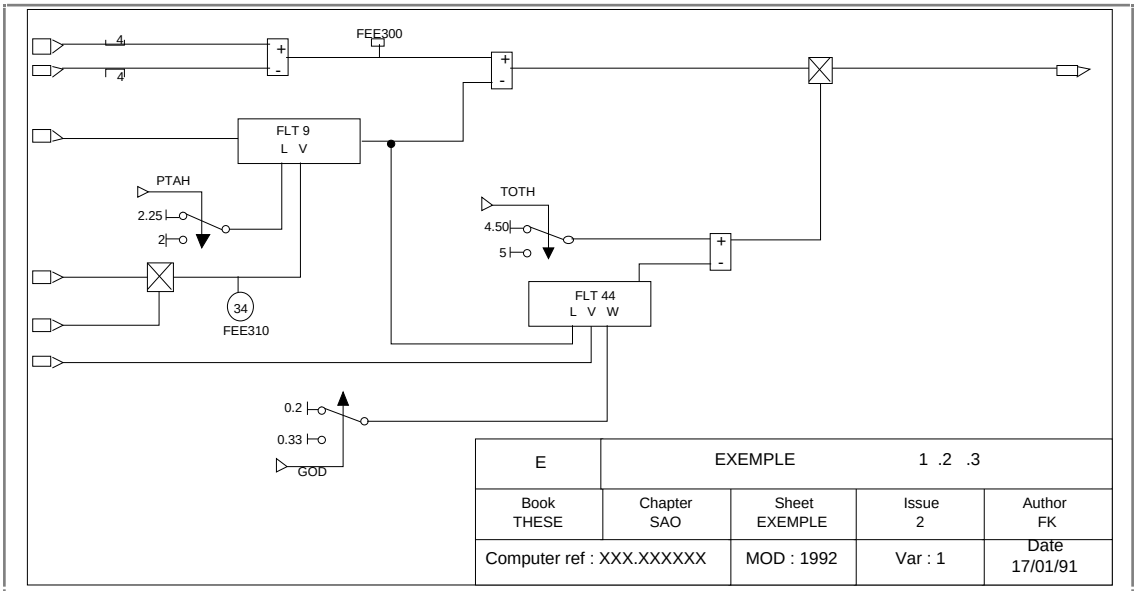


Figure I.7 : Exemple de planche S.A.O.

Des outils graphiques ont été définis (sur des stations de travail Sun). Ils permettent de saisir des planches et de gérer plusieurs versions incrémentales du modèle d'un système. Cela est particulièrement important dans la phase de développement mais aussi dans la phase de commercialisation.

En effet, les calculateurs, à bord de différents avions, ne sont pas identiques : la version de l'avion, le type d'utilisation et les desiderata des compagnies sont des paramètres que le constructeur doit prendre en compte. La spécification finale est ainsi paramétrable : plusieurs versions peuvent être produites simultanément en fonction des souhaits exprimés par les clients.

2.1.2. L'outil S.A.O.

Un système embarqué (pilote automatique...) se réalise par raffinements successifs. Un modèle est donc construit, le prototype associé généré et testé sur un simulateur. Ensuite, en fonction des résultats, le modèle doit être transformé afin d'obtenir un nouveau prototype.

Les applications étant soumises à de fortes contraintes temps-réel, elles sont directement réalisées en langage d'assemblage. Le développement du prototype n'en est pas facilité. L'automatisation du prototypage a été envisagée :

- afin de réduire la durée de réalisation du prototype;

- afin d'augmenter la conformité du prototype par rapport aux spécifications.

La conception d'un système est réalisée en plusieurs étapes à l'aide d'une suite d'outils :

- Simulation des lois de pilotage : elle est réalisée à l'aide d'OCAS (Outil de Conception Assistée par Simulation) qui permet d'effectuer une première validation d'un système. C'est un "simulateur de vol du bureau" qui est en général utilisé au niveau des planches S.A.O.
- Etude du comportement en parallèle de plusieurs calculateurs : elle est réalisée avec OSIME (Outil de SImulation Multi-Equipement). Il permet d'étudier la façon dont ils cohabitent, d'effectuer des mesures de tolérance aux pannes (étude de la redondance) et de détecter les problèmes d'asynchronisme.
- Simulation d'entraînement : elle est réalisée avec un simulateur "en cabine" beaucoup plus complet et efficace qu'OCAS.
- Réalisation du logiciel implanté sur les équipements : un générateur de code en langage d'assemblage produit des programmes à partir des planches S.A.O. Seuls les Systèmes d'Exploitation (auto-test, surveillance des bus, démarrage, reset...) sont codés une fois pour toute "à la main".

  S.A.O. calcule, en fonction des planches modifiées, les zones de code qui doivent être réécrites. Puis la génération se fait automatiquement. Elle se poursuit par la compilation des programmes réécrits en vue d'obtenir un exécutable.

2.1.3. Etude critique

L'outil S.A.O. est opérationnel : plus de 70% des logiciels embarqués pouvaient être spécifiés à l'aide de planches S.A.O. Le formalisme est très adapté au type d'applications développées et permet d'atteindre rapidement un bon niveau de spécification.

S.A.O. permet de gérer l'instabilité d'un logiciel en cours de développement, ainsi que la livraison de multiples versions adaptées à des utilisations différentes d'un même modèle d'avion.

L'efficacité de l'approche par raffinements a pu être prouvée sur des systèmes de grande taille et à degré de fiabilité élevé. Les programmes produits sont exempts d'erreurs de programmation ou d'interprétation des spécifications.

L'impact de l'utilisation du logiciel a pu être évalué en termes de performances et de sécurité :

- Il faut environ quatre heures pour, à partir de la modification d'un livre définissant un calculateur, produire un nouveau programme, contre plusieurs jours auparavant. Le procédé est donc très intéressant dans la phase de mise au point, jusqu'au prototype volant de l'avion.

  Dans la réalisation du système de pilotage automatique de l'Airbus A340, les simulations ont lieu dans la matinée. Les modifications des planches S.A.O, en fonction des résultats, sont réalisées dans l'après-midi. Durant la nuit, S.A.O. reconstruit l'application (certains systèmes comportent

jusqu'à 1000, voire 2000 planches S.A.O.) et fournit une nouvelle version pour le lendemain.

- Si le formalisme utilisé ne permet aucune validation comportementale *a priori* (seules des simulations sont effectuées) les logiciels produits sont exempts d'erreurs d'interprétation des spécifications. Le générateur de code étant certifié, il est possible de supprimer les tests unitaires des différents calculateurs. Seuls les tests d'intégration sont à effectuer.

  Les programmes étant générés automatiquement en langage d'assemblage, S.A.O. peut vérifier des contraintes temps-réel (comptage des cycles machines de façon pessimiste). Les concepteurs du logiciel peuvent ainsi obtenir des informations sur la durée d'exécution d'un ensemble de planches S.A.O. et, en fonction des résultats, reconsidérer leur conception.

L'utilisation des programmes générés par S.A.O. avec d'autres unités définies avec des méthodes "traditionnelles" ne pose aucun problème. Elle s'effectue au niveau de l'édition de liens.

Le prototype produit par S.A.O. est directement utilisable. Il ne s'agit donc pas d'une maquette. Cependant, la génération des programmes est bien adaptée à la conception d'un logiciel séquentiel. La gestion d'architectures multi-processeurs n'est pas prévue; le parallélisme et la redondance sont étudiés à part.

2.2. PROTO

PROTO est à la fois une méthode, un formalisme de définition d'un modèle et un outil de prototypage rapide [Burns 90]. Elle s'inspire de la méthode DataFlow [De Marco 78].

PROTO offre une approche graphique dans la construction de diagrammes de type DataFlow décrivant un système. Un modèle PROTO est défini sous forme d'un diagramme et de commandes exprimées dans un langage dédié. Le modèle peut être interprété à des fins de validation des systèmes ainsi décrits.

Dans sa version finale [Burns 91], PROTO permet de construire des modèles d'applications et des modèles d'architectures. Les applications peuvent ainsi être implantées sur des structures matérielles différentes et leur efficacité étudiée.

2.2.1. Les diagrammes PROTO

Le formalisme consiste en un graphe appelé diagramme, contenant plusieurs types de sommets (Figure I.8) :

- Les *composants* : il s'agit de fonctions appliquées à des paramètres et produisant un résultat en sortie. Un composant peut avoir plusieurs paramètres et produire plusieurs résultats.
- Les *zones de stockage des données* : il s'agit de "registres" dans lesquels les données transitent temporairement.
- Les *arcs* : ils relient les zones de stockage des données aux composants.

- Les *aiguillages* : ils permettent de connecter plusieurs arcs et indiquent que le résultat d'un composant peut être dirigé vers l'un ou l'autre des composants successeurs.
- Les *connecteurs* : ils permettent de nommer les arcs et d'éviter, par là-même, leur prolifération. Les connecteurs ne sont là que pour rendre le graphe plus lisible.

Figure I.8 : Les différents objets composant un diagramme PROTO.

La Figure I.9 donne un exemple de modèle PROTO chargé de la gestion d'une décision dans le cadre d'un système de détection de cible. La présence des connecteurs BD permet d'éviter bon nombre de recouvrements d'arcs et d'alléger ainsi le diagramme.

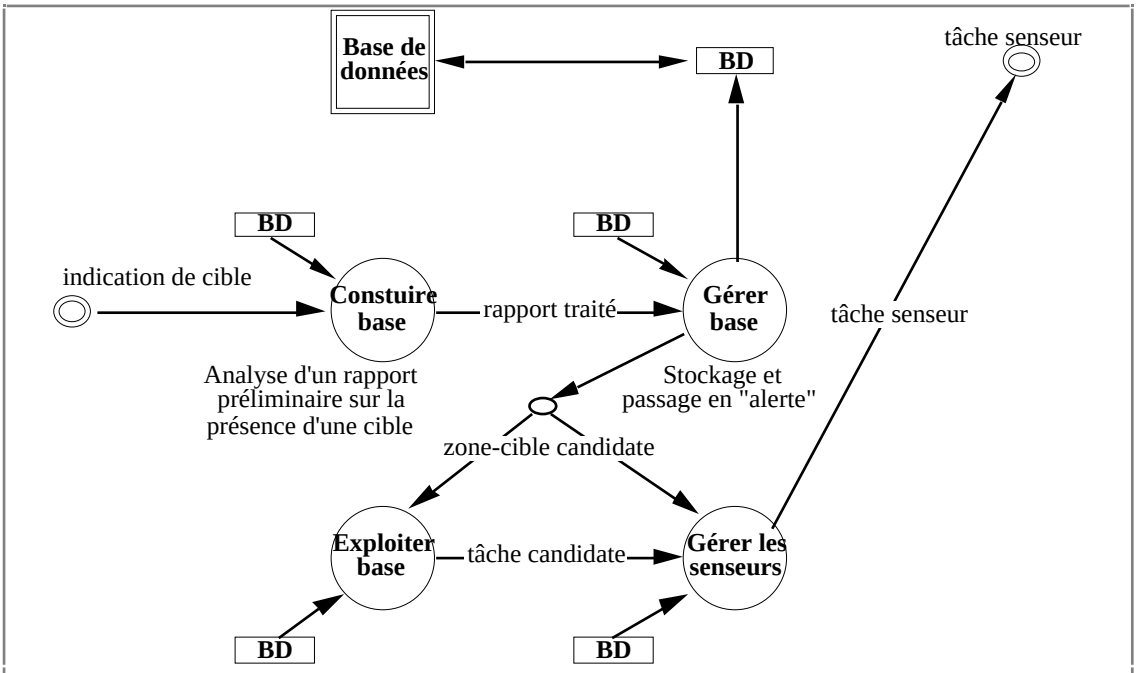


Figure I.9 : Exemple de diagramme PROTO.

2.2.2. Construction et interprétation d'un modèle

A l'aide de l'éditeur graphique de PROTO, le concepteur d'un système réalise une représentation graphique de son système. L'outil permet de raffiner les spécifications fonctionnelles à l'aide de différents diagrammes structurés entre eux de façon hiérarchique : un composant peut être lui-même détaillé sous la forme d'un diagramme.

Une fois le modèle conçu de façon précise, les composants "de bas niveau" doivent être spécifiés dans un langage de commandes spécifique à PROTO. Ce

langage permet de décrire des structures de type DataFlow. Cependant, PROTO autorise également l'utilisation du langage C.

Par ailleurs, un composant peut référencer une fonction PROTO. Ces fonctions sont situées dans une librairie contenant les éléments déjà prédéfinis à l'aide d'un diagramme PROTO. Les librairies contiennent un certain nombre de composants prédéfinis mais peuvent être enrichies par les utilisateurs.

Une fois le modèle spécifié, il peut être exécuté via l'interpréteur PROTO. Il s'agit d'une simulation : les différents composants du diagramme sont mis en évidence lorsqu'ils sont en activité. Le concepteur peut observer son comportement de différentes façons :

- il peut observer le modèle complet : si un composant n'est pas terminal, le diagramme PROTO le représentant apparaît, afin de permettre le suivi de l'exécution au niveau de modélisation le plus fin;
- il peut décider de ne visualiser que certains diagrammes : seuls ces diagrammes seront animés.

Le concepteur peut également placer des points d'arrêt afin de visualiser l'état de son système dans des conditions particulières. Il est ainsi possible de raffiner et déboguer un modèle.

2.2.3. Définition d'une architecture matérielle

L'ensemble des fonctionnalités décrites précédemment a été étendu afin de permettre l'étude d'applications réparties [Burns 91].

Le concepteur d'un système, une fois le diagramme PROTO réalisé, définit une architecture multi-processeurs. Il s'agit, encore une fois, d'une représentation graphique permettant de lier des processeurs (cercles) à des mémoires (carrés) par l'intermédiaire de bus (arêtes). Un exemple d'architecture comprenant 3 processeurs reliés à une mémoire commune est donné en Figure I.10.

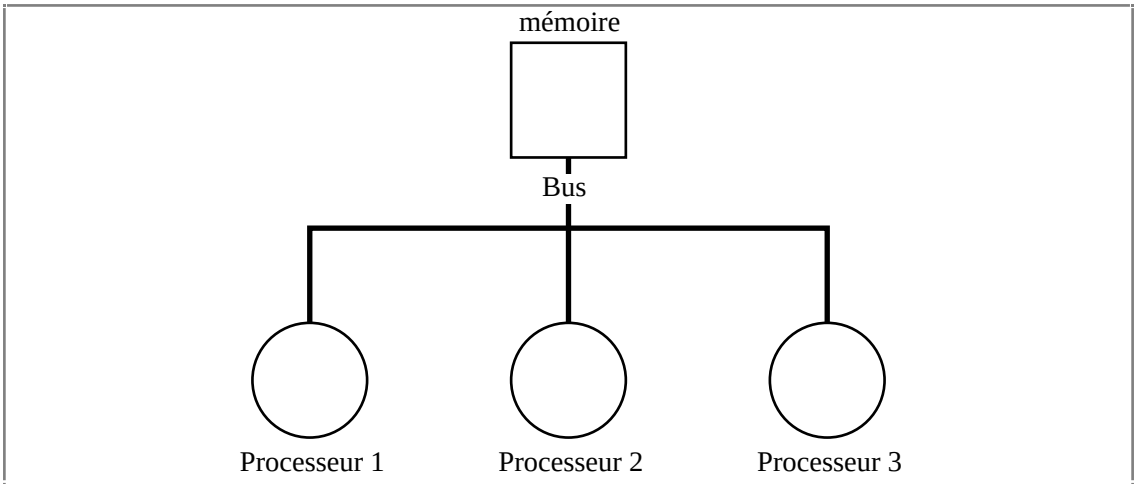


Figure I.10 : Exemple de représentation d'une architecture dans PROTO.

Pour un modèle donné, il est possible de définir plusieurs architectures. Il existe deux architectures prédéfinies :

- celle correspondant à une exécution strictement séquentielle du modèle,
- celle permettant une exécution totalement parallèle du modèle.

L'architecture prédéfinie n'autorisant qu'une exécution séquentielle associe l'ensemble des nœuds-composants à un seul processeur. Les zones de stockage des données sont toutes situées dans une seule mémoire. Il s'agit là de l'architecture par défaut choisie par PROTO si le concepteur n'en spécifie pas une.

L'autre architecture prédéfinie est construite à partir du diagramme PROTO. Chaque nœud-composant est associé à un processeur. Chaque zone de stockage de données est également associée à une mémoire propre. PROTO génère automatiquement les configurations matérielles pour certaines architectures.

2.2.4. Etude d'une application répartie sur une architecture multi-processeurs

Il existe dans PROTO un éditeur permettant de préciser les parties de la spécification dont l'exécution peut se dérouler en parallèle. Il suffit pour cela d'examiner le modèle et de trouver, en se basant sur le paradigme DataFlow, les composants qui n'interagissent pas entre eux. Le concepteur du modèle détermine graphiquement les affectations "zone de stockage de données - mémoire" et "composant - processeur". Une telle opération peut également être réalisée par PROTO, soit sur la base d'une heuristique donnée, soit par affectation aléatoire des unités fonctionnelles.

L'outil permettant d'effectuer cette association se comporte "intelligemment". Si l'affectation d'un composant ou d'une zone de stockage de données a été oubliée, elle sera effectuée au moment de l'exécution du modèle. PROTO signale alors l'association qu'il a choisie. Le concepteur d'un modèle peut associer plusieurs architectures à une spécification ou faire varier le placement des divers objets pour une architecture donnée.

Dans sa version parallèle, PROTO signale les interblocages et donne des informations statistiques sur le fonctionnement des processeurs et les taux d'accès aux différentes mémoires de l'architecture.

2.2.5. Etude critique

Une version de PROTO, appelée PARALLEL-PROTO, est opérationnelle. Le concepteur d'un système peut ainsi faire des observations sur le comportement de son système en fonction des architectures cibles ou du placement des objets pour une architecture donnée.

Le prototype vu par PROTO est une maquette simulée. Rien n'est a priori réutilisable par la suite. Un effort important a été effectué dans le domaine de l'interface avec l'utilisateur. Un utilisateur néophyte peut spécifier rapidement son système, sans avoir à se soucier des détails d'implémentation.

L'autre aspect intéressant concerne l'existence d'un mécanisme de simulation évolué permettant d'effectuer une simulation du modèle défini pour un ensemble d'architectures.

Cependant, le formalisme utilisé n'autorise pas de validation formelle du modèle. Un interblocage existant peut fort bien ne pas apparaître pendant les simulations.

Les outils de mesure par simulation proposés offrent néanmoins des informations intéressantes sur l'impact des choix architecturaux et l'association des objets du modèle aux unités fonctionnelles. L'environnement est suffisamment convivial pour permettre des mesures.

2.3. CONCLUSION

Les deux outils que nous avons décrits correspondent à deux philosophies différentes :

- S.A.O. vise, par raffinements, à la production d'un prototype commercialisable. La "programmation" est effectuée au niveau du formalisme; les versions successives du prototype fournissent des informations permettant son évolution.
- PROTO vise à donner une idée complète du fonctionnement d'un système dans un environnement multi-processeurs. L'utilisateur peut influer sur le modèle de son application comme sur l'architecture, afin qu'ils correspondent l'un à l'autre.

S.A.O. est une excellente illustration de l'approche par raffinements. Il est possible de l'appliquer à des domaines où, traditionnellement, la fiabilité d'un programme et le respect de fortes contraintes temps réel sont un souci majeur. La démarche et l'outil, une fois certifiés, apportent un gain de temps considérable :

- le prototype étant généré automatiquement, il reste une image exacte de la spécification;
- le processus de production de code étant certifié, seule l'intégration du système est à tester.

Les résultats fournis par PROTO sont d'un tout autre ordre : ils concernent aussi bien la structure de l'application que l'architecture sur laquelle elle s'exécute. Ces résultats sont également le fruit d'un processus répétitif. Même si le prototype est de type "maquette non réutilisable", il y a approche par raffinements, en ce qui concerne l'adéquation de l'application à l'architecture qui la supporte.

L'outil de prototypage que nous développons tente de concilier les avantages des deux approches :

- Le prototype produit peut devenir exploitable : cependant, nous ne gérons pas de contraintes de type temps réel.
- Nos efforts portent également sur l'aide au choix d'une architecture. Celles que nous considérons à l'heure actuelle sont relativement simples :

seul le nombre de processeurs est sujet à variation. Nous supposons qu'ils sont tous reliés les uns aux autres.

- Enfin, nous nous efforçons de définir des règles de placement des différents objets logiciels sur les sites d'exécution. L'effort, à l'heure actuelle, porte essentiellement sur la réduction des coûts de communication.

De l'étude de ces deux systèmes, nous avons choisi de retenir, pour définir notre méthodologie :

- La notion de maquette autonome réutilisable : la réalisation d'une maquette soumise à l'environnement dans lequel elle a été créée permet d'obtenir un grand nombre d'informations sur son comportement. Cependant, le passage à la réalisation du système réel est toujours sujet à des problèmes d'interprétation des spécifications.
- L'approche par raffinements : il nous paraît intéressant de travailler au niveau des spécifications, et non, après modélisation et simulation d'un système, au niveau de l'implémentation d'une maquette. Ainsi, la vie du modèle formel et celle du prototype sont parallèles; cela constitue un avantage certain, si le prototype est réutilisable, pour la maintenance et le suivi du système.
- L'association d'un modèle formel à une architecture : elle permet d'étudier l'impact des choix architecturaux et de répartition. Autoriser la génération d'un prototype opérationnel dans de telles conditions constitue un atout important : une même version peut être produite pour plusieurs architectures cibles existantes.

3. Le prototypage à partir de réseaux de Petri

La réalisation d'un système à partir d'un modèle formel est délicate. Le problème le plus crucial réside dans l'interprétation des spécifications ainsi exprimées.

Dans un premier temps, nous définissons les différents problèmes posés par l'implémentation des modèles formels décrivant des applications parallèles : la gestion des conflits et le degré de parallélisme. Nous évoquons ensuite les différentes "familles d'interprétation" d'un réseau de Petri en vue de la réalisation du modèle qui est exprimé.

Ce choix résulte d'un compromis entre la puissance de représentation et la puissance de validation d'un formalisme. Les réseaux de Petri ont un grand pouvoir d'expression mais une modélisation dans ce formalisme cesse rapidement d'être représentable de façon concise.

Avec l'apparition de réseaux de haut niveau [Jensen 81, Chiola 90, Dutheillet 91, Petrucci 91] et la notion de hiérarchie [Hubert 90], il devient possible d'établir des modèles plus concis et plus lisibles. Cependant, certaines représentations graphiques orientées objets restent plus évocatrices.

L'atout fondamental des réseaux de Petri réside dans les possibilités de validation qu'ils offrent. Issues d'une théorie mathématique en continuelle évolution [Couvreur 90a, Couvreur 90b, Murata 90, Chehaibar 91, Chiola 91, Dutheillet 91, Haddad 91, Petrucci 91], des propriétés peuvent fournir des informations sur le comportement réel du système. Par exemple, un interblocage peut ne pas apparaître, même après des heures de simulation. La notion de vivacité [Brams 82] permet d'évaluer de façon certaine les risques de blocage d'un système.

3.1. LES RÉSEAUX DE PETRI

Les réseaux de Petri permettent de spécifier un système parallèle sous la forme d'un ensemble de dépendances reliant des opérations entre elles. Il faut étudier l'ordonnancement des opérations tel qu'il est spécifié dans le modèle.

Le prototype produit doit absolument respecter les contraintes définies dans le modèle, mais l'interprétation d'un modèle mal validé peut aboutir à la réalisation d'un système ne fonctionnant pas.

3.1.1. Les réseaux de Petri Colorés

Les réseaux de Petri Colorés [Jensen 81, Peterson 81, Murata 90], dérivés des réseaux de Petri Ordinaires [Brams 82] sont des graphes bipartis. Les nœuds sont appelés *places* (représentées par des cercles) ou *transitions* (représentées par des rectangles) et sont connectés par des arcs valués. La valuation des arcs en entrée permet d'indiquer les conditions d'*activation* d'une transition alors que la valuation des arcs en sortie détermine l'effet de l'action représentée par la transition.

Définition I.5 : *Réseau de Petri Coloré*

Un réseau coloré est un 5-uplet R = <P,T,C,Pré,Post> où

- P est un ensemble fini dont les éléments sont appelés places ,
- T est un ensemble fini disjoint de P dont les éléments sont appelés transitions ,
- C est une famille indexée par P T d'ensembles C(x); C(x) est appelé domaine de couleurs associé à la place ou à la transition x,
- Pré et Post sont deux familles indexées par PxT d'applications Pré(p,t) et Post(p,t) telles que pour tout (p,t) de PxT, Pré(p,t) et Post(p,t) aient pour domaine C(t) et codomaine Bag(C(p))[1]. Pré et Post sont appelées respectivement fonction d'incidence avant et fonction d'incidence arrière.

Chaque place p contient un ensemble de *marques*. L'ensemble des marques contenues dans l'ensemble des places du modèle définit le *marquage* du modèle.

Définition I.6 : *Marquage d'un réseau de Petri Coloré*

Un marquage M d'un réseau coloré R est une famille indexée par P, telle que M(p) est un élément de Bag(C(p)).

Un réseau coloré marqué est un couple <R,M_0> où R est un réseau de Petri et M_0 est un marquage de R appelé marquage initial.

Une place p est précondition d'une transition t s'il existe un arc orienté de p vers t. Symétriquement, p sera une postcondition de t s'il existe un arc reliant t à p. Le marquage d'un réseau de Petri évolue à chaque *activation* d'une transition. Un tel événement est régi par des *règles de franchissement* : une transition ne peut être activée que si le marquage de l'ensemble des places précondition l'autorise. A l'activation d'une transition, il y a consommation du nombre de marques adéquat dans les places précondition et production de marques dans les places postcondition.

Définition I.7 : *Règles de franchissement d'un réseau de Petri coloré*

Soit *R* un réseau de Petri, *M* un marquage de *R*, on dit qu'une transition *t* de *R* est *franchissable* à partir de *M* pour la couleur *x* de *C(t)* si et seulement si :

- p P, M(p) ≥ Pré (p,t) (*x*)

On note *M [t(x)>* la franchissabilité de *t* à partir de *M* pour la couleur *x*.
Si *t* est franchissable à partir de *M* pour la couleur *x*, le marquage *M'* obtenu par le franchissement de *t* est défini par :

- p P, M'(p) = M (p) - Pré (p,t) (*x*) + Post (p,t) (*x*)

On note *M [t (x)> M'* le franchissement de *t* à partir de *M* pour la couleur *x*.
Si, à partir d'un marquage M, plusieurs transitions sont candidates, n'importe laquelle d'entre elles peut être activée.

---

1 Bag (A) : un multi-ensemble a sur un ensemble non vide A est une fonction a [A → ℕ]. Intuitivement, un multi-ensemble est un ensemble qui peut contenir plusieurs occurrences du même élément. Bag (A) désigne l'ensemble des multi-ensembles finis sur A.

Exemple I.4 : La Figure I.11 illustre les conditions d'activation d'une transition conformément à ce qui est défini dans [Chiola 90, Dutheillet 91]. La transition T possède trois places en précondition (R1, R2, R3) et deux places en postcondition (R4, R5). La seule contrainte sur T est qu'il existe dans R1 et R2 deux marques semblables, ce qui est le cas pour le marquage initial donné. Après activation de T pour x=b, z=b et y=2 (le triplet <b,b,1> était parfaitement envisageable), il y a production d'une marque composée dans R4 et d'une marque simple dans R5.

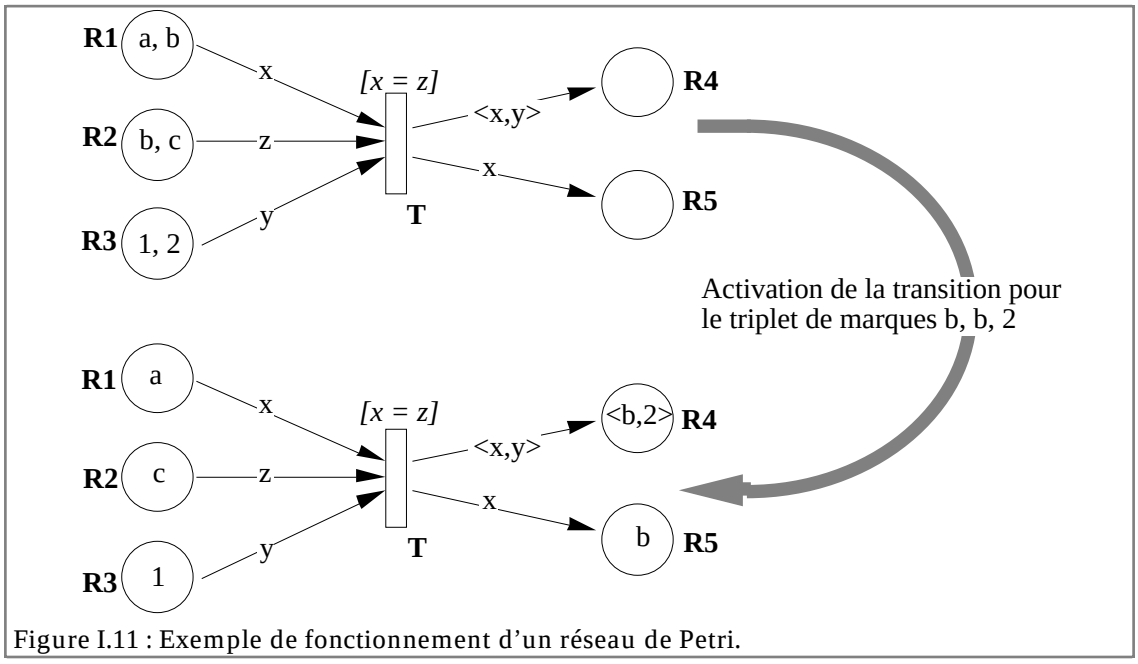


Figure I.11 : Exemple de fonctionnement d'un réseau de Petri.

Les réseaux de Petri permettent de définir un système sous la forme de contraintes entre des actions. En général, les actions sont représentées par les transitions du modèle et sont reliées entre elles par des ressources : les places du modèle.

L'avantage des réseaux de Petri, par rapport à la grande majorité des autres représentations, est leur aptitude à donner des informations concernant l'exécution du système. De la théorie mathématique, on peut déduire un certain nombre de propriétés (invariants, vivacité, existence d'un état d'accueil...) qui sont autant d'informations à la disposition du concepteur pour se faire une idée du comportement de son système.

Par contre, la notion de hiérarchie, toute récente, en est encore à ses débuts. Ainsi, les réseaux de Petri ne permettent pas, à moins d'utiliser des règles bien précises, une validation d'interfaces, dans le cadre d'une conception modulaire.

3.1.2. Les réseaux de Petri Bien Formés

Nous avons choisi les *réseaux Bien Formés*, définis dans [Chiola 90, Dutheillet 91]. Cette classe de réseaux de Petri a une puissance de modélisation équivalente à celle des réseaux de Petri Colorés. Nous rappelons brièvement, dans cette section, les principes de fonctionnement des réseaux bien formés.

Le modèle des réseaux bien formés est construit de manière structurée. Les différents objets composant le système à étudier sont regroupés en classes. Ces classes d'objets peuvent elles-mêmes être associées pour former des domaines de couleurs. Des fonctions opèrent sur ces domaines de couleurs en vue de modifier l'état du réseau. Cette construction est à rapprocher de la construction d'un langage, où des types composés sont construits à partir des types simples, et où les constructeurs de types sont aussi des constructeurs de fonctions.

Classes de couleurs

Les classes d'objets regroupent en ensembles les entités de même nature d'un système. Ces classes peuvent être ordonnées si une telle relation existe entre les objets qui la composent (par exemple, des sites sur un anneau). Habituellement, tous les objets d'une même classe ont le même comportement, tant sur le plan qualitatif que quantitatif. Nous ne considérons que des classes ordonnées.

Nous pouvons définir les fonctions successeur et prédécesseur d'ordre N. Le successeur d'ordre 1 du dernier élément de la classe est défini comme étant le premier élément de cette même classe. Il en est de même avec le prédécesseur du premier élément qui n'est autre que le dernier élément.

Si la classe de couleurs ne possède qu'une seule valeur, appelée "couleur neutre", les marques de ce type sont alors équivalentes à celles rencontrées dans les réseaux ordinaires.

Définition I.8 : *Classes de couleurs*

Soit C l'ensemble des classes d'objets disjointes : $C = \{C_1, \ldots, C_h, C_{h+1}, \ldots, C_n\}$, $1 \leq h \leq n$ avec $i \neq j$, $C_i \cap C_j = \emptyset$.
On distingue dans C deux types de classes d'objets :

- les classes non ordonnées : C_i , $1 < i \leq h$
- les classes ordonnées : C_i , $h < i \leq n$.

Les classes ordonnées sont utilisées pour représenter des compteurs, des sites sur un anneau, etc… Le successeur d'un objet c dans une classe ordonnée est noté c. On peut choisir pour fonction successeur toute fonction vérifiant :

$$c \in C_i, \quad C_i = \left\langle \sum_{k=1}^{|C_i|} {}^{k}c \right\rangle,$$

où ^{k} dénote la k-composition de la fonction , et donc ${}^{k}c$ le $k^{ème}$ successeur de c. L'ordre est circulaire, c'est-à-dire :

$$c \in C_i, \quad {}^{|C_i|}c = c.$$

Si la classe C_i est définie par une liste numérotée des objets qui la composent, c'est cette numérotation qui définit par défaut l'ordre dans la classe C_i .

Domaines de couleurs

Les classes d'objets précédemment définies peuvent être composées par produit cartésien, afin de définir des couleurs qui sont des associations entre objets. Un domaine de couleurs permet de définir des marques composées.

Définition I.9 : *Domaines de couleurs*

Soit I = {1, …, n} l'ensemble des indices des classes d'objets. Soit J = (e_1, …, e_n), un vecteur représentant un élément du multi-ensemble Bag(I) (e_i désigne le nombre d'occurrences de C_i dans le produit cartésien). Un produit cartésien de classes d'objets est noté :

$$C_J = \prod_{i=1}^{n} (C_i)^{e_i}$$

Une couleur étant une association d'objets, C_J définit donc un domaine de couleurs. Par définition, si $J = \vec{0}$, alors $C_J = \{\varepsilon\}$ ou ε est la couleur neutre. Un N-uplet d'objets c_J C_J est noté :

$$c_J = \sum_{i=1}^{n} \sum_{j=1}^{e_i} c_i^j$$

Prédicats

Lorsqu'un prédicat est associé au domaine de couleurs d'une transition, il permet de limiter les différents franchissements possibles. Les prédicats prennent leur valeur dans l'ensemble {vrai, faux}.

Les prédicats que nous autorisons dans un réseau de Petri bien formé sont :

- Vrai
- *f* (x) *op* g (y) : les marques x et y sélectionnées respectent la relation définie par *op*, *f* et *g*.
  - *op* doit être une fonction de comparaison ou d'appartenance : =, ≠, <, ≤, > et ≥.
  - *f* et *g* peuvent être des fonctions successeur, prédécesseur ou constantes.

Il est possible de combiner entre eux ces prédicats, à l'aide des opérateurs , et ¬.

Les fonctions de couleurs

Les fonctions de couleurs d'un réseau Bien Formé sont définies à partir des fonctions identité, des fonctions de diffusion ainsi que des fonctions prédécesseur et successeur.

Les fonctions identité, successeur, prédécesseur et diffusion se définissent sur une classe de couleurs. La fonction de diffusion sélectionne l'ensemble des objets qui forment une classe.

Exemple I.5 : Deux trains circulent dans le même sens sur un tronçon de voie circulaire découpé en sept sections. Il est spécifié que deux trains ne pourront jamais, pour des raisons de sécurité, se situer sur deux segments contigus. Des feux gèrent l'accès à chacune des sections.

Le réseau de Petri de la Figure I.12 modélise ce problème : les sections sont représentées par les places *Section 1* à *Section 7*. La présence d'une marque dans l'une de ces places indique qu'un train s'y trouve. Les feux sont modélisés par les places *F1* à *F7*. La présence d'une marque signifie que le feu est au vert, l'entrée dans la section qu'il garde est alors possible. Le passage d'une section x à une section y s'effectue à l'activation de la transition *x vers y*.

Pour autoriser un suivi détaillé des convois, on souhaite avoir une idée plus précise des informations connues pour chacun des trains. Le concepteur du modèle décide qu'un train est caractérisé par un numéro. En outre, le conducteur doit pouvoir connaître l'identité de la section dans laquelle il se trouve.

L'utilisation de la couleur permet de modéliser l'existence de telles informations : un train est caractérisé par deux variables :

- Son identité,*t*. Comme il n'y a que deux trains, le domaine de *t* est défini par les valeurs 1 et 2.
- Le numéro de la section dans laquelle il se trouve, *s* qui peut prendre des valeurs comprises entre 1 et 7 (il y a 7 sections).

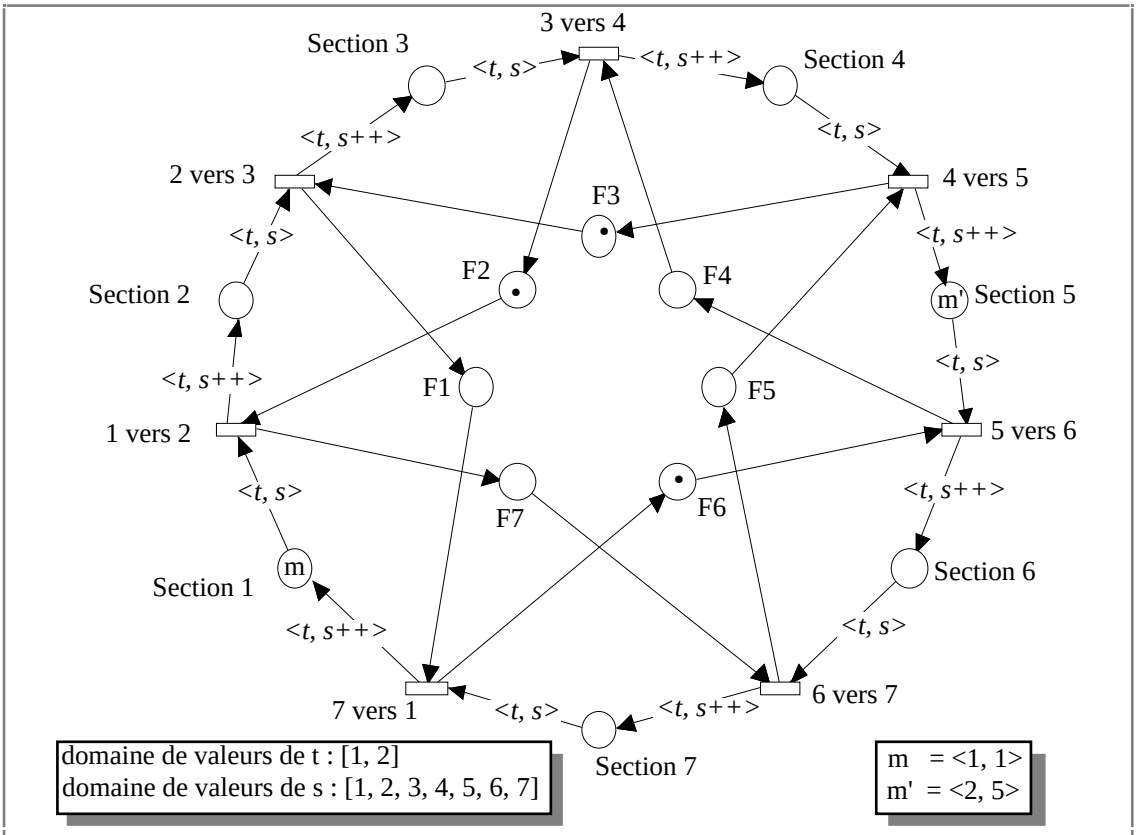


Figure I.12 : Réseau de Petri associé au modèle des trains avec suivi détaillé des informations.

A chaque entrée dans une nouvelle section, le numéro de section est mis à jour. la définition de la fonction successeur, conformément à ce qui est défini dans [Chiola 90, Dutheillet 91] est circulaire, de sorte que 7++ = 1.

La marque m représentant le train numéro 1, initialement situé sur la section 1, vaut <1,1>. De même, m', qui représente le train numéro 2, initialement sur la section numéro 5, possède la valeur <2,5>. Les autres marques du modèle ne sont pas colorées. Seule leur présence a une signification.

3.2. SÉMANTIQUE DES RÉSEAUX DE PETRI

L'utilisation des réseaux Bien Formés dans un outil de prototypage nous semble intéressante. Elle permet au concepteur d'un système de valider son modèle *avant* de créer un prototype. Son interprétation peut alors se baser sur ses propriétés intrinsèques.

La sémantique d'un réseau de Petri doit être respectée dans l'exécution du prototype qui en est issu. Les problèmes se situent au niveau :

- des règles de franchissement des transitions,
- des interblocages entre ressources (places) partagées.

Une interprétation cohérente d'un modèle exprimé à l'aide de réseaux de Petri doit prendre en compte de tels facteurs.

3.2.1. Règle de franchissement des transitions

Conformément aux règles de franchissement d'un réseau de Petri (Définition I.7), si plusieurs transitions sont candidates, n'importe laquelle est "tirée au hasard". Les transitions sont donc franchies les unes après les autres de façon indivisible.

Dans le prototype, les transitions, auxquelles sont associés les traitements, deviennent des *Actions*. L'exécution d'un prototype dans un environnement multi-tâches mono-processeur est triviale puisque seule une Action s'exécute à un instant donné.

Par contre, pour les architectures comportant plusieurs unités fonctionnelles (plusieurs processeurs ou un processeur et des dispositifs matériels externes), il peut y avoir exécution *simultanée* de plusieurs actions. Nous montrons, dans les Chapitres III et IV, les mécanismes permettant de résoudre ce problème.

3.2.2. La gestion des interblocages

Dans un réseau de Petri, le franchissement d'une transition est indivisible. Comme nous nous plaçons dans le contexte d'une implémentation multi-tâches, il est nécessaire de résoudre les problèmes d'accès aux ressources partagées (les places du modèle) entre plusieurs transitions activables pour un marquage donné.

La résolution de ces conflits permet de modifier le marquage de chaque place du modèle de manière indivisible. L'accès (lecture ou écriture) à une place doit être protégé par un *monitor* [Hoare 74]. Toutefois, l'application de cette technique risque de faire apparaître des interblocages.

Les interblocages [Coffman 71] découlent naturellement des contraintes d'intégrité nécessaires aux résolutions des conflits d'accès aux ressources partagées. Une transition doit, pour être franchie, vérifier des conditions de déclenchement, c'est-à-dire obtenir une marque de chaque place liée à un arc d'entrée de la transition. Les accès aux ressources étant bloquants, l'interblocage se produit si par exemple deux transitions, partageant les deux mêmes places, veulent accéder simultanément à la marque que chacune d'elles

contient (Figure I.13). Chacune des transitions accapare une marque et se met en attente (infinie !) de la deuxième marque que l'autre possède.

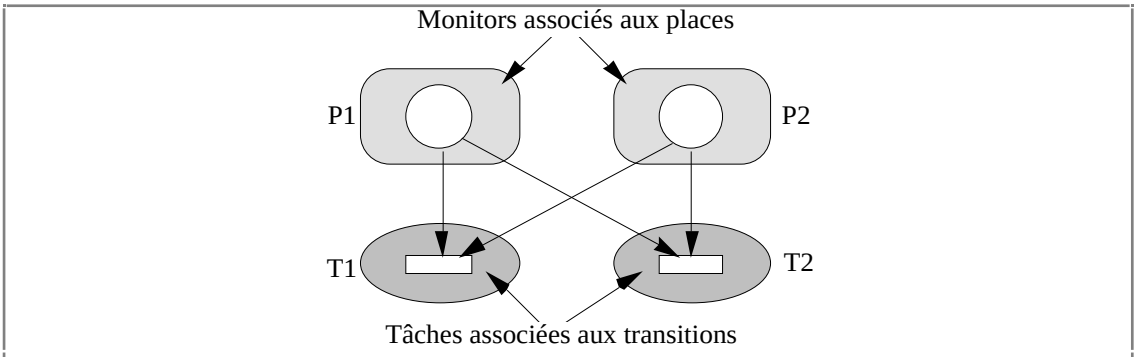


Figure I.13 : Exemple d'interblocage simple entre deux transitions effectuant un accès à des places communes.

Les interblocages nécessitent la mise en œuvre de mécanismes similaires à ceux employés dans le domaine des Systèmes de Bases de Données Réparties [Rosenkrantz 77, Bernstein 81]. Les actions associées au déclenchement d'une transition doivent alors constituer une unité de traitement atomique (une transaction), et des mécanismes tels que l'estampillage [Lamport 78, Bernstein 80] ou le verrouillage à deux phases [Eswaran 76], permettent d'assurer la cohérence nécessaire.

Ces mécanismes permettent de conserver la sémantique de fonctionnement du schéma de contrôle associé au réseau de Petri. Ils ne sont pas prévus pour traiter les interblocages présents dans le modèle qui seront fidèlement reproduits.

3.2.3. Le degré de parallélisme

Les performances des applications complexes sont souvent directement liées au degré de parallélisme.

Le parallélisme du modèle est qualifié de potentiel. Il fait référence aux opérations pouvant comportementalement s'exécuter en parallèle. Naturellement, il tient compte des synchronisations devant être mises en œuvre pour gérer les contraintes de précédence et les conflits d'accès aux ressources partagées.

Le degré de parallélisme d'un support d'implémentation est naturellement limité à celui du modèle. Si le modèle de l'application est purement séquentiel, le support de l'implémentation sera restreint à une seule tâche.

3.3. DIFFÉRENTES EXÉCUTIONS D'UN RÉSEAU DE PETRI

Nous récapitulons, dans le tableau donné ci-après, les caractéristiques des études traitant du prototypage à partir de réseau de Petri dont nous avons eu connaissance.

Jusqu'à présent, les études ont essentiellement porté sur les réseaux de Petri Ordinaires [Brams 82]. Quelques travaux s'appuient sur des formes de réseaux de Petri Colorés [Thuriot 85, Bruno 86].

Référence	Classe de RdP	Implémentation		Techniques		Remarques
		Matérielle	Logicielle	Exécution	Approche	
Silva 79	RdP binaires	mono-processeur	Assembleur	Compilé	Centralisé	• Investigation sélective des transitions.
Silva 80	RdP binaires	mono-processeur	Assembleur	Compilé	Centralisé	• Utilisation d'un scanner spécial dédié à l'investigation des transitions.
Silva 82	RdP binaires	mono-processeur	Assembleur	Interprété	Centralisé	• Comparaison de différentes techniques.
Valette 82	RdP ordinaires	mono-processeur	RT Monitors (tâches en Pascal)	Interprété	Centralisé	• Investigation sélective des transitions.
Nelson 83	RdP ordinaires + arc inhibiteurs	mono-processeur	PL/1 et PL/S	Compilé	Centralisé (remarques pour le décentralisé)	• Investigation sélective des transitions. • Utilisation d'un langage intermédiaire.
Valette 83	RdP binaires	mono-processeur	Assembleur	Interprété	Centralisé	• Investigation sélective des transitions.
Thuriot 85	Une forme de RdP colorés	mono-processeur	RT Monitors (tâches en Pascal)	Interprété	Centralisé	• Le coordinateur (le synchroniseur coloré) est suspendu durant l'exécution des transitions.
Bruno 86	PROT-Nets	mono et multi-processeurs	Ada	Compilé	Partiellement décentralisé	• Deux types de tâches : certaines sont associées à des ensembles de places d'un type donné, les autres aux transitions.
Murata 86	Control-Nets	mono-processeur	PL	Interprété	Centralisé	• Investigation sélective des transitions. • Etude des places précondition et postcondition.
Colom 86	Rdp colorés	mono et multi-processeurs	Ada	Compilé	Centralisé, décentralisé et hybride	• Etude et comparaison des approches centralisées et décentralisées. Introduction de l'approche hybride.
Hauschildt 87	RdP ordinaires	mono-processeur et multi-tâches	C	Compilé	Décentralisé	• Chaque nœud du modèle est une tâche.
Taubner 87	RdP ordinaires	multi-processeurs (transputer)	OCCAM	Compilé	Décentralisé	• Chaque nœud du modèle est une tâche.
Bréant 89, Bréant 90	RdP ordinaires	multi-processeurs (transputer)	OCCAM	Compilé	Hybride	• Décomposition du modèle en processus communicants.
Kordon 89, Kordon 90	RdP ordinaires	mono-processeur	Ada	Compilé	Hybride	• Décomposition du modèle en processus communicants.
El Fallah 89	RdP ordinaires	mono-processeur	Système expert	Interprété	Centralisé	• Le modèle est décrit en une base de règles "moulinées" par un moteur d'inférence.
Paludetto 90, Paludetto 91	RdP ordinaires	mono-processeur	Ada	Compilé	Centralisé	• Développement d'une approche dans la réalisation d'un système complexe.

La plupart des implémentation centralisées proposent des stratégies permettant d'accélérer le choix d'une transition activable parmi les candidates. [Silva 79, Silva 80 et Silva 82] proposent de n'évaluer que le marquage des transitions dont les places précondition sont marquées. Ils associent, à chaque transition, une place précondition la représentant. La précondition d'une transition n'est alors évaluée que si cette place possède le marquage adéquat.

[Murata 86] propose une extension de la méthode d'investigation sélective des transitions d'un modèle en étudiant non seulement les places précondition, mais aussi les places postcondition d'une transition. Cela permet de prévoir

des séquences d'activation et d'effectuer à l'avance la mise à jour de la liste des transitions candidates pour l'étape suivante.

[Valette 83, Thuriot 85] proposent des méthodes centralisées comportant plusieurs tâches (réalisation en assembleur et Pascal). Cependant, lorsque les tâches associées aux transitions sont actives, le coordinateur est suspendu. [Thuriot 85] introduit une notion de couleur : les marques peuvent être modifiées par les tâches qui implémentent les transitions.

Dans un premier temps, les études ont porté sur des classes de réseaux de Petri comportant de fortes restrictions : [Silva 79, Silva 80, Silva 82] s'intéressent aux réseaux de Petri binaires. Les travaux sont ensuite étendus aux réseaux de Petri Ordinaires [Valette 82, Nelson 83, Valette 83, Hauschildt 87, Taubner 87, El Fallah 89, Bréant 90, Kordon 90]. [Murata 86] propose une technique d'implémentation de Control-Nets : une sorte de réseaux de Petri à capacités. [Thuriot 85, Bruno 86, Colom 86] sont les premiers travaux à évoquer l'implémentation de classes de réseaux Colorés.

Les domaines d'applications visés sont essentiellement liés aux ateliers flexibles [Silva 79, Silva 80, Silva 82, Valette 82, Valette 83, Thuriot 85, Murata 86, Colom 86] Les autres études s'intéressent aux structures parallèles utilisables pour implémenter un réseau de Petri [Nelson 83, Hauschildt 87, Taubner 87], ou voient le prototype comme un moyen de valider quantitativement le fonctionnement d'un modèle [El Fallah 89].

L'exécution des opérations est prise en charge par différents supports d'implémentation (tâche…). Pour définir le nombre de supports nécessaires, quelques techniques d'interprétation de modèles spécifiés à l'aide de réseaux de Petri ont déjà été publiées.

Après avoir décrit un modèle qui nous servira pour illustrer les différentes approches, nous allons présenter les principales techniques afin d'en dégager les différentes caractéristiques essentielles [Cousin 88a].

3.3.1. Un exemple de modèle

Nous donnons en Figure I.14 un exemple de modèle dérivé de celui de la Figure I.12, qui nous servira dans la suite de ce chapitre. Ce modèle est exprimé en réseaux de Petri ordinaires [Brams 82]. Structurellement, il est identique au précédent. Seule la sémantique liée aux classes et domaines de couleurs a disparue.

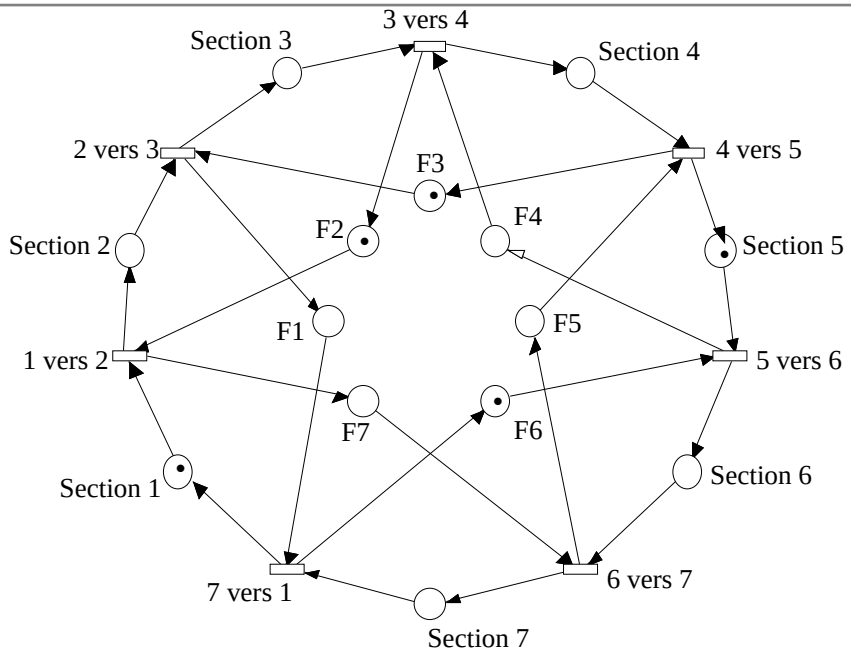


Figure I.14 : Réseau de Petri associé au modèle des trains.

Exemple I.6 : Le problème modélisé est celui décrit dans Exemple I.5. Ici, seule la position des convois (indifférenciés) est représentée.

3.3.2. Exécution centralisée

La première de ces techniques est une technique séquentielle, dans laquelle un *exécuteur* interprète le modèle (Figure I.15). L'exécuteur maîtrise le contexte du modèle (le marquage des ressources) et lance les actions lorsque les préconditions sont vérifiées.

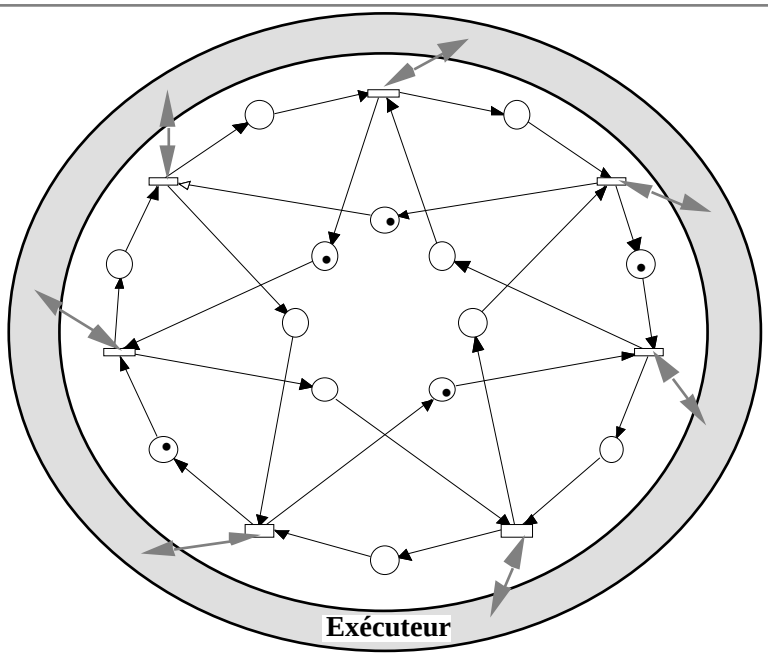


Figure I.15 : Interprétation centralisée d'un modèle.

L'exécuteur étant le seul, pour décider quelle opération doit être effectuée, à accéder à l'ensemble des ressources, les conflits sont trivialement résolus.

L'exécuteur peut effectuer lui-même les travaux associés à chaque transition du modèle. Il exécute ces procédures et modifie le marquage courant de l'application à la fin de chacune d'elles (Figure I.16). Pour ce faire, il doit :

- définir l'ensemble des transitions activables pour le marquage courant,
- choisir une transition parmi l'ensemble calculé,
- effectuer les actions qui lui sont associées (appel d'une procédure par exemple),
- mettre à jour le marquage courant.

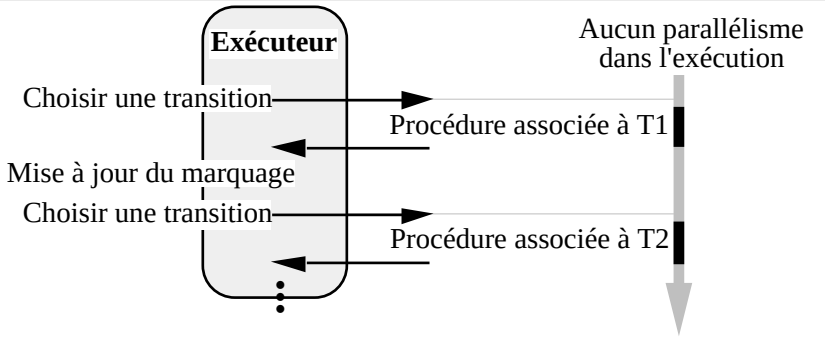


Figure I.16 : Cas où l'exécuteur effectue lui-même les tâches associées à chaque transition.

Cela revient à simuler l'exécution du modèle. L'*exécuteur* n'est, dans ce cas, qu'un interpréteur. Une illustration de cette technique d'implémentation, à l'aide d'un système expert, est décrite dans [El Fallah 89]. D'autres implémentations de ce type, en assembleur [Silva 79, Silva 80, Silva 82, Valette 83], en PL/I [Nelson 83] et en langage Ada [Colom 86], sont également proposées.

Si elle présente l'avantage d'être extrêmement simple et générique, cette technique d'interprétation du modèle, et le prototype qui en découle, possèdent les inconvénients suivants :

- L'exécution séquentielle n'autorise aucun parallélisme. Il est donc inutile d'utiliser des réseaux de Petri pour spécifier un tel problème. Il s'agit bien là de simulation et non de prototypage.
- L'*exécuteur* est un goulot d'étranglement important. Les performances du prototype sont directement liées à ses capacités dans le calcul de l'ensemble des transitions activables après modification du marquage courant.

Une telle politique se justifie si l'on souhaite avoir un point de vue centralisé du modèle.

3.3.3. Exécution complètement décentralisée

La deuxième interprétation d'un réseau de Petri est située à l'autre extrémité du spectre des interprétations possibles. Elle consiste à associer une tâche à chaque objet (place et/ou transition) du modèle. C'est le support d'implémentation maximal possible.

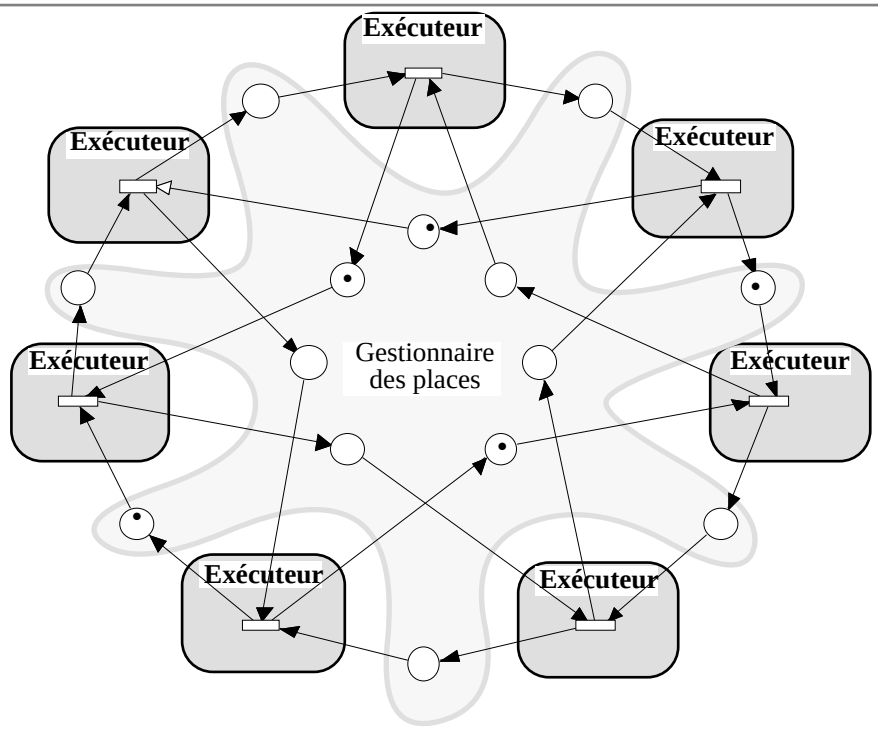


Figure I.17 : Interprétation complètement décentralisée.

Considérons, dans un premier temps, que des tâches soient associées aux seules transitions du modèle (Figure I.17). Ces exécuteurs de transition doivent coopérer afin de gérer le contexte (marquage) du modèle. Il est possible de confier la gestion de ce contexte à une tâche unique : le *gestionnaire des places*. Chacune des tâches associées aux transitions attend un ordre du gestionnaire des places, exécute les actions associées à la transition, rend compte de la fin de l'exécution, puis se met en attente d'un nouvel ordre. Le gestionnaire des places peut donc lancer une autre action sans attendre la fin de l'exécution de celle qu'il vient de lancer.

Cette méthode autorise un parallélisme et peut aisément être déduite de la méthode centralisée décrite dans [Colom 86]. Cependant, le gestionnaire des places constitue un goulot d'étranglement important, et cela même si des Actions accèdent à des places différentes.

Cette technique convient parfaitement à des solutions centralisées ou séquentielles. Son avantage primordial réside dans la simplicité de sa mise en œuvre.

Considérons désormais que la gestion des places est confiée à plusieurs tâches. Il est alors possible d'atteindre le parallélisme potentiel induit par le schéma de contrôle, et donc de bénéficier pleinement de la puissance. Cependant, la résolution des problèmes d'accès aux ressources partagées (les places du modèle) entre plusieurs exécuteurs potentiellement actifs simultanément n'est pas triviale. Le nombre de conflits, directement dépendant du nombre de tâches composant le prototype, est ici maximal.

[Haushildt 87] propose que des tâches (une par place) gèrent le marquage du modèle. L'exécuteur d'une transition effectue des demandes aux tâches gérant

les places auxquelles il est associé en entrée et attend qu'elles lui aient toutes notifié un accord. Lorsque les actions associées à la transition sont réalisées, il signale aux tâches gérant les places qui sont en sortie une modification de leur marquage.

L'implémentation proposée dans [Haushildt 87] concerne les réseaux de Petri non colorés, et les protocoles de communication entre les différentes tâches du prototype sont excessivement complexes. Le nombre de tâches étant directement proportionnel au nombre de sommets du modèle, leur gestion atteint un coût rapidement prohibitif.

Une autre implémentation de ce type, basée sur le langage OCCAM, est proposée dans [Taubner 87]. Chaque nœud du modèle (les places et les transitions) devient une tâche OCCAM. Le flot de messages échangés entre les différentes tâches est, là encore, très important.

En conclusion, si cette interprétation semble donner un prototype dont le parallélisme est maximal, elle comporte les inconvénients suivants :

- La gestion des places par une seule tâche (une sorte d'exécuteur) n'engendre pas de conflits d'accès. Par contre, la complexité de mise en œuvre d'une gestion parallèle des places est phénoménale, en particulier dans le domaine des réseaux de Petri Colorés pour lesquels la notion d'activabilité d'une transition est sémantiquement beaucoup plus complexe. Les protocoles de communication entre ces tâches sont facilement très complexes, surtout lorsque des places sont précondition de plusieurs transitions. Le coût d'un tel protocole nuit aux performances apparemment introduites avec le parallélisme.
- Le nombre de tâches devient rapidement trop important. Le coût induit par leur gestion devient prohibitif.

3.4. CONCLUSION

Les deux catégories d'interprétation (centralisée et complètement décentralisée), aboutissent à des prototypes pathologiques. Elles ne se justifient que dans des cas assez particuliers :

- la première si le modèle est quasiment séquentiel, ou dans le cadre d'une simulation.
- la seconde si le parallélisme effectif du modèle est élevé (i.e. il y a peu de conflits sur les places du modèle) et si l'architecture cible est hautement parallèle (transputers...).

Les méthodes exposées comportent cependant des avantages :

- elles sont simples et systématiques,
- elles s'appliquent à tous les modèles possibles pour la classe de réseaux de Petri considérée.

En fait, les études présentées ici considèrent des degrés de parallélisme extrêmes. La granularité du parallélisme d'un réseau de Petri est rarement située au niveau d'une transition. Ainsi, si l'on souhaite obtenir un prototype

plus performant, il faut analyser le modèle et en extraire des propriétés comportementales. Des techniques d'implémentation basées sur ce principe sont exposées dans la section suivante.

4. Degré de parallélisme d'un système modélisé en RÉSEAU DE PETRI

Dans un modèle complexe, le degré de parallélisme maximum est indépendant du nombre de transitions. Il faut trouver des sous-ensembles de transitions pour lesquelles aucun parallélisme n'est possible : les *Ensembles séquentiels*.

Définition I.10 : *Ensemble séquentiel*

Ensemble de transitions au sein duquel aucun parallélisme n'est possible.

Exemple I.7 : Nous donnons en Figure I.18 une interprétation possible du modèle de la Figure I.14. Cette interprétation définit trois ensembles de transitions dont les exécutions ne peuvent être simultanées :

- E1 contenant les transitions "1 vers 2", "2 vers 3" et "3 vers 4",
- E2 contenant les transitions "4 vers 5", "5 vers 6" et "6 vers 7".
- E3 contenant les transitions "1 vers 2", "7 vers 1" et "6 vers 7",

Il est ainsi possible de définir trois *Ensembles séquentiels*. Chacun d'eux communique avec ses homologues. Un mécanisme assure la gestion de ces communications. Il doit permettre des communications de type synchrone (rendez-vous) ou asynchrone (messages).

Dans l'exemple donné, E3 communique de façon synchrone (transitions "1 vers 2" et "6 vers 7") mais aussi de façon asynchrone (places F1 et F6) avec ses deux voisins. E1 et E2 n'ont que des échanges de type asynchrone (places F3 et F4 et Section 4).

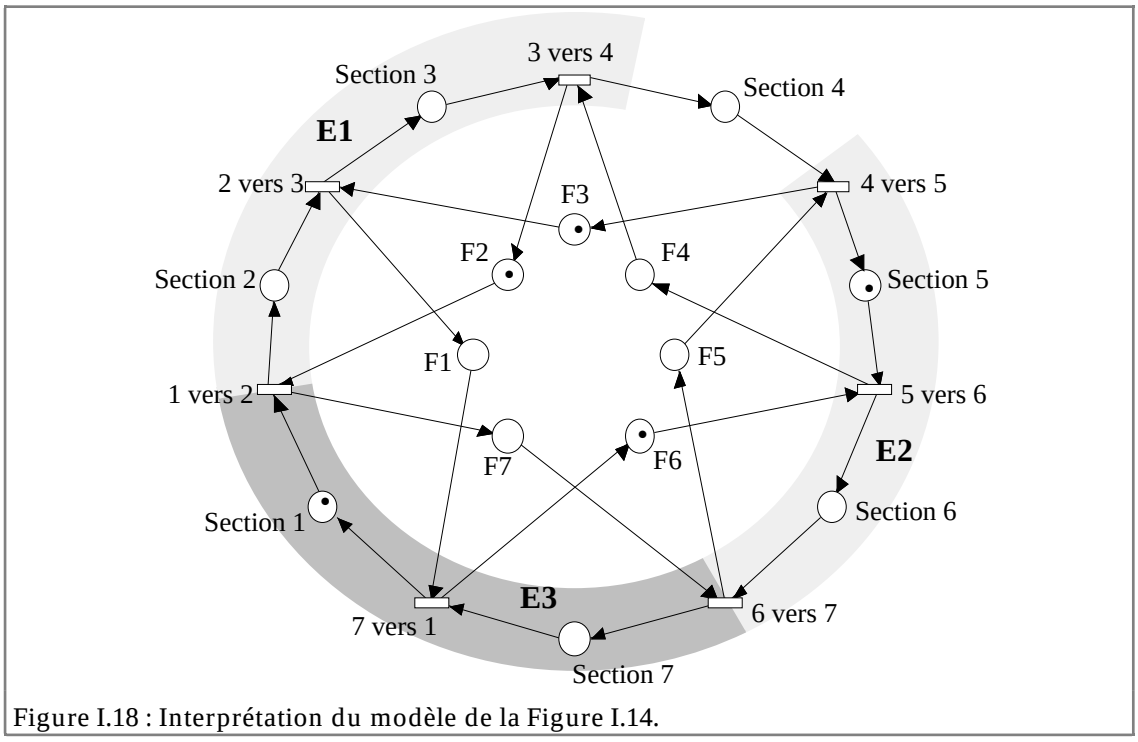


Figure I.18 : Interprétation du modèle de la Figure I.14.

La calcul des *Ensembles séquentiels* d'un modèle équivaut à évaluer un degré de parallélisme. Un tel résultat dépend de l'*interprétation* du modèle. Nous verrons plus loin que, pour l'exemple de la Figure I.14, il en existe de

nombreuses. Il faut donc posséder des informations d'ordre *sémantique* afin de calculer les processus. Pour cela, il faut analyser le modèle.

L'avantage indéniable de ce type d'interprétation est de permettre la création d'un prototype au parallélisme adéquat. Les conflits d'accès sont ramenés à une proportion raisonnable. Il ne faut cependant pas les ignorer au moment des choix de réalisation.

Cependant, il faut partitionner le modèle. Quelques travaux ont déjà été menés dans ce domaine. Nous évoquons les plus importants d'entre eux.

4.1. LES TRAVAUX MENÉS À L'UNIVERSITÉ DE SARAGOSSE

Les premiers travaux, à l'université de Saragosse, ont commencé dès la fin des années 1970 [Silva 79, Silva 80, Silva 82, Colom 86] dans le cadre des systèmes de production en automatique. Les auteurs se sont, dans un premier temps, intéressés au prototypage de réseaux de Petri en assembleur [Silva 79, Silva 80, Silva 82]. Le prototype était centralisé; ces travaux diffèrent par les stratégies de choix d'une transition à activer.

Dans une étude plus récente [Colom 86], le partitionnement d'un modèle en ensembles est évoqué comme alternative à une solution centralisée. Leur étude porte sur les "1-firable binary P / T nets".

Les auteurs définissent des *conflits structurels*. Il y a conflit structurel entre deux transitions lorsqu'elles partagent des places en entrée.

La résolution des conflits structurels repose sur une transformation du modèle permettant d'obtenir un réseau de Petri équivalent, c'est-à-dire ayant les mêmes séquences de franchissement. Le modèle ainsi obtenu peut alors être décomposé en ensembles séquentiels connectés par des mécanismes de communication.

La transformation du modèle initial en un modèle décomposable repose sur l'adjonction de places implicites. Cette décomposition repose sur :

- La description fonctionnelle du système (le modèle).
- Les spécifications d'implémentation : elles sont explicites si le concepteur du modèle le partitionne lui-même. S'il se contente d'indiquer quelles propriétés il souhaite mettre en valeur (minimisation des processus séquentiels, minimisation des communications synchrones ou asynchrones...), elles seront implicites.

Nous donnons en Figure I.19 un exemple de transformation en vue d'obtenir un modèle décomposable. Les modèles (a) et (b) sont équivalents (ils ont le même langage associé : $L(M_a, M0_a) = L(M_b, M0_b)$). La décomposition présentée ici ne comporte aucune communication asynchrone; le prototype associé est composé de deux tâches : E1 et E2 qui communiquent via des *buffers* (p1, p2, p3 et p4). Les auteurs évoquent également un autre modèle comportant ce type de communication.

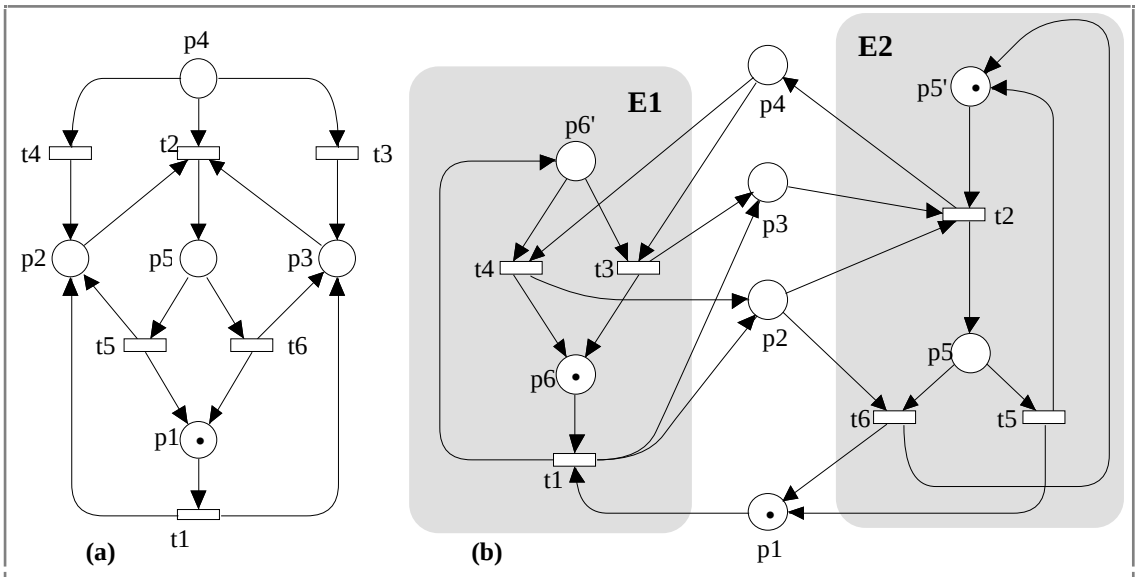


Figure I.19 : Transformation d'un modèle d'après la technique donnée dans [Colom 86].

Pour certains modèles (Figure I.20), la décomposition ne peut cependant pas être réalisée. Il s'agit des modèles comportant des *conflits effectifs*. Deux transitions en conflit structurel qui, pour un marquage donné, sont toutes deux activables, sont en conflit effectif.

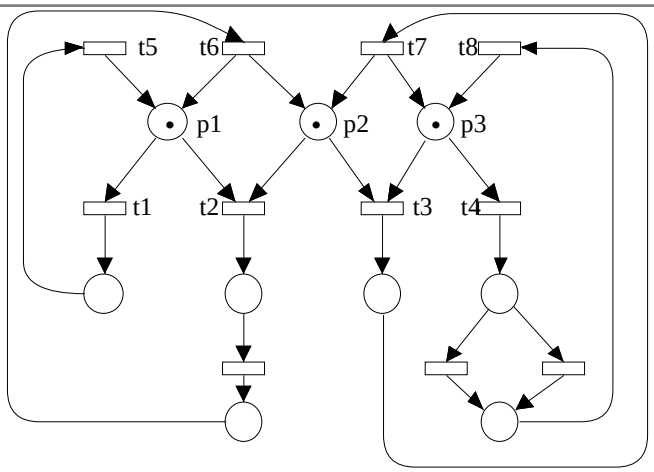


Figure I.20 : Modèle pour lequel la décomposition pose des problèmes.

Dans le modèle de la Figure I.20, t1 et t2 sont déjà en conflit structurel (partage de la place p1). Le conflit devient effectif car l'activation de t6 permet à la fois l'activation de l'une ou de l'autre, mais pas des deux à la fois. Il en est de même pour t3 et t4. Par contre, le conflit structurel entre t2 et t3 n'est pas effectif.

Pour résoudre les conflits effectifs, il faut superposer au mécanisme de communication global à l'ensemble du modèle des gestionnaires spécialisés : les *synchroniseurs locaux*. La décomposition du modèle de la Figure I.20 est donnée en Figure I.21.

Le prototype associé à un tel modèle sera implémenté sous la forme de trois tâches : E1, E2 et le synchroniseur local.

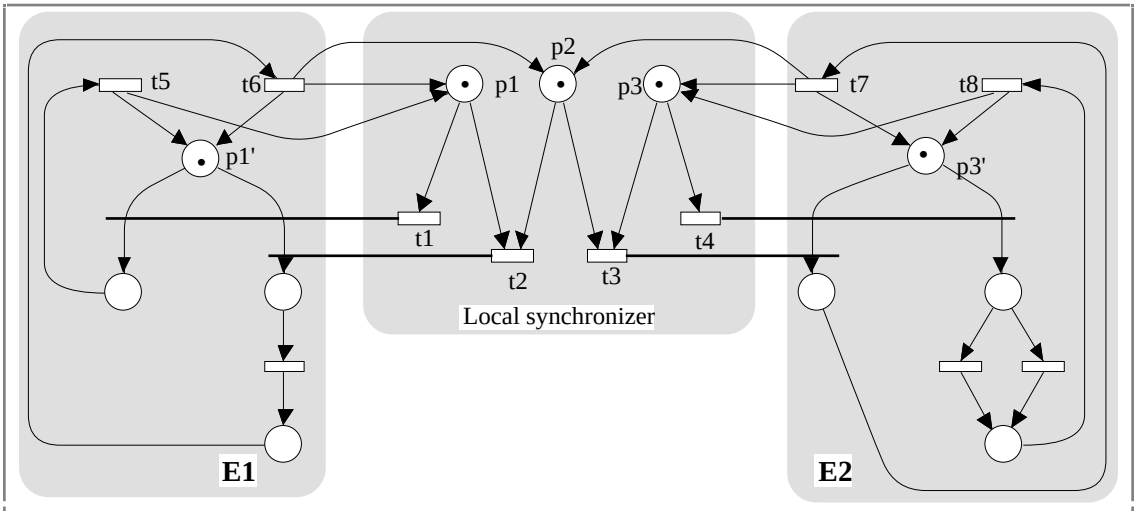


Figure I.21 : Décomposition proposée par [Colom 86] pour le modèle de la Figure I.20.

L'intérêt principal des travaux menés à l'université de Saragosse réside en deux points :

- après avoir clairement évalué les limites du modèle centralisé, leurs auteurs se sont attachés à définir des ensembles séquentiels dont l'exécution peut être concurrente;
- ils ont caractérisé la notion de conflits structurels et effectifs et ont proposé une solution en vue de les réduire à l'aide des synchroniseurs locaux.

Cependant, les entités correspondant, dans le prototype, à ces synchroniseurs locaux, deviennent rapidement complexes. Cela est dû au fait que, dans leur travaux, les auteurs ont refusé de restreindre la classe de modèles pour lesquels leur méthode est applicable.

4.2. LES TRAVAUX MENÉS À L'UNIVERSITÉ DE TOULOUSE

D'autres travaux sont en cours au laboratoire LAAS sur la méthode HOOD/PNO [Paludetto 90, Paludetto 91]. Pour effectuer la décomposition du réseau de Petri, les auteurs s'appuient sur une représentation HOOD [HOOD 89]. L'étude n'a pas encore donné lieu au développement d'un outil : l'objectif étant, dans un premier temps, de définir une méthode de conception d'applications pour la commande de procédés industriels.

L'idée de base est d'utiliser les avantages de deux formalismes : HOOD pour la décomposition en objets, les réseaux de Petri pour représenter le comportement du système et obtenir ses propriétés comportementales.

La démarche comporte cinq étapes :

- décrire le système et son environnement,
- déterminer les objets physiques du procédé,
- rattacher chaque objet physique à sa classe d'objets,
- décrire chaque classe physique,
- déterminer les classes et les objets logiciels.

Les concepts de HOOD sont utilisés en vue de décomposer le problème en objets qui doivent correspondre, si possible, à des entités physiques. La méthode est itérative : un objet complexe peut être décomposé en plusieurs objets de niveau inférieur. Lorsque le concepteur est arrivé au niveau de description le plus bas, les objets sont tous physiques; c'est d'ailleurs un critère d'arrêt d'une décomposition.

Le comportement des objets ainsi déterminés est alors modélisé à l'aide de réseaux de Pétri ordinaires. Chaque réseau de Petri est contenu dans un module. Les relations entre modules sont définies à l'aide de places : il est possible de représenter de la sorte différents types de communication :

- strictement et faiblement synchrone,
- asynchrone,
- requêtes à attente limitée.

Sur la base de la décomposition définie à l'aide de HOOD, et des comportements des différents composants exprimés à l'aide de réseaux de Petri, il est possible de générer un squelette d'application.

Des contraintes temps réel peuvent être introduites dans la description HOOD. Elles seront intégrées sous la forme de requêtes à attente limitée.

L'approche permet de faciliter l'analyse du modèle. Il n'y a pas d'ambiguïté puisque c'est le concepteur du système qui donne ses directives via HOOD.

Cependant, il travaille avec deux formalismes différents. Il est donc impossible de cacher les réseaux de Petri. Cet inconvénient est tempéré par leur petite taille. Le modèle de l'ensemble du système est important mais il a été obtenu par étapes.

La validation du système peut être effectuée à l'aide des réseaux de Petri. Elle fournit également des indications dans le processus de décomposition. Si le réseau de Petri décrivant le comportement d'un objet ne comporte qu'un seul semi-flot (Définition II.9), il peut être considéré comme terminal.

L'apport principal de cette méthode réside en deux points :

- Les réseaux de Petri construits possèdent des caractéristiques précises puisqu'ils sont conçus au fur et à mesure, en s'appuyant sur la méthode HOOD. Un certain nombre de cas d'école peu significatifs peuvent ainsi être ignorés puisque les modèles ont une structure correspondant à des critères précis.
- La notion de semi-flots, et d'ensemble séquentiel, est caractérisée de façon explicite.

4.3. LES VERSIONS D'ÉTUDE DU GÉNÉRATEUR DE CODE ADA

Les travaux que nous menons sur le prototypage ont commencé en 1987. Les objectifs que nous nous sommes donnés sont différents de ceux envisagés à l'époque dans [Colom 86] puis, plus tard, dans [Paludetto 90] :

- nous nous intéressions au prototypage de systèmes répartis;

- nous avons volontairement choisi de restreindre les classes de modèles sur lesquelles la méthode était applicable [Cousin 88b] en vue d'étudier une généralisation du processus pour différents langages.

Dans ce but, nous avons développé deux versions d'étude du générateur de code.

La première d'entre elles produit un programme Ada, à partir d'un modèle exprimé dans une sous-classe des réseaux de Petri ordinaires [Cousin 88b, Kordon 89, Kordon 90]. Il s'agissait dans un premier temps de valider nos stratégies de décomposition hybrides, déduites des travaux présentés dans [Colom 86].

Par la suite, un deuxième générateur de code a été développé : il s'agissait de produire un programme OCCAM pour un ensemble de transputers [Bréant 90].

Les leçons que nous avons tirées de ces deux expériences sont multiples :

- le processus est automatisable à condition de pouvoir extraire du modèle des informations d'un niveau sémantique élevé;
- la génération de code, en Ada et en OCCAM, posent des problèmes complètement différents, bien que les deux langages manipulent tous deux le parallélisme;
- il est possible de se référer à une analyse commune : la décomposition du prototype en ensembles séquentiels communicants.

Les versions d'étude du générateur de code développées au MASI se basent également sur une décomposition du modèle. Cette décomposition est indiquée par l'utilisateur.

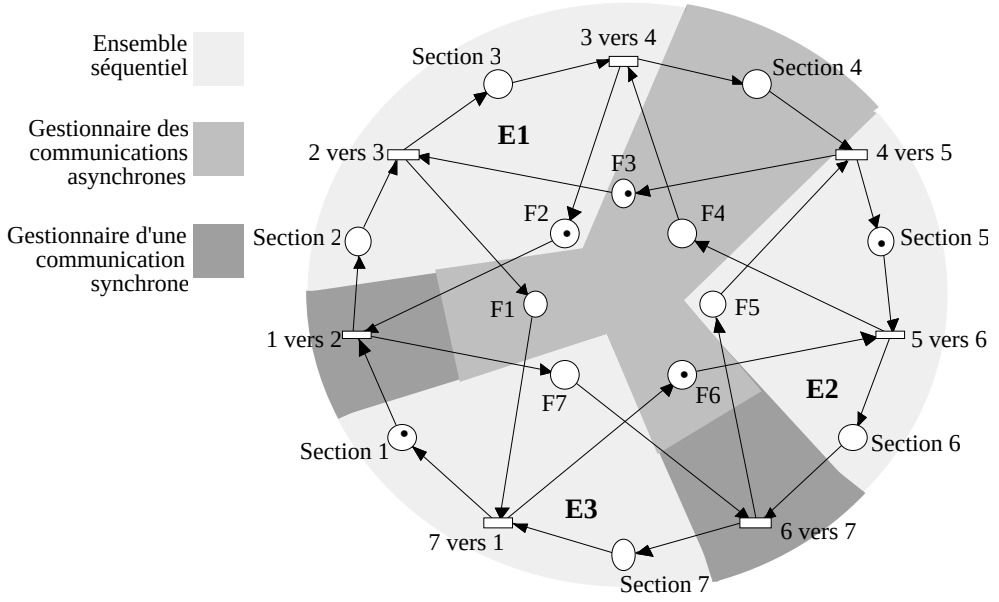


Figure I.22 : Organisation du prototype dans la version préliminaire du générateur de programme Ada.

Le prototype comporte deux types de gestionnaires de communications : ceux qui prennent en charge les communications asynchrones et ceux qui prennent en charge les communications synchrones.

Les choix d'implémentation sont directement liés au langage cible. Dans le prototype Ada [Kordon 89, Kordon 90], la gestion des communications asynchrones est assurée par une seule tâche, tandis qu'en OCCAM [Bréant 89, Bréant 90], chaque lien (place) est géré par une tâche dédiée. Dans les deux cas (OCCAM et Ada), chaque communication synchrone est gérée par un serveur.

Exemple I.8 : Nous donnons en Figures I.22 et I.23 deux types d'organisation issus de la décomposition de la Figure I.18 indiquée dans Exemple I.7. La seule différence réside dans la gestion des communications asynchrones. Le choix de l'une ou l'autre politique est étroitement lié au langage et à la configuration architecturale de la machine cible (station de travail en Ada, ensemble de Transputers pour OCCAM).

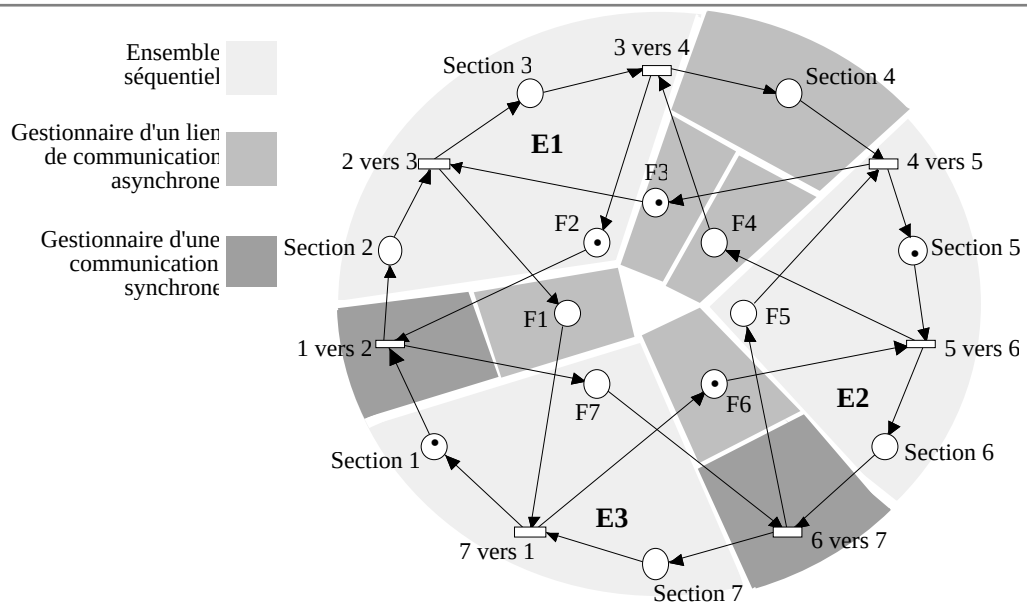


Figure I.23 : Organisation du prototype OCCAM pour la version OCCAM du générateur de code [Bréant 90].

Ces versions préliminaires sont capables de générer un prototype à partir de réseaux de Petri ordinaires. Cependant, seule une classe précise de réseaux de Petri peut être traitée de la sorte :

- Les modèles doivent pouvoir se décomposer en machines à états communicantes. Les communications peuvent être synchrones ou asynchrones;
- Il doit y avoir une marque et une seule dans les places support de l'ensemble séquentiel;
- Dans la version Ada, les ensembles séquentiels doivent avoir un comportement cyclique;

En outre, le concepteur du modèle doit indiquer lui-même le partitionnement du modèle.

Ces fortes restrictions se justifient par le fait qu'il ne s'agissait que de versions d'étude (des prototypes!), destinées à mettre en évidence les problèmes posés,

tant du point de vue du procédé que du point de vue de la structure du prototype.

Les travaux sur architecture de type Transputer sont poursuivis par F. Bréant avec la réalisation d'un nouveau générateur de code en "Concurent C"; l'analyse de la version du générateur Ada a abouti aux travaux présentés dans ce document.

Ces premières études nous ont permis de démontrer la faisabilité du problème, sur la base d'hypothèses dont les plus restrictives ont progressivement disparu. Elles ont également mis en évidence les problèmes liés à la génération, nous obligeant à redéfinir le processus afin de profiter des résultats réutilisables.

4.4. CONCLUSION

L'interprétation d'un réseau de Petri en terme d'ensembles séquentiels permet d'obtenir un prototype plus simple. Son élaboration se divise en deux étapes :

- il faut analyser le modèle afin d'extraire les ensembles séquentiels;
- il faut trouver au prototype une structure cohérente en fonction de la compréhension que l'on en a.

Les trois études présentées dans cette section proposent des techniques de compréhension différentes d'un modèle, toutes basées sur un partitionnement du réseau de Petri en ensembles séquentiels préfigurant les différentes tâches du prototype.

La décomposition peut s'effectuer uniquement à partir des propriétés du réseau de Petri [Colom 86, Bréant 90, Kordon 90]. Dans [Paludetto 91], le modèle est associé à une autre représentation; la décomposition utilise également les informations qu'elle fournit.

Considérant la complexité liée à l'interprétation des réseaux de Petri, il est intéressant de faire des hypothèses sur la façon dont sont formulés les modèles.

Le respect de telles hypothèses peut être réalisé de deux manières :

- En s'appuyant, durant la phase de conception, sur une méthode de spécification de haut niveau : c'est le cas, avec HOOD, dans [Paludetto 91].
- En envisageant la traduction automatique de spécifications exprimées dans un formalisme de plus haut niveau. Les réseaux de Petri sont alors invisibles pour le concepteur du modèle [Estraillier 91].

Dans un cas comme dans l'autre, les modèles obtenus possèdent des structures types. Certains cas d'école, difficiles à décomposer, peuvent donc être écartés au profit d'une plus grande efficacité du prototype obtenu.

5. Conclusion

L'étude des travaux effectués, dans le domaine du prototypage de systèmes parallèles, nous a permis de dégager les idées suivantes :

- Le modèle doit permettre une validation formelle du système : les réseaux de Petri constituent un formalisme de spécification intéressant [Ayache 85, Tu 90, Chehaibar 90, Vinci 90, Moitessier 91, Shapiro 91].
- La réalisation d'un système, sur la base de spécifications, aussi formelles soient-elles, pose des problèmes d'interprétation et de compréhension complexes [Murphy 89]. Aussi est-il intéressant d'automatiser le processus.
- Rendre le prototype autonome, par rapport à l'environnement dans lequel il a été créé, constitue un atout majeur. De simple maquette jetable, il peut devenir une base pour une première version d'un système. Si le prototype est efficacement généré, les évolutions futures peuvent être effectuées sur la base des spécifications, comme dans [Cassigneul 91].

5.1. L'APPORT DES RÉSEAUX DE PETRI

Les réseaux de Petri sont actuellement un des rares modèles formels, avec les spécifications algébriques [Gaudel 91] à pouvoir offrir des résultats concernant le comportement d'un modèle. En effet, la seule façon d'étudier le comportement d'un système, pour bien des formalismes, réside en de longues phases de simulation. Ces simulations, aussi complètes soient-elles, peuvent ne pas mettre en évidence l'ensemble des défauts du système.

Il est cependant indéniable que les réseaux de Petri ne sont pas aussi explicites que d'autres formes de représentation d'un système, surtout lorsque la taille du modèle devient importante. L'étude des réseaux de Petri hiérarchiques résout le problème [Hubert 90]. Il est également possible de les considérer comme une représentation intermédiaire obtenue automatiquement depuis un formalisme de plus haut niveau. Les avantages sont les suivants :

- à partir de cette représentation interne, on peut déduire des propriétés avec plus de fiabilité que par de simples simulations effectuées à l'aide du formalisme d'origine;
- la structure des réseaux de Petri devient "typique"; leur interprétation en est facilitée d'autant.

5.2. AUTOMATISATION DU PROTOTYPAGE

Les modèles de type "transition-état" [Arnold 92] sont fréquemment utilisés dans la conception d'applications industrielles. Baser un outil de prototypage, ou de génération de code, sur une représentation de ce type est donc intéressant.

L'automatisation du prototypage permet d'éviter la dérive observée entre la spécification d'un système et le prototype associé. L'expérience donnée par le

projet S.A.O. prouve que, si le générateur de code est certifié, le code qu'il génère l'est également.

Un tel outil doit aussi permettre au concepteur d'un système de procéder par raffinements successifs. C'est l'enseignement tiré de l'étude des méthodes de conception utilisées dans la réalisation de projets de grande taille. L'utilisateur doit pouvoir intervenir sur un grand nombre de points afin d'obtenir un prototype réaliste, et non une maquette dont l'interprétation serait sujette à caution.

L'intégration avec un environnement extérieur doit également constituer une préoccupation majeure, qu'il s'agisse de l'interfaçage avec un environnement déjà existant ou de l'implantation d'un système logiciel sur une architecture donnée.

Les techniques d'implémentation des réseaux de Petri reposent sur une observation des modèles afin de choisir des caractéristiques d'implémentation. Ces caractéristiques peuvent être déduites à l'aide d'indications supplémentaires fournies par le concepteur du système [Paludetto 90, Paludetto 91] ou de l'analyse des propriétés du modèle [Cousin 88b]. Les travaux exposés dans ce document appartiennent à la seconde catégorie : les réseaux de Petri sont interprétés sur la base des propriétés résultant de leur validation [Kordon 91b].

Cependant, l'analyse du modèle fait appel à de nombreux éléments qui ne sont plus forcément d'ordre structurel. Il faut donc pouvoir disposer de moyens d'analyse plus poussés. Il risque d'exister des classes de modèles pour lesquelles un outil automatique ne disposerait pas de suffisamment d'informations pour discerner les entités logicielles.

5.3. DU PROTOTYPAGE À LA GÉNÉRATION DE CODE OPTIMISÉ

Dans le domaine du logiciel, ramener le prototypage à une simple automatisation du maquettage est réducteur. Pour les applications à fortes contraintes temps-réel, considérer un prototype comme une version préliminaire d'un produit est, certes, délicat, mais l'expérience S.A.O. a montré que c'était possible. Ainsi, dans bien des domaines, un outil de prototypage doit pouvoir produire un code satisfaisant.

Nous pouvons distinguer deux philosophies. La première est celle du prototype, illustrée par PROTO. Le modèle n'est qu'une maquette astreinte à un environnement donné. Il est possible d'extraire toutes sortes d'informations du modèle mais son implémentation sera à effectuer avec les moyens traditionnels. Les risques de mauvaise interprétation du modèle ne sont donc pas complètement éliminés, même si, a priori, le problème et sa solution ont été complètement définis.

L'autre approche est plus que du prototypage [Cassigneul 91, Estraillier 91] : il s'agit réellement de *génération automatique de programme*. Le prototype généré est récupérable hors de l'environnement qui l'a produit. La conception d'un système se fait donc uniquement par l'intermédiaire de sa spécification. Cela

peut poser des problèmes de maintenance du programme généré; il est cependant tout à fait concevable que l'évolution d'un produit conçu de cette manière se base sur sa modélisation, et non sur son implémentation.

+

		Introduction générale
Chapitre I	:	*Prototypage de systèmes parallèles*
Chapitre II	:	*Interprétation sémantique d'un réseau de Petri*
Chapitre III	:	*Méthodologie de prototypage*
Chapitre IV	:	*Structure du prototype généré*
Chapitre V	:	*Caractéristiques du prototype Ada*
Chapitre VI	:	*Le prototype centralisé*
Chapitre VII	:	*Le prototype réparti*
Chapitre VIII	:	*CPN/TAGADA : L'outil de prototypage*
		Conclusion générale
		Bibliographie
		Annexes & Appendices

Chapitre II :

Interprétation sémantique d'un réseau de Petri

1. Introduction 67
2. Les G-objets 69
   - 2.1. G-objet de type processus....69
   - 2.2. G-objet de type Actions....72
   - 2.3. G-objet de type Etats_Processus....73
   - 2.4. G-objet de type Ressources....74
   - 2.5. Semi-flots et G-objets....75
   - 2.6. Synthèse....80
3. Algorithme de décomposition en processus 81
   - 3.1. L'algorithme....81
   - 3.2. Transformation d'un modèle en vue de le rendre "prototypable"....87
   - 3.3. Vers une extension de l'algorithme de décomposition en processus....92
4. Synthèse 100

1. Introduction

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

- Analyse du modèle ouvert d'un système, car il correspond à la spécification du système que l'on réalise (Chapitre I, section I.3). Il nous permet de générer le squelette de contrôle du prototype.
- Interprétation de la représentation des activités concurrentes; à cette fin, nous introduisons les *G-objets* qui constituent des types de composants logiciels.

Définition II.1 : **G-objets**

Sous-ensemble d'un réseau de Petri (modèle ouvert). A partir des G-objets, selon leur nature, des séquences spécifiques de code seront générées.

Nous avons déterminé les quatre types de G-objets suivants :

- Les *Processus* sont constitués par des sous-ensembles du réseau de Petri assimilables à une machine à états [Hack 74]. Notre méthode de prototypage est basée sur une stratégie optimale au sens défini dans le Chapitre I.
- Les *Actions* sont associées aux transitions du modèle et correspondent aux traitements effectués par le système. Une Action peut également réaliser un mécanisme de synchronisation entre Processus.
- Les *Etats_Processus* correspondent à certaines places du modèle et représentent les états d'un Processus.
- Les *Ressources* correspondent aux places du modèle qui ne sont pas des Etats_Processus et représentent des zones de stockage de données.

Dans un premier temps, nous définissons la notion de Processus. Par la suite, nous décrivons les Actions et les Etats_Processus, qui en sont directement issus, avant de définir les Ressources d'un modèle.

Dans la deuxième partie de ce chapitre, nous précisons l'algorithme de décomposition automatique du modèle ouvert en G-objets. Un tel modèle doit préalablement avoir été validé.

Notre démarche consiste, à partir d'une décomposition en ensembles séquentiels (les Processus), à caractériser les G-objets d'un réseau de Petri. La méthodologie de prototypage est détaillée dans le Chapitre III.

Certains résultats de la validation sont utilisés pour obtenir la décomposition du réseau de Petri en G-objets. En particulier, l'algorithme proposé s'appuie sur la notion d'invariant de places (semi-flots) [Brams 82, Memmi 83, Kordon 91b].

La caractérisation des G-objets n'est possible que pour les réseaux de Petri vérifiant quatre propriétés fondamentales décrites en section 3. En fonction des propriétés respectées, nous définissons trois catégories de réseaux de Petri :

- Les modèles “prototypables”, pour lesquels nous pouvons obtenir automatiquement une décomposition en G-objets;
- Les modèles “transformables”, pour lesquels nous pouvons obtenir automatiquement un modèle “prototypable” équivalent;
- Les modèles “non-prototypables”, pour lesquels la caractérisation des G-objets est actuellement impossible. Ces modèles doivent être modifiés par leur concepteur afin de respecter les propriétés de “prototypabilité”;

Enfin, nous proposons quelques extensions à l’algorithme actuel en vue d’étendre la classe des réseaux de Petri “prototypables”.

2. Les G-objets

Nous présentons dans ce paragraphe les G-objets caractérisés par l'algorithme de décomposition d'un réseau de Petri. Cet algorithme est détaillé dans le paragraphe suivant.

2.1. G-OBJET DE TYPE PROCESSUS

Cette section propose de construire les *Processus* à partir des ensembles séquentiels (Définition I.10). De tels ensembles représentent des modèles de comportement séquentiels, liés à la notion de machine à états dans : aucun parallélisme n'est possible. Nous les interprétons comme des *classes* de Processus [Hack 74].

Chaque classe est instanciable : 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* est une instance du modèle de comportement défini par un Processus.

Dans les systèmes d'exploitation, un processus est caractérisé par son contexte d'exécution [Krakowiak 85]. Il regroupe l'ensemble des informations courantes du processus, c'est-à-dire les registres (processeur et utilisateurs). Nous devrons transposer cette notion de contexte dans le prototype. Nous définissons ainsi les notions de *compteur ordinal* et de *mot d'état*.

2.1.1. Compteur ordinal d'un Processus

Pour désigner l'Action à exécuter, il est nécessaire de transposer la notion de compteur ordinal : il s'agit de la transition activée (traitement à effectuer).

Définition II.2 : **Compteur ordinal d'un Processus**

Le compteur ordinal désigne une étape dans le comportement du Processus. Ces étapes correspondent, soit à des Actions (Définition II.4), soit à des Etats_Processus (Définition II.5).

Exemple II.1 : Nous donnons en Figure II.1 une interprétation du modèle décrit dans Exemple I.5. Cette décomposition est conforme à ce qui a été indiqué dans Définition I.10. Les Processus (E1, E2 et E3) définissent un modèle de comportement. Chacun d'eux communique avec ses voisins de façon synchrone ou asynchrone.
Le compteur ordinal du Processus E1 peut prendre les valeurs suivantes :

- *Section 2* : signifiant que la transition *2 vers 3* est candidate à l'activation pour le Processus;
- *2 vers 3* : signifiant que la transition *2 vers 3* est en cours d'activation;
- *Section 3* : signifiant que la transition *3 vers 4* est candidate à l'activation pour le Processus;
- *3 vers 4* : signifiant que la transition *3 vers 4* est en cours d'activation;
- *F2* : signifiant que la transition *1 vers 2* est candidate à l'activation pour le Processus;
- *1 vers 2* : signifiant que la transition *1 vers 2* est en cours d'activation.

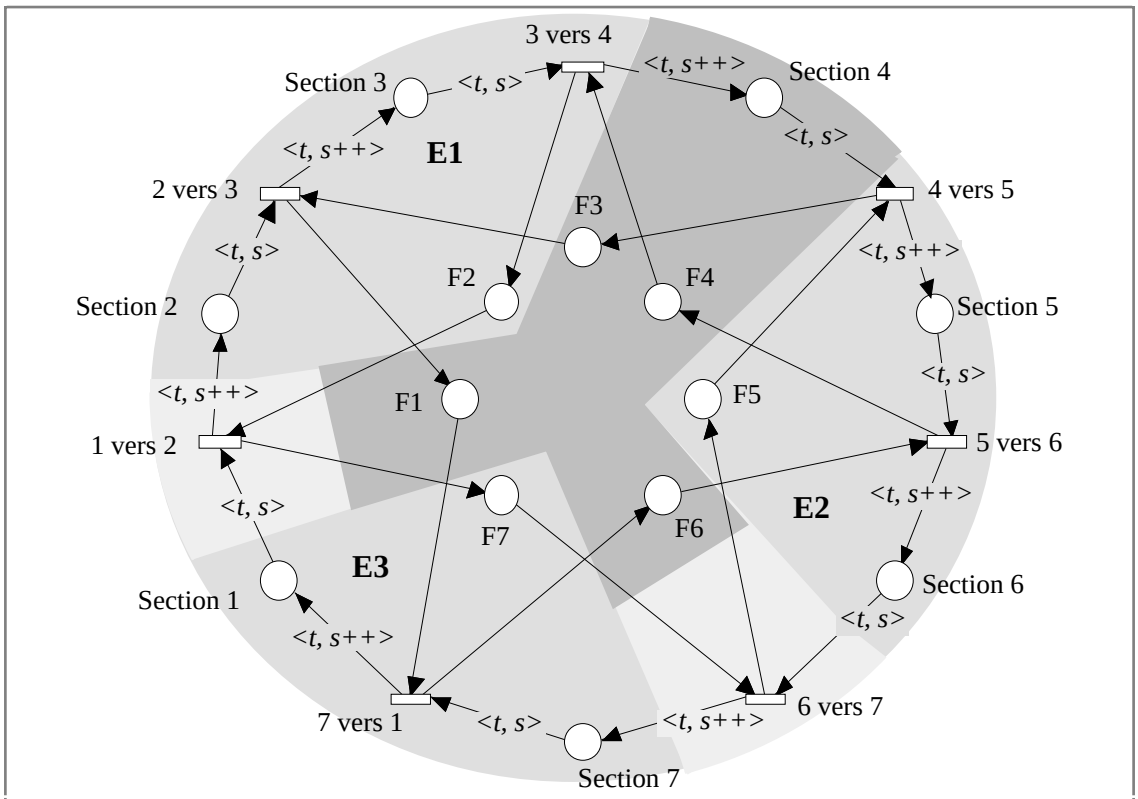


Figure II.1 : Décomposition en Processus du modèle défini dans Exemple I.5.

2.1.2. Mot d'état d'un Processus

Un mot d'état caractérise les données manipulées par un Processus instancié.

Définition II.3 : **Mot d'état d'un Processus**

> Ensemble des informations manipulées durant les étapes de comportement d'un Processus.
> Il est représenté par un N-uplet de valuations des arcs de pré et post conditions des transitions. Naturellement, seuls les arcs qui relient les étapes de comportement d'un Processus sont considérés .

La notion de mot d'état d'un Processus est liée à la coloration d'un modèle. Si aucune marque colorée ne circule dans les places reliant les transitions d'un Processus entre elles, il n'y a pas de mot d'état.

Dans le cas contraire, l'activation des transitions de ce Processus peut être soumise à la présence de marques de couleurs particulières véhiculées avec le compteur ordinal. Ces marques constituent le mot d'état.

Exemple II.2 : Considérons le Processus E1 (Figure II.2). Certains arcs de ce Processus véhiculent une information sous la forme de marques colorées (t et s). Nous définissons de la sorte un mot d'état pour le Processus E1. Il s'agit du couple représenté par les variables t et s. Dans le cas présent, les valeurs des variables *t* et *s* n'influent pas sur "l'activabilité" des transitions du Processus.
Ici, le mot d'état est modifié après le passage des transitions *1 vers 2* et *2 vers 3*, transmis de façon synchrone au Processus E2 via la place *Section 4*, et enfin réceptionné de façon synchrone au niveau de la transition *1 vers 2*.
Entre les transitions *3 vers 4* et *1 vers 2*, les valeurs contenues dans le mot d'état n'ont pas de signification.

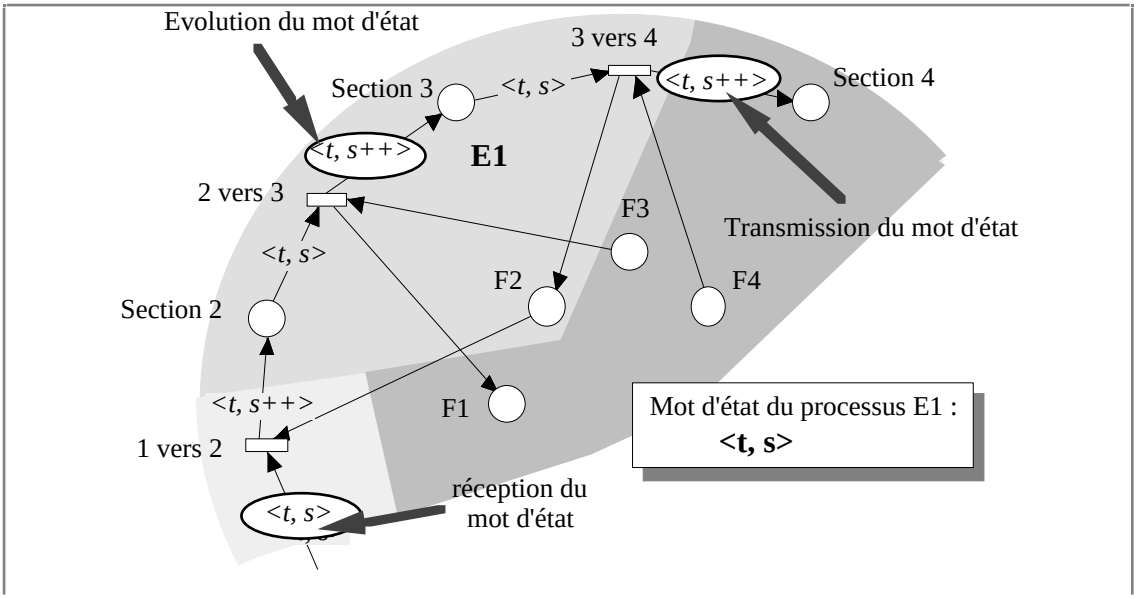


Figure II.2 : Le Processus E1 et le mot d'état qui lui est associé.

2.1.3. Processus instancié

La notion de Processus permet de définir des modèles de comportement séquentiels reposant sur la notion de machine à états. 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 :

- le nombre de processus instanciés est égal à la cardinalité du marquage de l'ensemble des places reliant les transitions du Processus entre elles,
- la valeur du compteur ordinal de chaque processus instancié à son démarrage est indiquée par la position de la marque initiale définissant l'instance,
- le mot d'état initial de chaque processus instancié est défini, s'il y a lieu, par la couleur de la marque repérant l'instance.

Exemple II.3 : Reprenons la décomposition en trois Processus du modèle défini dans Exemple I.5. Le marquage initial de ce modèle permet de déterminer que chacun des trois Processus E1, E2 et E3 est instancié une seule fois (Figure II.3) :

- Le compteur ordinal de l'unique instance de E1 possède initialement la valeur *F2* signifiant que 1 *vers 2* est la prochaine Action à exécuter. Son mot d'état n'a pas de valeur initiale définie.
- Le compteur ordinal de l'unique instance de E2 possède initialement la valeur *Section 5*, signifiant que *5 vers 6* est la prochaine Action à exécuter. Son mot d'état a la valeur <2, 5>.
- Le compteur ordinal de l'unique instance de E3 possède initialement la valeur *Section 1*, signifiant que *1 vers 2* est la prochaine Action à exécuter. Son mot d'état a la valeur <1, 1>.

Il n'est pas tenu compte, dans le calcul du nombre de processus instanciés, du marquage des places correspondant à des zones de stockage de données.

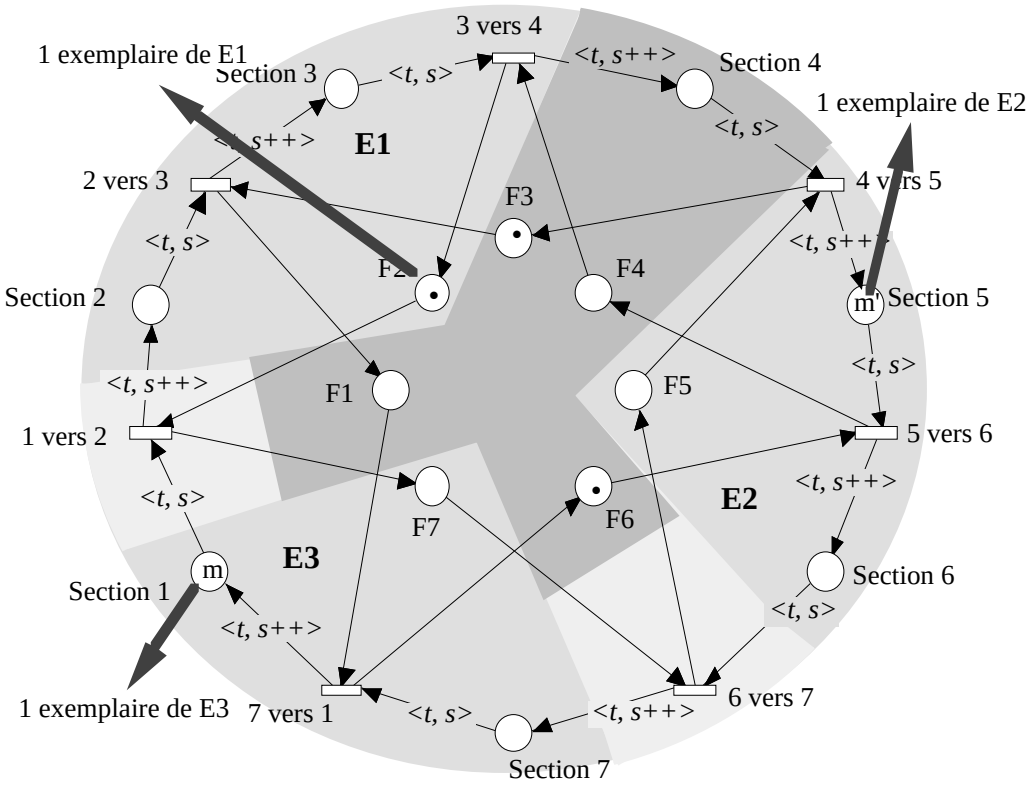


Figure II.3 : Caractérisation des processus instanciés du modèle défini dans Exemple I.5.

2.2. G-OBJET DE TYPE ACTIONS

Une Action représente une partie du traitement effectué par un Processus. Dans le modèle, elle correspond à un niveau de description atomique. Elle est naturellement associée aux transitions.

Conformément à la théorie des réseaux de Petri [Jensen 81, Brams 82, Chiola 90, Dutheillet 91], "l'exécution" d'une transition se décompose en :

- évaluation de la précondition (éventuellement bloquante);
- réalisation du traitement associé;
- production du marquage postcondition.

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 œuvre 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 (Définition II.6), c'est-à-dire de zones de stockage contenant des données produites par d'autres processus instanciés; il y a réalisation d'une communication asynchrone.

Les contraintes directes définissent des Actions dont les conditions d'activation dépendent du contexte de plusieurs processus instanciés. Elles

sont exprimées par des arcs reliant des Etats_Processus (Définition II.5) de différents Processus à l'Action.

Les contraintes indirectes définissent des Actions dont "l'activabilité" est soumise au contexte d'un seul processus instancié. Elles sont exprimées par des arcs reliant des Ressources à l'Action ou par un prédicat, définissant des relations entre les marques colorées consommées à l'activation.

***Définition II.4 :* Action (simple, synchronisée, gardée)**

Une Action est une transition du modèle. Elle correspond à un traitement.
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.

Exemple II.4 : Nous nous intéressons au Processus E1 (Figure II.4) : *2 vers 3* et *3 vers 4* sont des Actions simples gardées. *1 vers 2* est une Action synchronisée.

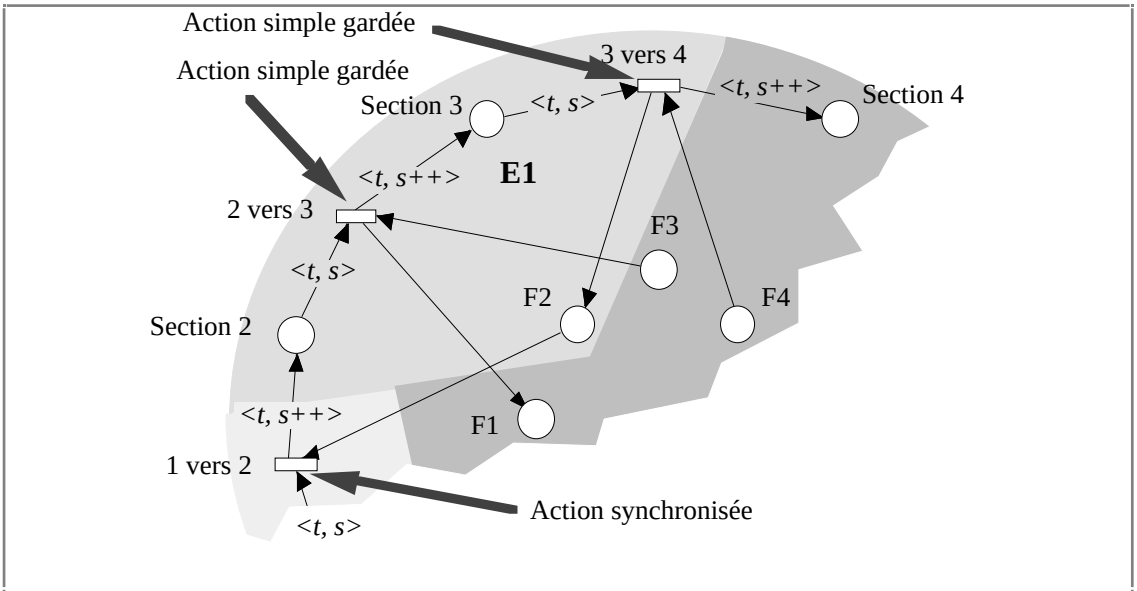


Figure II.4 : Classification des Actions dans le Processus E1 du modèle défini dans Exemple I.5.

2.3. G-OBJET DE TYPE ETATS_PROCESSUS

Certaines places du modèle relient les Actions d'un Processus entre elles (places "état" dans [Souissi 90]). Elles sont appelées Etat_Processus.

Dans un modèle validé, un Etat_Processus sans successeur est un état d'accueil [Brams 82] pour le Processus. Il est associé à une terminaison du traitement. Un Etat_Processus ayant plusieurs successeurs est considéré comme un point de choix.

***Définition II.5 :* Etat_Processus (simple, alternatif, de terminaison)**

Un Etat_Processus est une place définissant une contrainte directe entre plusieurs Actions d'un Processus. S'il ne comporte pas de successeur, il est dit de terminaison. S'il comporte un seul successeur, il est simple. Dans les autres cas, il est qualifié d'alternatif.

Exemple II.5 : Si nous considérons le modèle défini dans Exemple I.5 et sa décomposition en trois Processus (Figure II.1), tous les Etats_Processus de E1, E2 et E3 sont simples.
Si nous considérons le modèle donné en Figure II.5, P1 et P2 sont des Etats_Processus simples, P3 un Etat_Processus alternatif et P4 un Etat_Processus de terminaison.

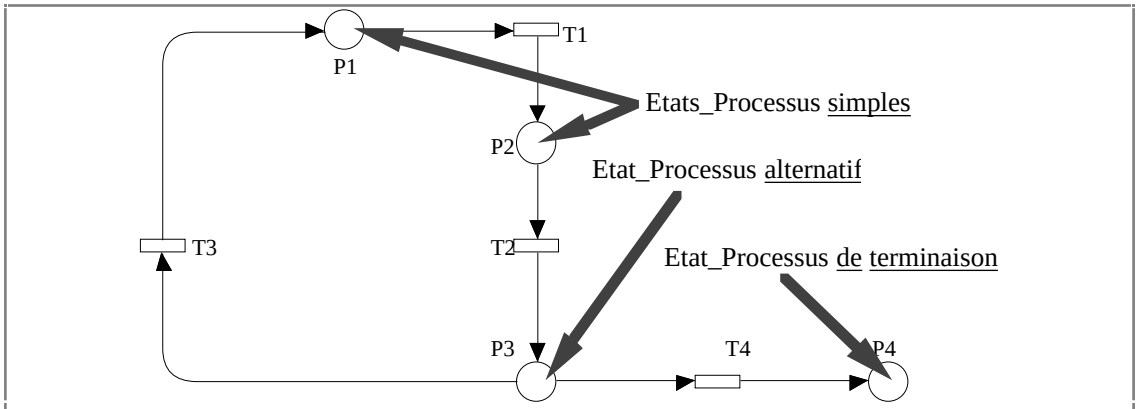


Figure II.5 : Classification des Etats_Processus.

2.4. G-OBJET DE TYPE RESSOURCES

Certaines places du modèle représentent des zones de stockage de données. Elles sont interprétées comme des supports de mécanismes de communication asynchrone entre les Processus du modèle (les buffers dans [Colom 86] ou canal dans [Souissi 90]). Nous appelons de telles places des Ressources.

***Définition II.6 :* Ressource**

Une Ressource est une place qui n'est pas contenue dans un Processus. Elle 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.

Exemple II.6 : Si nous considérons le modèle défini dans Exemple I.5 et sa décomposition en trois Processus (Figure II.1), les Ressources du modèle sont les places associées à la transmission asynchrone de données entre Processus : *F1*, *F3*, *F4*, *F6* et *Section 4* (Figure II.6). Ici, seule la présence d'un message dans *F1*, *F3*, *F4* et *F6* a une influence sur l'activation des Actions des Processus concernés. L'information transitant dans *Section 4* possède une structure définie par le domaine de couleurs de la Ressource.

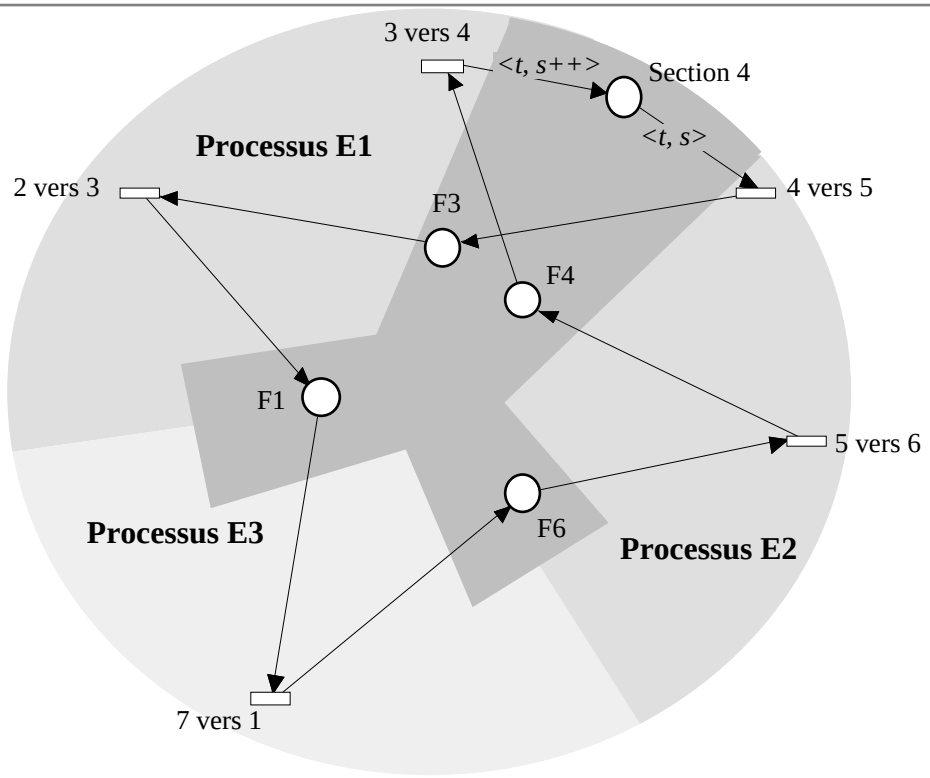


Figure II.6 : Les Ressources du modèle décrit dans Exemple I.5 si l'on s'intéresse à la décomposition en trois Processus (Figure II.1).

2.5. SEMI-FLOTS ET G-OBJETS

Après avoir défini les différents G-objets d'un modèle, nous allons les relier à la théorie des réseaux de Petri, et, en particulier, à la notion de semi-flot [Brams 82, Memmi 83, Haddad 88].

Pour caractériser les activités concurrentes représentées par les Processus, nous utilisons des propriétés structurelles du réseau de Petri. Cela n'est cependant possible que si le modèle est structurellement contraint [Souissi 90, Chehaibar 91, Valette 82]. Cette notion est liée à des propriétés que nous définirons en Section 3.

Ainsi, tous les réseaux de Petri ne sont pas interprétables. Cela ne constitue pas un problème majeur dans le cadre de notre approche. Si le modèle n'est pas directement interprétable en terme d'activités concurrentes, le concepteur peut, par raffinements successifs, "constituer" un modèle conforme à notre méthode.

Par ailleurs, si le réseau de Petri est le résultat d'une traduction depuis un formalisme de plus haut niveau, il comporte des structures caractéristiques. Il suffit alors de vérifier qu'elles intègrent les restrictions structurelles que nous avons définies.

2.5.1. Semi-flots d'un réseau de Petri Ordinaire

Les semi-flots sont des invariants définissant des relations entre le marquage de certaines places d'un réseau de Petri.

Les semi-flots d'un réseau de Petri Ordinaire sont actuellement parfaitement définis d'un point de vue théorique. Et plusieurs algorithmes de calcul ont été proposés [Chvatal 83, Trèves 89, Chiola 87].

Dans un réseau de Petri, tout franchissement d'une transition entraîne une modification du marquage (Définition I.7). Cette différence est exprimée par la matrice d'incidence [Brams 82, Haddad 88].

Définition II.7 : **Matrice d'incidence d'un réseau de Petri**

Etant donné un réseau de Petri Ordinaire *R*, sa matrice d'incidence *W* est une application *PxT* dans $\mathbb{Z}$ définie par :

- $p \quad P, \quad t \quad T, W(p,t) = Post(p,t) - Pré(p,t)$

L'équation de changement d'état d'un réseau de Petri exprime que la différence entre deux marquages accessibles reste dans l'image de la matrice d'incidence *W*.

Nous définissons les flots comme des formes linéaires annulant la fonction d'incidence. Les flots définissent alors des invariants linéaires pour les réseaux de Petri.

Définition II.8 : **Flot d'un réseau de Petri**

Soit R un réseau de Petri Ordinaire de matrice d'incidence *W* et *f* un vecteur de places. f est un flot de R si et seulement si :

- ${}^{t}f.W = 0$

Les flots à coefficients positifs ont un intérêt particulier : ils impliquent que les places à coefficient non-nul dans le flot restent bornées.

Définition II.9 : **Semi-flot d'un réseau de Petri**

Un semi-flot de R est un flot à coefficients positifs.

Une famille génératrice de semi-flots est le plus petit ensemble de semi-flots permettant d'exprimer, par combinaison linéaire, tous les semi-flots d'un réseau de Petri.

Définition II.10 : **Famille génératrice de semi-flots**

Soit R un réseau de Petri Ordinaire, et $F = \{f_1,\ldots,f_n\}$ une famille génératrice de semi-flots. Nous avons :

- f semi-flot de R | $f \quad F, \quad a_1,\ldots,a_n \quad \mathbf{Q}^+ \mid f = a_1.f_1 + \ldots + a_n.f_n$.

F est une famille génératrice de support minimal de semi-flots si et seulement si :

- $f \quad F, \ /, \ a_1,\ldots,a_n \quad \mathbf{Q}^+ \mid f = a_1.f_1 + \ldots + a_n.f_n$.

2.5.2. Le modèle structurel

Un ensemble séquentiel déterminant un Processus est constitué de places et de transitions. Dans un tel ensemble, un invariant de places, représentant les étapes du comportement d'un Processus est directement lié à la notion de machine à états. Le mécanisme d'instanciation dynamique des Processus préserve naturellement la cohérence des invariants.

Comme nous recherchons une décomposition optimale en Processus, nous devons la calculer à partir d'un ensemble minimal de semi-flots (une *famille génératrice de support minimal*).

Cependant, l'extension de la définition aux réseaux de Petri Colorés pose un problème. Deux interprétations des semi-flots sont possibles :

- ils sont considérés comme des valuations de places par des fonctions de couleurs [Jensen 81, Couvreur 90a];
- ils sont considérés comme étant associés à un dépliage du réseau Coloré [Vautherin 84].

Dans le premier cas, leur calcul est complexe. Actuellement, il existe des algorithmes pour certaines classes de réseaux de Petri [Couvreur 90a] mais leur interprétation reste délicate. Par ailleurs, cette notion n'a pas encore été étendue aux réseaux Bien Formés définis dans [Chiola 90, Dutheillet 91].

L'autre interprétation consiste à considérer des flots d'un point de vue structurel. Une telle approche coïncide avec notre interprétation d'un réseau de Petri en termes d'activités concurrentes. Une base de semi-flots indépendante de la coloration du réseau de Petri suffit : les semi-flots globaux [Kordon 91b] correspondent à cette définition. Cela revient à prendre le modèle équivalent en modifiant la valuation des arcs. Nous définissons ainsi le *modèle structurel*.

S'il est indéniable que nous perdons des informations durant le passage d'un réseau de Petri au modèle structurel associé, son utilisation se justifie pour les raisons suivantes :

- En utilisant le modèle structurel, nous pouvons prototyper des modèles pour lesquels il est encore impossible d'obtenir de tels résultats théoriques.
- Pour les réseaux de Petri Colorés, il est difficile d'interpréter, en terme de modèles de comportement, une famille génératrice de semi-flots. En effet, ces semi-flots sont très nombreux et souvent paramétrés en fonction de la parité des domaines de couleurs [Couvreur 90a].

Définition II.11 : **Modèle structurel**

Il s'agit du modèle structurellement "équivalent" mais non coloré obtenu par transformation du modèle initial :

- le prédicat associé à chaque transition disparaît (il est toujours vérifié);

- le domaine de couleurs associé à chaque place disparaît (il est forcé à "pas de couleur");
- les valuations des arcs sont calculées de la manière suivante :

  $$Val(Arc_{\text{modèle structurel}}) = | Val(Arc_{\text{modèle initial}}) |$$

  la valuation de chaque arc du modèle structurel correspond au nombre de marques circulant sur l'arc correspondant dans le modèle initial;
- le marquage initial des places du modèle structurel est calculé de la manière suivante :

  $$M(P_{\text{modèle structurel}}) = | M(P_{\text{modèle initial}}) |$$

  chaque place du modèle structurel contient un nombre de marques *non colorées* égal à celui contenu dans la place correspondante du modèle initial.

Exemple II.7 : Considérons le modèle de la Figure II.7. Un ensemble de producteurs dialoguent avec un ensemble de consommateurs. Le modèle structurel associé ne comporte plus de valuations colorées. A l'exception de celui reliant la place Message à la transition Attendre, sur lequel trois marques circulent, les arcs du modèle structurel associé sont valués à 1.
Le marquage de la place Prêt, dans le modèle structurel, est composé de n marques non colorées, n étant le nombre de marques contenues dans Prêt, dans le modèle initial.

REMARQUE : Il n'y a pas de différence entre un réseau de Petri non-coloré et le modèle structurel qui lui est associé.

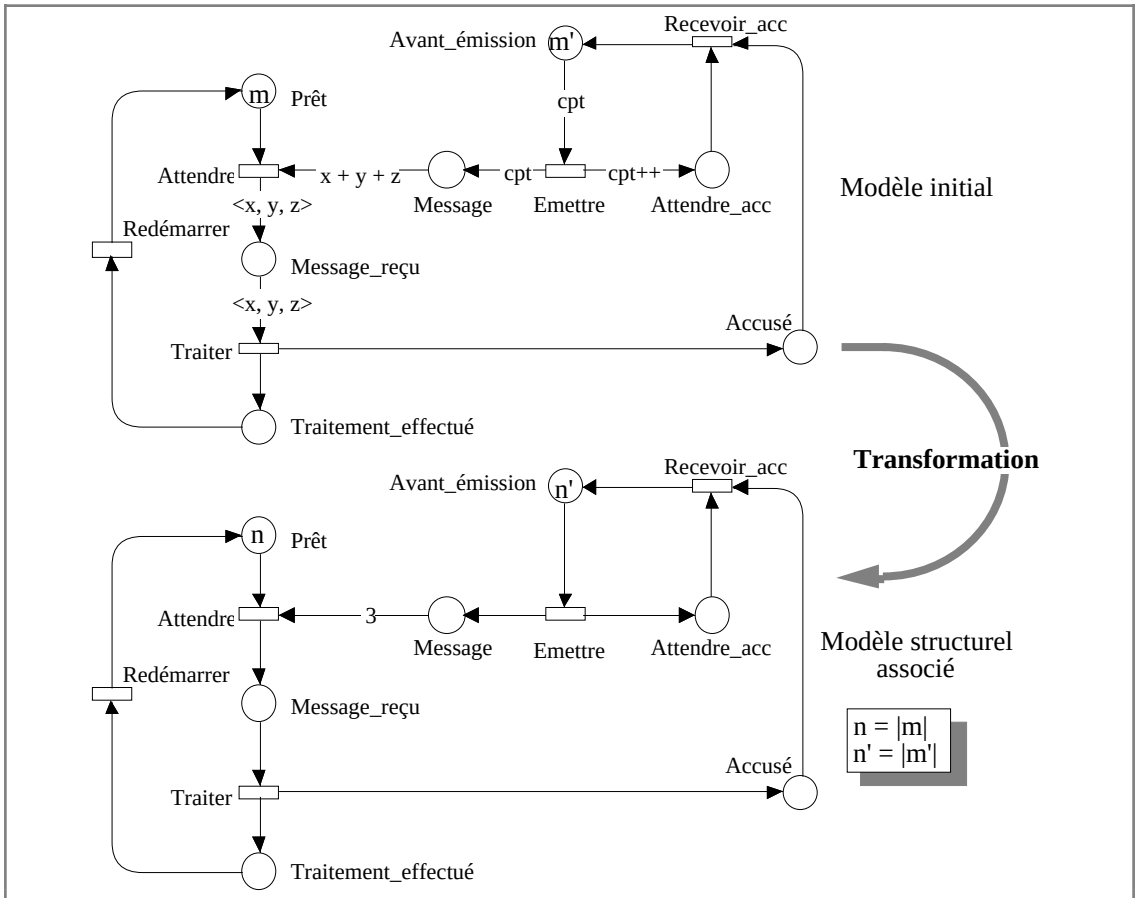


Figure II.7 : Exemple de réseau de Petri Bien Formé avec le modèle structurel qui lui est associé.

La décomposition d'un réseau de Petri bien formé en un ensemble de Processus s'appuie sur une famille génératrice de semi-flots de support minimal calculée sur le modèle structurel associé.

Les algorithmes de calcul existent et sont déjà implémenté dans divers environnements de travail. Deux catégories d'algorithmes sont actuellement répertoriés. Les premiers se basent sur la méthode de Farkas [Chiola 87, Trèves 89], les autres sur la méthode du simplexe [Chvatal 83].

Pour notre prototypage, nous utilisons les logiciels COMBAG [Trèves 89] et GreatSPN [Chiola 87], intégrés dans l'Atelier de Modélisation Interactive AMI [Bernard 89, Bernard 90], présenté dans le Chapitre VIII.

2.5.3. Vers une interprétation des semi-flots

Soit F, une famille génératrice de semi-flots de support minimal du modèle structurel. Nous observons que certains éléments fp F permettent de décrire les Processus. Si fp décrit un Processus, alors il s'énonce ainsi:

$$\sum_{i=1}^{\text{nombre de places}} \text{Place}_i$$

Chaque place de fp correspond à un Etat_Processus du Processus décrit par fp. Considérons F_P l'ensemble des semi-flots décrivant un Processus dans le modèle structurel. Nous avons : $|F_P| \leq |F|$.

Lorsque $|F_P| = |F|$, le modèle ne peut être interprété que d'une seule façon puisque tous les semi-flots de la base permettent de définir un Processus. Dans le cas contraire, il se peut qu'il y ait plusieurs décompositions possibles. Chacune d'elles correspond à un point de vue sémantique différent sur le modèle.

Exemple II.8 : Considérons le modèle structurel associé au réseau de Petri défini dans l'exemple I.5. Nous pouvons obtenir trois interprétations possibles (Figure II.8) :

1 í Un partitionnement en trois Processus (décomposition 1): E1 (*Section 2*, *Section 3* et *F2*), E2 (*Section 5*, *Section 6* et *F5*) et E3 (*Section 7*, *Section 1* et *F7*).

2 í Un regroupement en un Processus (décomposition 2) : Train (*Section 1*, *Section 2*, *Section 3*, *Section 4*, *Section 5*, *Section 6* et *Section 7*),

3 í Un regroupement en un Processus (décomposition 3) : Feux (*F1*, *F2*, *F3*, *F4*, *F5*, *F6* et *F7*).

Dans la première décomposition, nous regroupons les sections en vue d'une gestion décentralisée de la voie. Dans la seconde décomposition, nous considérons le cheminement d'un train comme un Processus. Dans la troisième décomposition, nous nous intéressons aux feux verts.

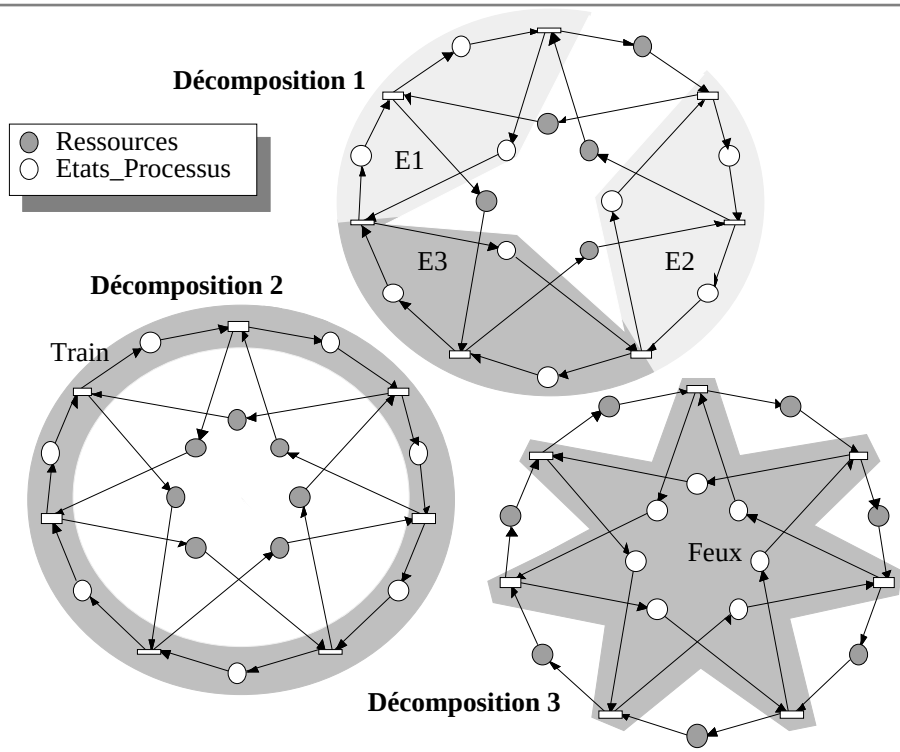


Figure II.8 : Plusieurs décompositions possibles et licites pour le modèle décrit dans Exemple I.5.

S'il existe plusieurs décompositions valides, il est impossible de résoudre le problème sans l'aide de son concepteur. Lui seul sait exactement ce qu'il veut modéliser. Par ailleurs, certains aspects de l'architecture cible (nombre de processeurs...) peuvent favoriser une interprétation plutôt qu'une autre.

2.6. SYNTHÈSE

Notre interprétation d'un réseau de Petri, en vue de son implémentation, repose sur la notion de Processus. Après l'avoir effectuée, nous pouvons caractériser les G-objets qui composent le prototype.

Nous avons ensuite caractérisé les relations entre Processus et semi-flots globaux d'un réseau de Petri. Cela nous permet, dans la section suivante, de présenter un algorithme de caractérisation des Processus.

Une fois la décomposition en Processus connue, les autres G-objets sont automatiquement identifiables :

- Les Etats_Processus sont donnés par le support des places des semi-flots décrivant les Processus. Leurs attributs (de terminaison, simple ou alternatif) sont calculés en fonction du nombre de successeurs des places ainsi désignées.
- Les Ressources sont désignées par les places n'appartenant à aucun semi-flot décrivant les Processus.
- Les Actions correspondent aux transitions du modèle. Les attributs qui les caractérisent se déduisent de la décomposition en Processus (Action partagée par plusieurs Processus, gardée par une Ressource...).

3. Algorithme de décomposition en processus

Ce paragraphe présente un algorithme de décomposition d'un réseau de Petri en ensembles de Processus, d'Actions, d'Etats_Processus et de Ressources.

Nous avons déterminé trois classes de réseaux de Petri :

- Les modèles "prototypables" pour lesquels la décomposition en Processus peut-être obtenue. Ces modèles vérifient les 4 propriétés fondamentales que nous avons définies (section 3.1.1).
- Les modèles "transformables", qui ne vérifient pas toutes les propriétés des modèles "prototypables". Ils peuvent cependant être automatiquement transformé en modèles "prototypables" équivalents [Colom 86].
- Les modèles "non-prototypables" ne vérifient pas toutes les propriétés des modèles "prototypables". De plus, ils ne peuvent être transformés de façon triviale.

Nous terminons cette section en donnant des indices en vue d'une extension de l'algorithme actuel.

3.1. L'ALGORITHME

L'algorithme que nous avons défini permet de décomposer un modèle en un ensemble de Processus communicants. Le calcul échoue si le modèle n'est pas "prototypable".

Nous définissons dans un premier temps la classe des modèles "prototypables" pour lesquels nous pouvons effectuer la décomposition. Nous détaillons ensuite l'algorithme permettant de calculer les Processus. Si l'algorithme échoue, nous pouvons déterminer si le modèle est "transformable" ou "non-prototypable".

3.1.1. La classe des modèles "prototypables"

Un modèle est "prototypable" s'il est décomposable en Processus. Pour cela, il faut pouvoir extraire de la famille génératrice de semi-flots de support minimal du modèle structurel associé au moins un sous-ensemble de semi-flots F_p vérifiant les quatre propriétés énoncées ci-après.

***Propriété 1* : Couverture de l'ensemble des Actions**

$\Gamma(F_P) = T$: les semi-flots composant F_P couvrent l'ensemble des transitions du modèle.

Aucune transition ne doit être écartée de la décomposition. Si tel est le cas, les transitions isolées ne pourront être incluses dans des Processus. Nous donnons en Figure II.9 quelques exemples de modèles pour lesquels la propriété 1 n'est pas vérifiée.

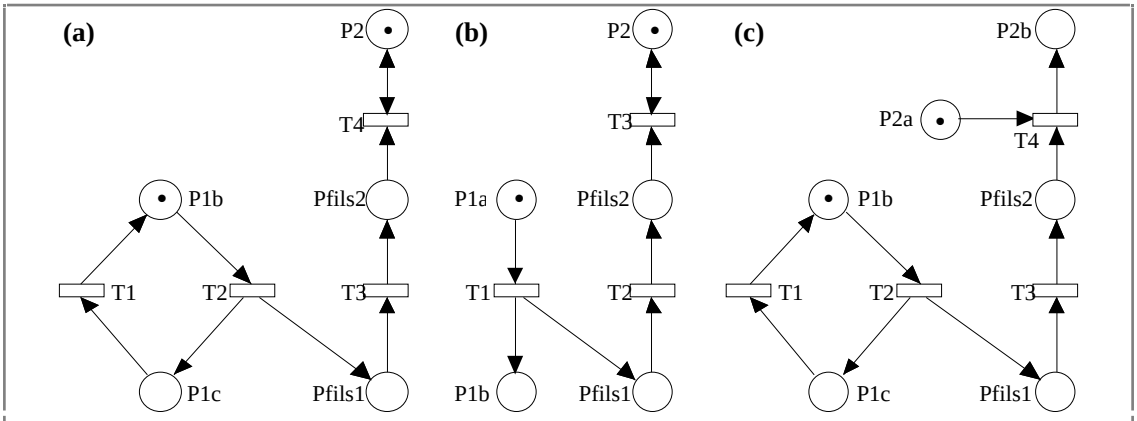


Figure II.9 : Exemples de modèles ne vérifiant pas la propriété 1.

Exemple II.9 : La base de semi-flots de support minimal du modèle (a) (Figure II.9) comporte deux éléments :

- P1b + P1c,
- P2.

Ces deux semi-flots ne permettent pas de couvrir l'ensemble des transitions du modèle. Il en est de même avec les semi-flots du modèle (b) (P1a + P1b et P2) et du modèle (c) (P1b + P1c et P2a + P2b). Ces deux derniers modèles sont des "cas d'école" non vivants.

Propriété 2 : Indépendance des Processus d'un modèle

p_1 p_2 F_P, $p_1 \cap p_2 =$: les semi-flots de F_P n'ont aucune place en commun.

Les places précondition ou postcondition d'Actions appartenant à différents Processus sont des Ressources. Il y a contradiction avec la Définition II.5 décrivant un Etat_Processus. Nous donnons en Figure II.10 quelques exemples de modèles pour lesquels la propriété 2 n'est pas vérifiée.

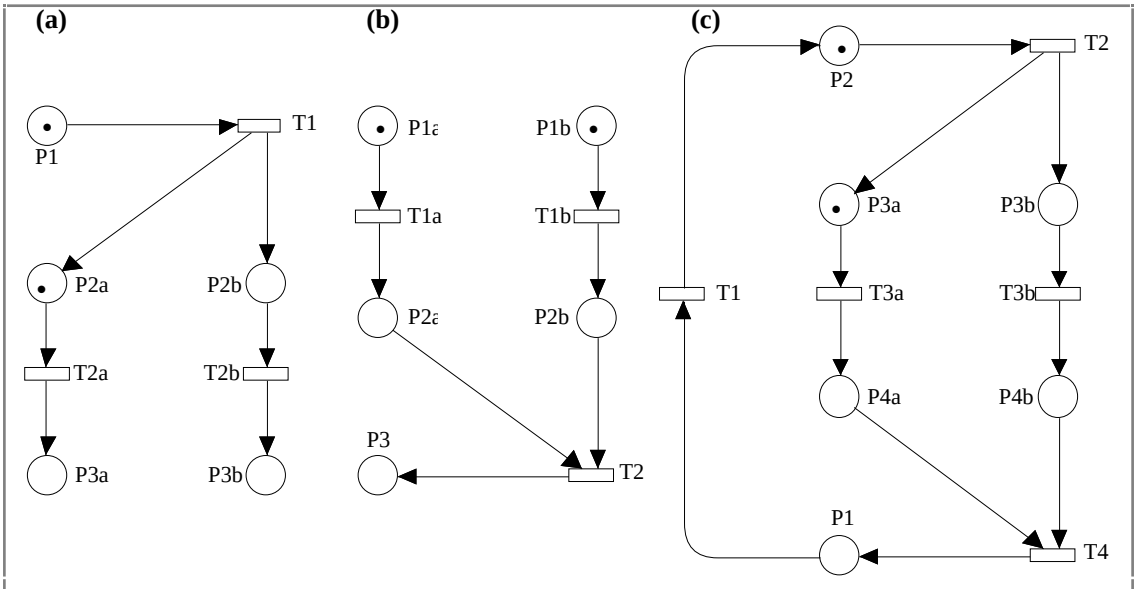


Figure II.10 : Exemples de modèles ne vérifiant pas la propriété 2.

Exemple II.10 : Considérons les trois réseaux de Petri de la Figure II.10.

La base de semi-flots de support minimal du modèle (a) comporte deux éléments :

- P1 + P2a + P3a,
- P1 + P2b + P3b.

Ces deux semi-flots ont P1 en commun. Il est impossible de déduire deux Processus disjoints.

Il en est de même avec la base de semi-flots de support minimal du modèle (b) :

- P1a + P2a + P3,
- P1b + P2b + P3.

Ces deux semi-flots ont P3 en commun. Il est impossible de déduire deux Processus disjoints.

Enfin, la base de semi-flots de support minimal du modèle (c) est composée de :

- P1 + P2 + P3a + P4a,
- P1 + P2 + P3b + P4b,

Ces deux semi-flots ont les places P1 et P2 en commun. Il est impossible de déduire deux Processus disjoints.

Propriété 3 : Conservation des processus instanciés

$\forall p, t \in \Gamma(F_P)$, alors $|val(p,t)| = |val(t, p)| = 1$: il ne transite qu'une seule marque sur chaque arc reliant les composantes des semi-flots de F_P aux transitions d'un Processus.

Cette propriété est étroitement liée à la notion de compteur ordinal. Par définition, ce compteur ordinal étant unique, il y a conservation du nombre de marques; un processus instancié ne peut se terminer que lorsque la marque matérialisant le compteur ordinal se situe dans une place correspondant à un Etat_Processus de terminaison.

Nous donnons en Figure II.11 quelques exemples de modèles ne respectant pas la propriété 3.

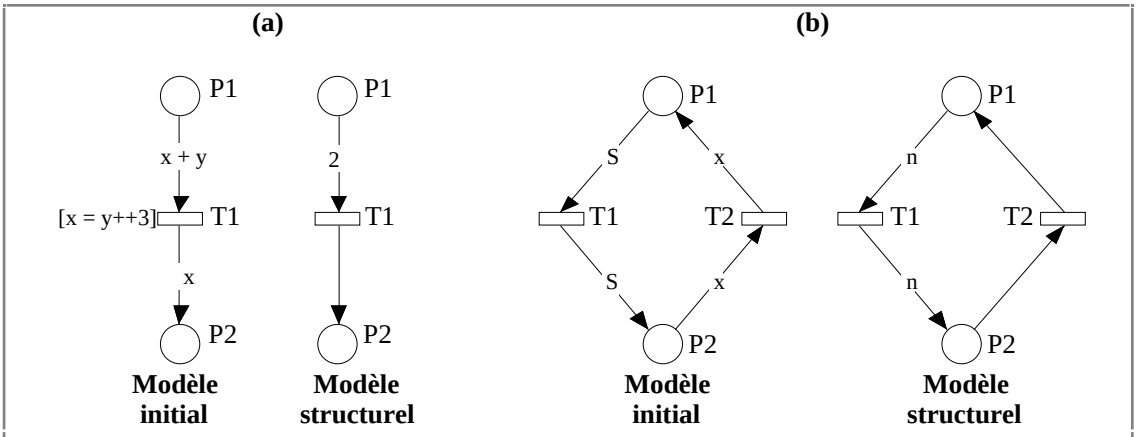


Figure II.11 : Exemples de modèles ne vérifiant pas la propriété 3.

Exemple II.11 : Le modèle (a) (Figure II.11) possède un modèle structurel associé ne vérifiant pas la propriété 3 sur l'arc reliant P1 à T1.

Dans modèle (b), S représente la fonction de diffusion associant un exemplaire de chacune des couleurs définies dans la classe où x prend ses valeurs. S'il y a n couleurs possibles, les arcs reliant P1 à T1 et T1 à P2 dans le modèle structurel ne vérifient pas la propriété 3.

***Propriété 4 :* Cohérence comportementale du modèle**

p F_P, $|M_0 (Sup^2 (F_P))| \geq 1$: tout Processus de la décomposition doit contenir au moins une marque initiale dans les places du semi-flot qui lui sont associées.

Conformément à la théorie des réseaux de Petri, un modèle qui ne vérifie pas la propriété 4 n'est pas vivant car il comporte une composante conservative vide.

3.1.2. Description de l'algorithme

La définition des Actions, Etats_Processus et Ressources repose sur la notion de Processus. Le calcul des décompositions valides d'un réseau de Petri permet donc de déterminer l'ensemble des objets qui le composent. L'algorithme comporte les étapes suivantes :

1. Création du modèle structurel associé au réseau de Petri.
2. Calcul de F, une famille génératrice de semi-flots de support minimal du modèle structurel.
3. Suppression des semi-flots ne vérifiant pas les propriétés 3 ou 4, le résultat est noté F'.
4. Etude de toutes les combinaisons de semi-flots possibles dans F' afin de déterminer celles qui vérifient les propriétés 1 et 2. Nous obtenons de la sorte tous les sous-ensembles F_P possibles.
5. Caractérisation des G-objets.

Pour réduire la durée d'exécution de l'algorithme, il est important de commencer par vérifier des propriétés 3 et 4 car elles se rapportent à des semi-flots. Ceci permet de réduire le nombre de combinaisons à étudier dans la phase 4, les propriétés 1 et 2 s'appliquant à des combinaisons de semi-flots.

La complexité théorique de l'algorithme est directement fonction du nombre d'éléments contenus dans F'. Il faut en effet étudier $2^{|F'|}$ combinaisons[3]. Cependant, il n'est pas nécessaire de toutes les étudier. En effet, nous pouvons nous appuyer sur deux caractéristiques des propriétés 1 et 2.

Caractéristique de la propriété 1 :

Soit E un ensemble d'éléments de F' vérifiant la propriété 1 :

- E' | E E', E' vérifie la propriété 1.

Notre intérêt, dans la décomposition du modèle, est de limiter le nombre d'Actions synchronisées afin d'augmenter le parallélisme potentiel du modèle. Nous nous intéressons par conséquent à l'ensemble E de cardinalité minimale.

Ainsi, dès que nous avons une combinaison d'éléments vérifiant la propriété 1, nous ignorons systématiquement tout sur-ensemble incluant cette combinaison.

---

2 Sup désigne le support de places du semi-flot.

3 où |F'| représente le nombre d'éléments de F'.

Caractéristique de la propriété 2 :

Soit E un ensemble d'éléments de F' ne vérifiant pas la propriété 2 :

- E' | E E', E' ne vérifie pas la propriété 1.

L'application directe de la caractéristique liée à la propriété 2 nous permet de ne pas étudier les combinaisons sur-ensemble d'un groupe d'éléments de F' ne vérifiant pas la propriété 2.

Une fois que nous avons obtenu F_P, un ensemble de semi-flots de F vérifiant les 4 propriétés énoncées, les G-objets sont caractérisés de la façon suivante :

- Les **Processus** : ils sont donnés par les éléments de F_P.
- Les **Etats_Processus** : il s'agit des places couvertes par les éléments de F_P. Leurs attributs (de terminaison, simple ou alternatif) sont calculés en fonction de leur postcondition :
  - |Post (p)| = 0 : l'Etat_Processus est de terminaison,
  - |Post (p)| = 1 : l'Etat_Processus est simple,
  - |Post (p)| > 1 : l'Etat_Processus est alternatif.
- Les **Ressources** : elles sont désignées par les places non couvertes par les éléments de F_P.
- Les **Actions** : elles correspondent aux transitions du modèle. Soit N le nombre d'Etats_Processus en entrée ou en sortie (il y a toujours autant d'Etats_Processus en entrée qu'en sortie) :
  - N = 1 : l'Action est simple,
  - N > 1 : l'Action est synchronisée.

  Si l'Action possède des Ressources en précondition, ou si un prédicat définit des relations entre les marques consommées lors de l'activation, l'Action est gardée.

Exemple II.12 : Détaillons l'algorithme de décomposition en Processus sur le modèle défini dans Exemple I.5 :

ETAPE 1 :

Le modèle structurel associé est donné en Figure II.12:

ETAPE 2 :

Le calcul des semi-flots du modèle structurel donne F, composée de :

1 í F1 + F2 + F3 + F4 + F5 + F6 + F7.
2 í Section 1 + Section 2 + Section 3 + Section 4 + Section 5 + Section 6 + Section 7.
3 í Section 1 + F1 + F3 + F5 + F7.
4 í Section 2 + F1 + F2 + F4 + F6.
5 í Section 3 + F2 + F3 + F5 + F7.
6 í Section 4 + F1 + F3 + F4 + F6.
7 í Section 5 + F2 + F4 + F5 + F7.
8 í Section 6 + F1 + F3 + F5 + F6.
9 í Section 7 + F2 + F4 + F6 + F7.
10 í Section 1 + Section 2 + F1.
11 í Section 2 + Section 3 + F2.
12 í Section 3 + Section 4 + F3.
13 í Section 4 + Section 5 + F4.
14 í Section 5 + Section 6 + F5.
15 í Section 6 + Section 7 + F6.
16 í Section 7 + Section 1 + F7.

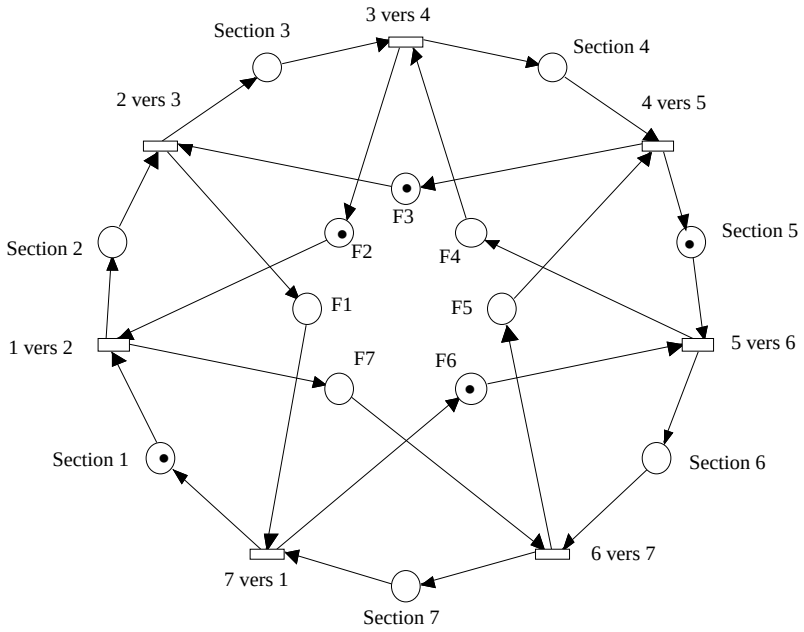


Figure II.12 : Modèle structurel associé au modèle défini dans Exemple I.5.

ETAPE 3 :

Tous les semi-flots calculés vérifient les propriétés 3 et 4, F = F'.

ETAPE 4 :

- {1} vérifie les propriétés 1 et 2,
- {2} vérifie les propriétés 1 et 2,
- {3} ne vérifie pas la propriété 1. En essayant de l'associer avec d'autres semi-flots, nous découvrons que la combinaison {3, 11, 13} vérifie les propriétés 1 et 2.
- etc...

En procédant de la sorte, et en examinant l'ensemble des combinaisons possibles (65536!), nous obtenons 37 décompositions en Processus regroupées en cinq "familles" :

- {1} : le Processus représente l'évolution des feux passant au vert. Si l'on examine le marquage initial du modèle de la Figure II.1, il est instancié trois fois (il y a toujours trois feux verts dans le circuit). Il ne possède pas de mot d'état puisque les Etats_Processus n'ont pas de domaine de couleurs.
- {2} : le Processus représente le cheminement d'un train. Il est instancié deux fois. Toutes les places support du semi-flot décrivant le processus possèdent un domaine de couleurs; ce processus possède un mot d'état : <t,s>.
- {(10, 12, 14), (11, 13, 15), (12, 14, 16), (13, 15, 10), (14, 16, 11), (15, 10, 12), (16, 11, 13)} : chacun des Processus représente la gestion de deux segments de voie. Ils sont instanciés une fois et possèdent un mot d'état : <t, s>.
- {(3, 13), (4, 14), (5, 15), (6, 16), (7, 10), (8, 11), (9, 12)} : le premier Processus correspond à la gestion de deux segments, il est instancié une fois. Le second associe à la fois certains feux verts et un segment, il est instancié deux fois. Les deux Processus possèdent un mot d'état: <t, s>.
- {(3, 11, 13), (4, 12, 14), (5, 13, 15), (6, 14, 16), (7, 15, 10), (8, 16, 11), (9, 10, 12), (3, 11, 15), (3, 13, 15), (4, 12, 16), (4, 14, 16), (5, 10, 13), (5, 10, 15), (6, 11, 14), (6, 11, 16), (7, 10, 12), (7, 12, 15), (8, 11, 13), (8, 13, 16), (9, 10, 14), (9, 12 14)} : deux Processus sur trois représentent la gestion de deux segments, ils sont instanciés une fois. Le troisième associe à la fois certains feux verts et un segment, il est instancié deux fois. Les trois Processus possèdent un mot d'état: <t, s>.

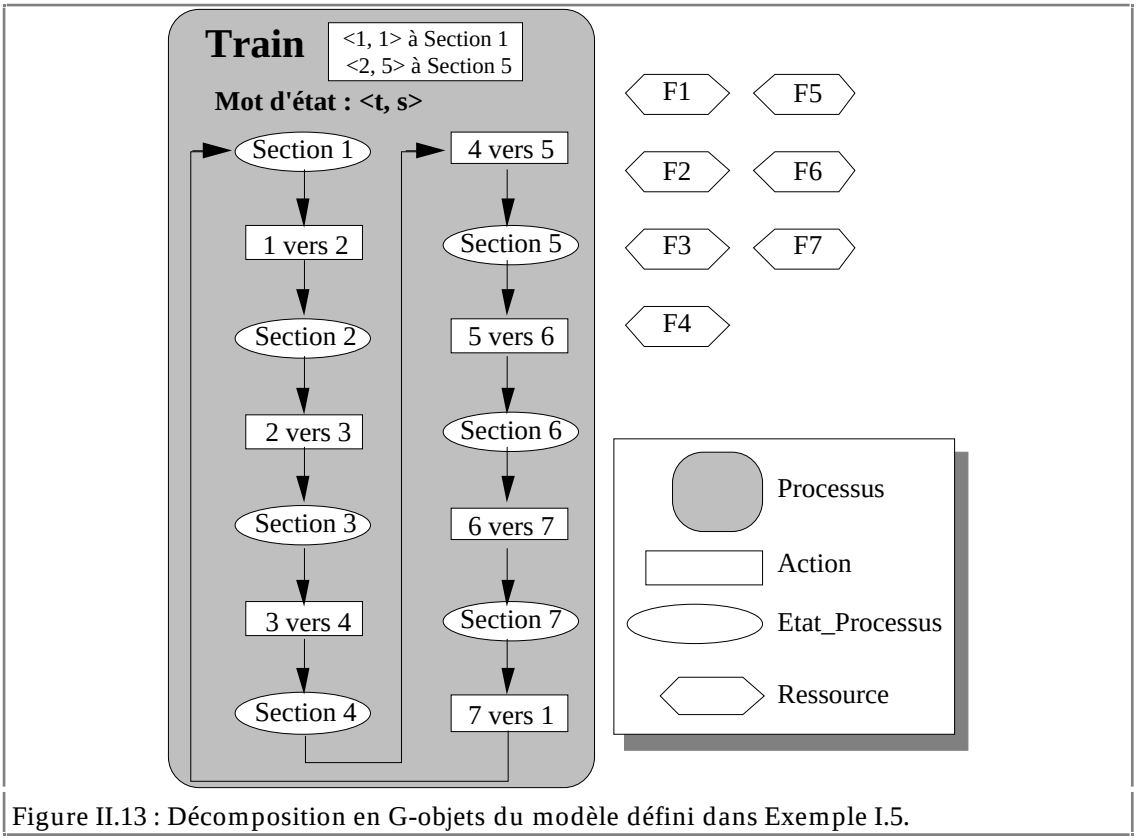


Figure II.13 : Décomposition en G-objets du modèle défini dans Exemple I.5.

Il est probable que le concepteur d'un tel modèle aura envisagé de modéliser le cheminement des trains sur la voie de chemin de fer. Dans ce cas, il choisira la décomposition {2}. Cependant, les autres décompositions peuvent s'avérer intéressantes, si l'on considère un autre point de vue (la gestion de portions de voies par exemple), ou pour une architecture particulière. Considérons que le concepteur du système choisisse la seconde décomposition. Il nomme l'unique Processus "train".

ETAPE 5 :

Nous donnons en Figure II.13 la décomposition en G-objets du modèle pour la décomposition choisie. L'unique Processus ne comporte que des Actions simples et des Etats_Processus simples.

3.2. TRANSFORMATION D'UN MODÈLE EN VUE DE LE RENDRE "PROTOTYPABLE"

Le non respect de l'une des propriétés énoncées entraîne l'échec de l'algorithme de décomposition. En proposant des transformations pour un modèle donné, nous élargissons, dans un premier temps, la classe de réseaux de Petri pour lesquels l'application de l'algorithme de décomposition est possible. Les transformations permettent d'obtenir un modèle équivalent "prototypable".

De telles transformations sont envisageables si les propriétés 1, 2 ou 3 ne sont pas satisfaites. Le non respect de la propriété 4 pose un problème de validation puisqu'elle est liée à la vivacité du réseau de Petri.

3.2.1. Processus instanciés dynamiques

Nous nous intéressons ici à l'échec de la propriété 1, c'est-à-dire que / F_p tel que $\Gamma(F_p) = T$ (Il n'existe pas de sous ensemble F_p de semi-flots couvrant l'ensemble des transitions du modèle structurel).

Le non respect de cette propriété signifie qu'il y a des "suites de places et de transitions pendantes". Nous l'interprétons comme des manipulations dynamiques de processus instanciés au cours du déroulement du système.

Une transformation visant à rendre statique l'existence de processus instanciés est théoriquement possible. Il faut faire apparaître un semi-flot au niveau des "suites de transitions pendantes" en ajoutant des places implicites. Cependant il est impossible de déterminer de façon formelle des règles strictes, applicables dans des conditions données.

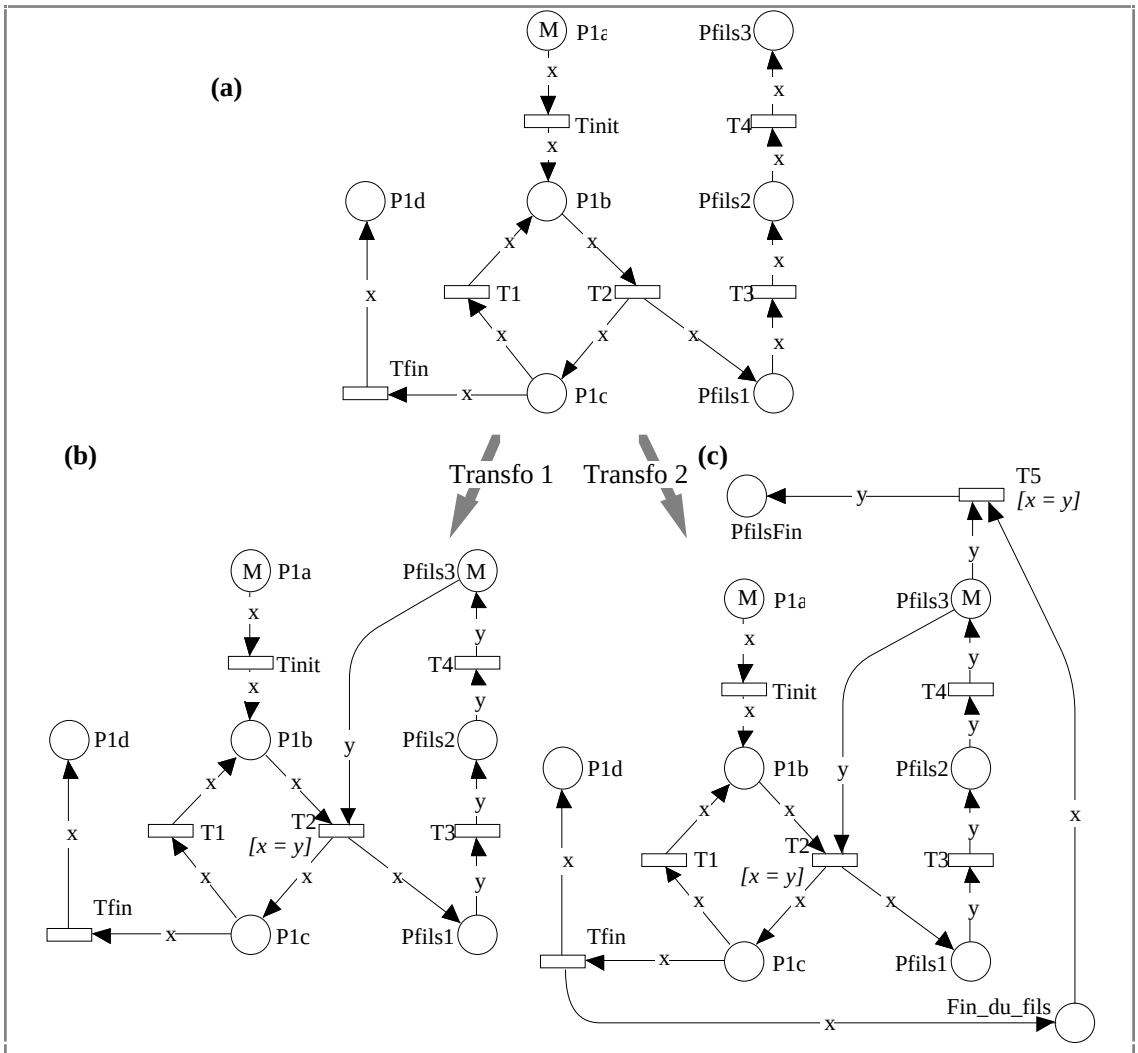


Figure II.14 : Transformation d'un modèle ne vérifiant pas la propriété 1 en vue de le rendre "prototypable".

En effet, pour obtenir un comportement équivalent, il faut souvent modifier la sémantique du réseau de Petri (Exemple II.13). Cette transformation n'est pas automatisable : le concepteur du système doit modifier lui-même son modèle.

Exemple II.13 : Considérons le modèle (a) (Figure II.14) pour lequel la propriété 1 n'est pas vérifiée, puisque l'unique semi-flot du modèle, composé des places *P1a*, *P1b*, *P1c* et *P1d* ne permet pas de couvrir l'ensemble des transitions.
Il est possible d'obtenir automatiquement, via transfo 1, le modèle (b), permettant de découvrir un second Processus défini par les places *Pfils1*, *Pfils2* et *Pfils3*. Ce modèle est "prototypable". Cependant, la sémantique initiale est fortement altérée : (a) peut se terminer (il existe un état d'accueil défini par les places P1d et Pfils3) alors que ce n'est pas le cas de (b).
Le modèle (c) est sémantiquement équivalent au modèles (a) mais le mécanisme de communication ajouté afin de retrouver un état d'accueil correspondant au modèle initial transforme le langage associé : (c) et (a) n'ont plus le même langage; ils ne sont plus équivalents au sens donné par [Colom 86]. Une telle transformation ne peut être effectuée que sur la base de connaissances sémantiques impossibles à déduire des seuls invariants du modèle.

3.2.2. Processus implicites

Nous nous intéressons désormais à l'échec de la propriété 2, c'est-à-dire que nous avons : F_p possible, p_1, p_2 F_p tel que $p_1 \cap p_2 \neq$ (Il n'existe pas de décomposition F_p dont tous les semi-flots qui la composent n'ont aucune place en commun).

Le non respect de cette propriété signifie qu'il existe des Processus implicites : il faut les faire apparaître. Cela est possible en effectuant la transformation suivante :

Transformation T1 :

- **Objectif** : créer des semi-flots couvrant des Processus auparavant implicites
- **Entrée** : un modèle coloré
- **Sortie** : un modèle coloré équivalent au précédent
- **Méthode** : il faut dupliquer l'ensemble des places (Etats_Processus) en commun pour chacun des semi-flots concernés ainsi que les arcs les reliant à leurs voisins. Le marquage de ces places doit être également dupliqué.

Cette transformation conserve les propriétés des modèles. Son seul inconvénient consiste en la création de nombreuses Actions synchronisées, préfigurant un mécanisme plus complexe.

Exemple II.14 : Considérons le modèle de la Figure II.15 pour lequel la propriété 2 n'est pas vérifiée. Le modèle structurel associé au réseau de Petri initial possède deux semi-flots ayant les places P1 et P2 en commun. En dupliquant ces places, le marquage initial qu'elles contiennent et les arcs associés, nous obtenons un modèle transformé équivalent au précédent. Le nouveau modèle comporte 4 semi-flots donnant lieu à deux décompositions duales :
DECOMPOSITION 1 :
- Processus 1 composé des places P1', P2', P3a et P4a,
- Processus 2 composé des places P1, P2, P3b et P4b.

DECOMPOSITION 2 :
- Processus 1 composé des places P1, P2, P3a et P4a,
- Processus 2 composé des places P1', P2', P3b et P4b.

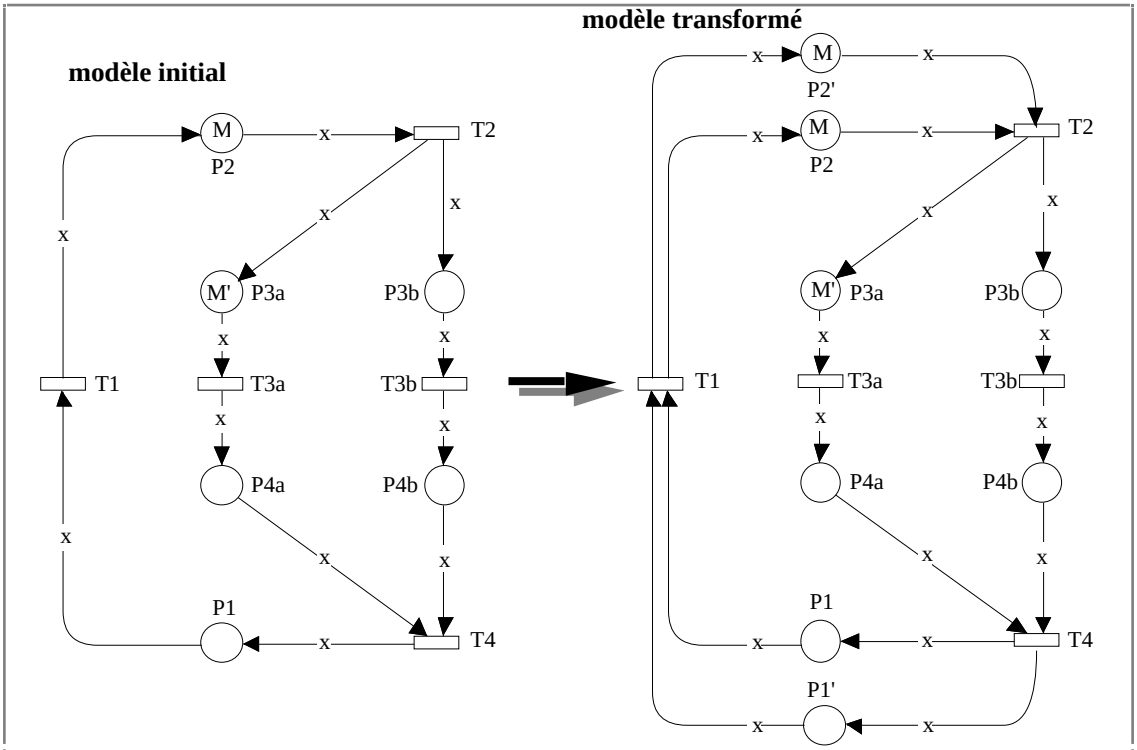


Figure II.15 : Transformation d'un modèle ne vérifiant pas la propriété 2 en vue de le rendre "prototypable".

Le non respect de la propriété 3 peut également s'expliquer par l'existence de Processus implicites. Cela est assimilable à un regroupement de plusieurs Processus.

Dans certains cas, une transformation peut être effectuée. Il s'agit toujours de faire apparaître des Processus se synchronisant.

Transformation T2 :

- **Objectif** : "dégrouper" des Processus et créer des semi-flots les caractérisant
- **Entrée** : un modèle coloré
- **Sortie** : un modèle coloré équivalent au précédent
- **Méthode** : iL faut dupliquer certaines places et transitions support du semi-flot concerné. La duplication est effectuée :
  - pour la séquence "en amont" de la transition t, si l'arc ne vérifiant pas la propriété 3 est en précondition de celle-ci;
  - pour la séquence "en aval" de la transition t, si l'arc ne vérifiant pas la propriété 3 est en postcondition de celle-ci;

  Les séquences sont dupliquées en fonction de la valuation, dans le modèle structurel, de l'arc fautif. Une fois la transformation effectuée, le modèle obtenu provoque l'échec de la propriété 2; il faut donc appliquer T1.

  Le groupe de transformations (T2 puis T1) doit être appliqué jusqu'à ce que le modèle respecte la propriété 3.

Exemple II.15 : Considérons le modèle (a) de la Figure II.16 pour lequel la propriété 3 n'est pas vérifiée. Une première transformation aboutit au modèle (b), pour lequel la propriété 2 n'est pas respectée. L'application de la transformation T1 nous permet

d'obtenir le modèle (c) qui est "prototypable". Les mêmes traitements sont associés aux transitions T1 et T1'.

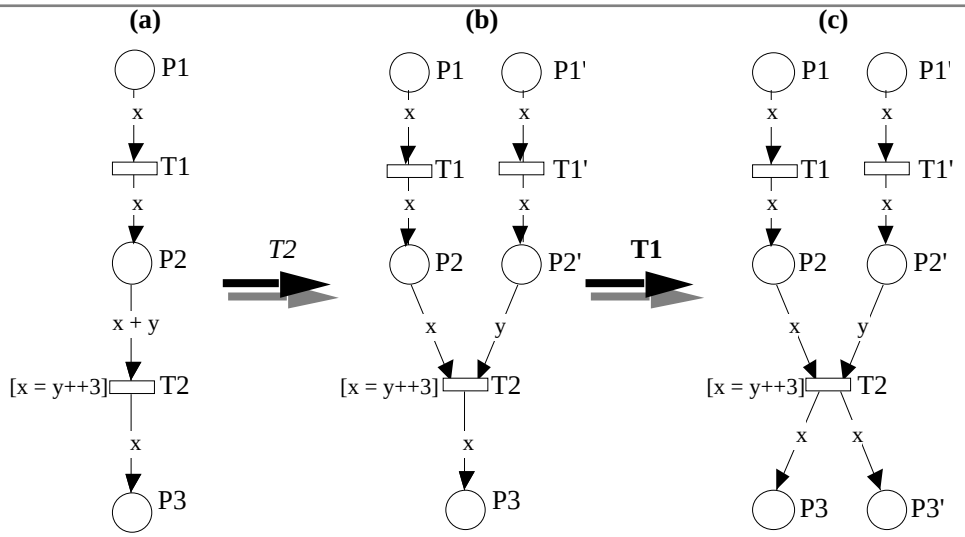


Figure II.16 : Exemple de modèle pour lequel le non respect de la propriété 3 est assimilée à l'existence de Processus implicites.

Une telle transformation n'est cependant pas automatisable :

- son application n'est pas systématique : elle est par exemple inapplicable si l'arc ne respectant pas la propriété 3 se trouve dans une itération (Figure II.17).
- elle peut aboutir à un modèle équivalent inextricable : c'est en particulier le cas si plusieurs arcs sont valués dans une séquence (Exemple II.16).

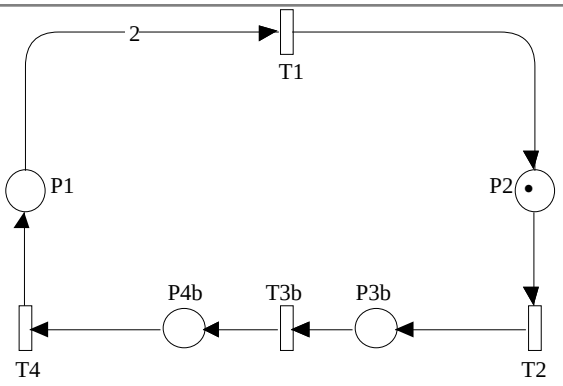


Figure II.17 : Exemple de modèle ne respectant pas la propriété 3 pour lequel la transformation T2 n'est pas applicable.

Exemple II.16 : Considérons le modèle initial de la Figure II.18 pour lequel l'application successive des transformations exposées ci-dessus donne un modèle "prototypable". A partir d'un modèle relativement simple, nous obtenons un réseau de Petri complexe dont la décomposition donnera un grand nombre d'Actions synchronisées. Une meilleure connaissance de la sémantique associée au modèle (difficile à obtenir à l'aide des seuls semi-flots) permettrait sûrement des modifications plus efficaces.

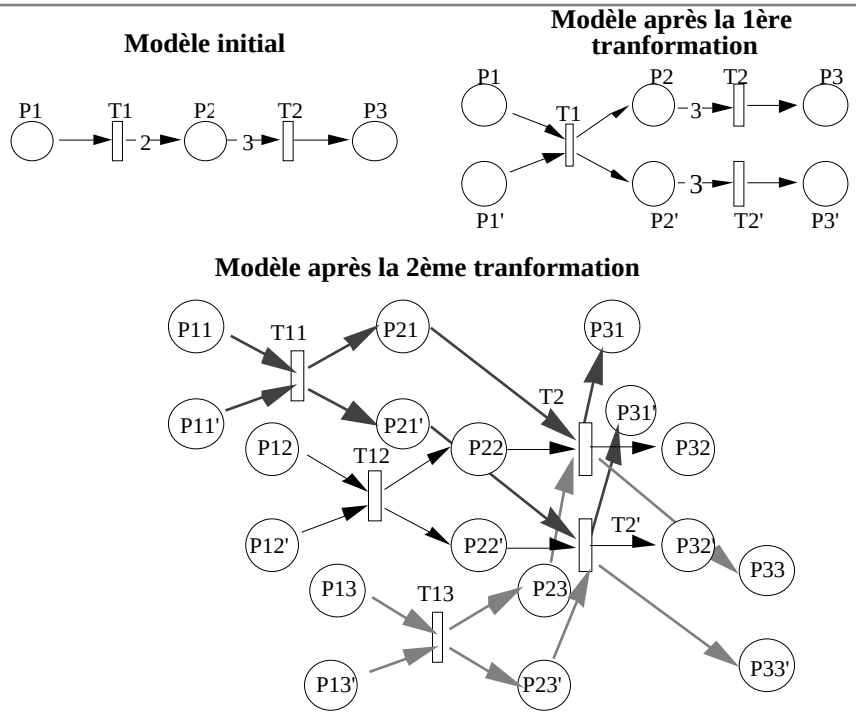


Figure II.18 : Exemple de modèle pour lequel des applications successives de la transformation T2 donnent un résultat complexe.

3.3. VERS UNE EXTENSION DE L'ALGORITHME DE DÉCOMPOSITION EN PROCESSUS

Les transformations que nous avons proposées dans la section précédente visent à étendre la classe des réseaux de Petri pour lesquels le prototypage est possible.

L'utilisation d'une transformation automatisable permet de rendre vérifiable la propriété 2. En cas d'échec des propriétés 1 ou 3, les transformations nécessitent une connaissance sémantique du modèle qu'il est difficile d'obtenir à l'aide des invariants.

Dans cette section, nous nous attachons à étendre la classe des réseaux de Petri "prototypables" en intégrant de nouvelles caractéristiques à certains G-objets.

3.3.1. Interprétation de l'échec des propriétés 1 et 2

Pour proposer la transformation T1, nous avons interprété l'échec de la propriété 2 par l'existence de Processus implicites (section 3.2.2). Elle peut cependant, comme pour l'échec de la propriété 1, s'expliquer par une modification dynamique du nombre de processus instanciés (section 3.1.1).

Pour générer automatiquement le programme associé à de tels modèles, il faut identifier (dans le modèle) et gérer (dans le prototype) les manipulations dynamiques d'instances de Processus.

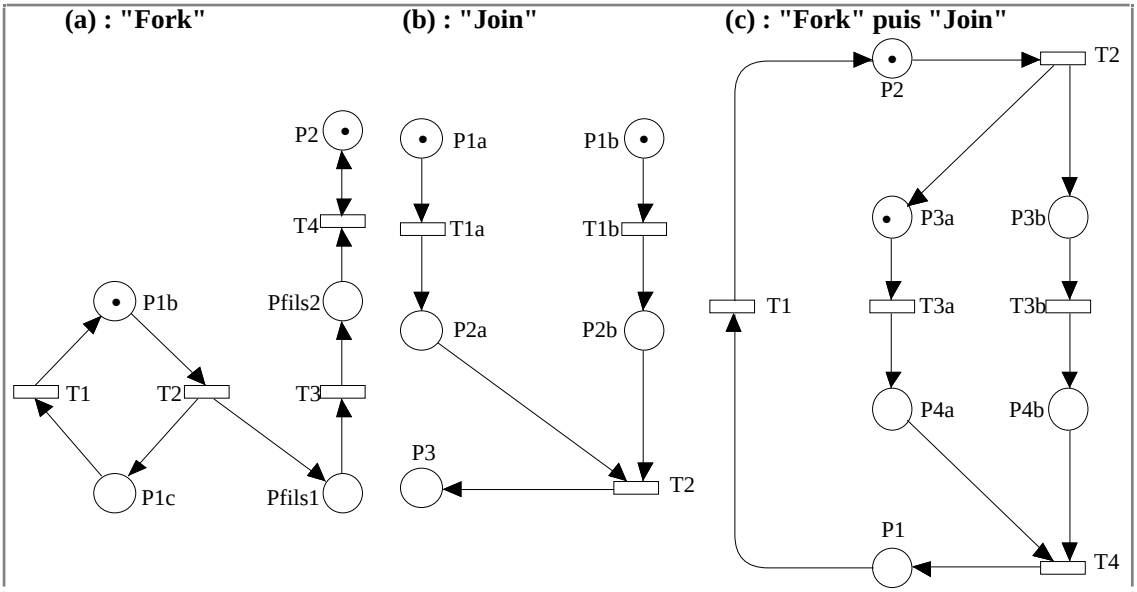


Figure II.19 : Interprétation de modèles ne respectant pas la propriété 2.

Exemple II.17: Considérons les modèles de la Figure II.19. Nous pouvons détecter des manipulations de processus instanciés (création ou disparition) :

- Dans (a), la transition *T1* correspond à un "fork" : il y a création de processus instanciés. Ce modèle ne respecte pas la propriété 1.
- Dans (b), la transition *T2* correspond à un "join". Deux processus instanciés se rejoignent et l'un d'eux disparaît. Ce modèle ne respecte pas la propriété 2.
- Dans (c), *T2* correspond à un "fork" et *T4* à un "join". Un processus crée dynamiquement des processus instanciés d'un autre type, chargés de réaliser un service en parallèle, puis de fournir un résultat. Ce modèle ne vérifie pas la propriété 2.

Le problème doit être étudié au niveau des transitions. Il faut caractériser les créations ("fork" de type UNIX) et les disparitions ("join") dynamiques d'instances de Processus. De telles configurations sont détectables :

- Non respect de la propriété 2 : il faut choisir l'un des semi-flots et en déduire le Processus "maître". Les autres semi-flots, diminués des places qu'ils avaient en commun avec celui qui a été choisi, décrivent les Processus "esclaves" (créés dynamiquement ou disparaissant juste après la synchronisation). Si la transition liée à la jonction des Processus possède plus d'Etats_Processus en sortie qu'en entrée, c'est un "fork". Dans le cas contraire, c'est un "join".

  Dans le cas où il serait impossible de trouver un Processus "maître", la transformation T1 reste applicable.
- Non respect de la propriété 1 : il faut examiner les séquences de transitions isolées et trouver le Processus "maître" auquel elles sont reliées. On peut ainsi définir des Processus sans passer par la notion de semi-flot. La différence entre le nombre d'Etats_Processus en entrée et en sortie permet, comme précédemment, de détecter un "join" ou un "fork".

Les Processus ainsi calculés doivent toujours respecter la propriété 3. Par contre, les Processus "esclaves" créés dynamiquement (à partir d'un "fork") peuvent s'affranchir de la propriété 4.

Pour créer dynamiquement des Processus (“fork”), ou accepter des synchronisations avec disparition (“join”), nous différencions les catégories d’Actions qui sont liées à de telles opérations. Les Actions possèdent alors les attributs suivants :

- simple ou synchronisée (Définition II.4),
- fork,
- join.

***Définition II.12 :* Action fork et Action join**

Une Action fork possède plus d’Etats_Processus en sortie qu’en entrée. Chaque Etat_Processus en sortie doit être lié à un Processus différent.
Une Action join possède plus d’Etats_Processus en entrée qu’en sortie. Chaque Etat_Processus en entrée doit être lié à un Processus différent.

Les Actions fork et join peuvent, tout comme les Actions simples et synchronisées, être gardées ou non. Les Actions fork et les Actions join ne sont pas autre chose que des demi-synchronisations. Certains des participants n’existent plus avant ou après la réalisation de l’Action. Les Processus qui sont précondition et postcondition d’une Action fork ou d’une Action join ont un statut particulier.

***Définition II.13 :* Processus pilote pour une Action fork ou join**

Nous appelons Processus pilotes pour une Action fork ou join, les Processus participant à une Action fork ou join pour lesquels il existe un Etat_Processus précondition et un Etat_Processus postcondition pour l’Action considérée. Tous les participants d’une Action synchronisée sont des pilotes pour cette Action.

Exemple II.18 : Dans le modèle (a) (Figure II.20), le Processus de gauche est pilote pour *T2* : il peut être considéré comme l’initiateur du “fork”.
Dans le modèle (b), représentant une Action join entre trois Processus, il existe deux Processus pilotes pour *T4* : la disparition des instances du Processus central, après exécution de *T4*, est soumise à l’état des instances des deux autres Processus.

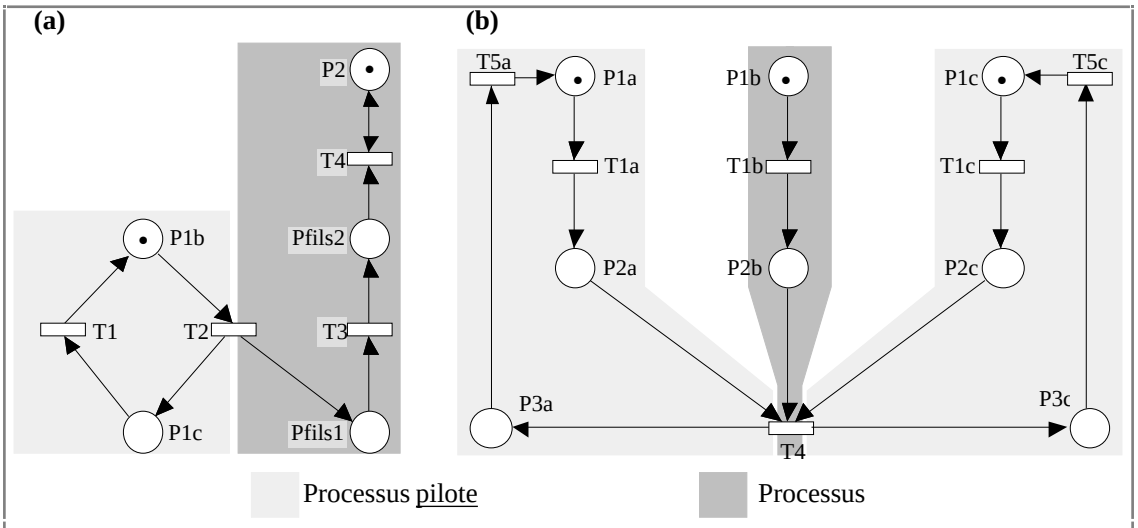


Figure II.20 : Exemple de Processus pilotes pour des Actions fork ou join.

Nous définissons, de façon informelle, une méthode de décomposition en Processus intégrant la manipulation dynamique d'instances.

Exemple II.19 : Considérons l'exemple de la Figure II.21. Nous avons la famille génératrice de semi-flots de support minimal F suivante :

1 í P1 + P2 + P3b.
2 í P2 + P3a + P4.
3 í P6b + P7 + P8.

F ne couvre pas l'ensemble des transitions du modèle (non respect de la propriété 1). Du semi-flot 3, nous pouvons dégager un Processus. Les semi-flots 1 et 2 ont P2 en commun (non respect de la propriété 2).

Il existe une séquence de transitions pendantes (*T5* et *T6*) reliées par *P5*. En entrée de *T5*, il existe une place précondition, elle-même issue d'une transition - *T2* - où apparaît une création dynamique de Processus. De l'autre côté, la séquence est reliée à *T8* : il s'agit d'un "join". Nous pouvons déduire un second Processus défini par les places *P3c*, *P5* et *P6a*. Ce Processus est créé dynamiquement.

Intéressons-nous désormais aux Processus décrits par 1 et 2 et recherchons le "maître". Il peut s'agir indifféremment du Processus décrit par 1 ou de celui décrit par 2. Nous choisissons 1. L'autre Processus sera décrit par les places de 2 diminuées de celles qu'il a en commun avec 1 (ici, *P2*).

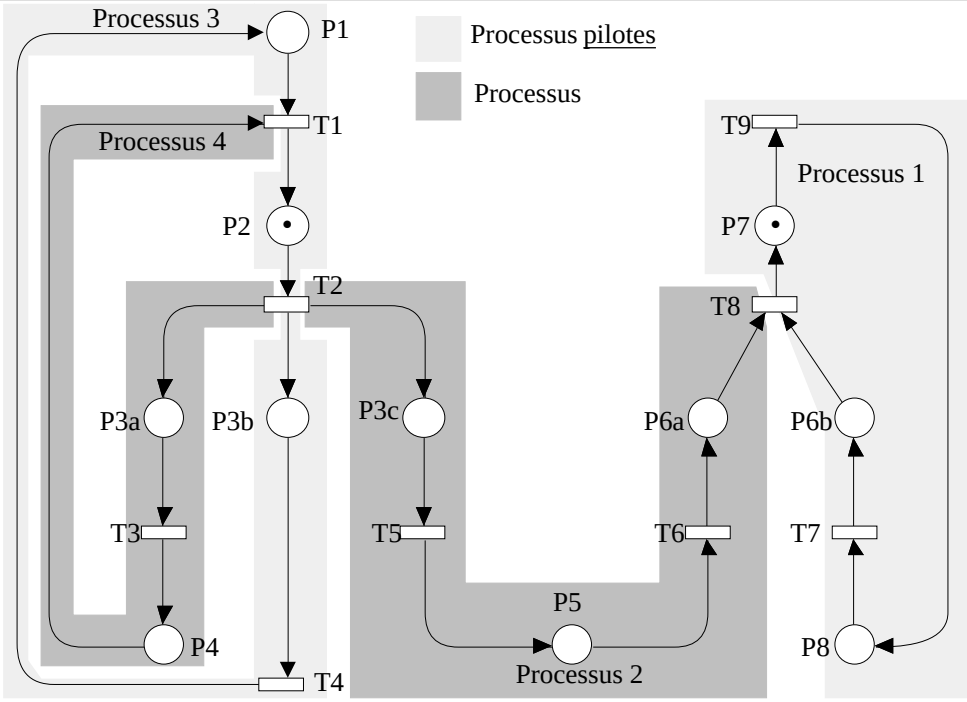


Figure II.21 : Exemple de modèle ne respectant pas les propriétés 1 et 2 pour lequel une décomposition est néanmoins caractérisable.

Nous obtenons finalement la décomposition suivante :

Processus 1 :
Etats_Processus : *P6b*, *P7* et *P8* (simples)
Actions : *T7*, *T9* (simples) et *T8* (join); Processus 1 est pilote pour *T8*
Une seule instance commençant en *P7*, pas de mot d'état.

Processus 2 :
Etats_Processus : *P3c*, *P5* et *P6a* (simples)
Actions : *T5*, *T6* (simples), *T2* (fork) et T8 (join)
Pas d'instance (Processus dynamique), démarrage en *P3c*, pas de mot d'état.

Processus 3 :
Etats_Processus : *P1*, *P2* et *P3b* (simples)
Actions : *T4* (simple), *T2* (fork) et *T1* (join); Processus 3 est pilote pour *T1* et *T2*
Une seule instance commençant en *P2*, pas de mot d'état.

Processus 4 :
Etats_Processus : *P3a* et *P3b* (simples)
Actions : *T3* (simple), *T2* (fork) et *T1* (join)
Pas d'instances (Processus dynamique), démarrage en *P3a*, pas de mot d'état.

L'extension n'est pas encore parfaite, il existe encore des modèles dont la décomposition en Processus ne peut être obtenue automatiquement, même lorsque l'on considère la manipulation dynamique d'instances.

Le problème réside en la détermination du Processus "maître". Lorsque les propriétés 1 et 2 ne sont pas respectées, il se peut qu'une analyse des semi-flots et des séquences de transitions non couvertes ne suffise pas, alors qu'il existe une décomposition. Il faut mieux connaître la sémantique du modèle. Nous donnons un exemple de ce type en Figure II.22.

Exemple II.20 : Le modèle de la Figure II.22 possède la famille génératrice de semi-flots de support minimale F suivante :
1 í P1 + P2 + P3b + P4b.
2 í P2 + P3a + P4a.
3 í P5 + P6b + P7.
F ne couvre pas l'ensemble des transitions du modèle (échec de la propriété 1). Les semi-flots 1 et 2 ont P2 en commun (échec de la propriété 2). Du semi-flot 3, nous pouvons dégager un Processus.
Il existe une séquence de transitions pendantes (*T5* et *T6*) reliées par *P4c*. En entrée de *T5*, il existe une place précondition, elle-même issue d'une transition - *T2* - où apparaît une création dynamique de Processus. De l'autre côté, la séquence est reliée à *T8* : il s'agit d'un join. Nous pouvons déduire un second Processus défini par les places *P3c*, *P4c* et *P6a*. Ce Processus est créé dynamiquement.
Il faut choisir, entre les deux semi-flots restants, celui qui définit le Processus "maître". Il ne peut s'agir de 2, à cause du rebouclage du Processus déduit de 1. Mais si 1 décrit le Processus "maître", il n'est pas instancié. Nous devons donc, conformément à l'algorithme, effectuer la transformation T1 et dupliquer *P2* (la seule place que les deux semi-flots ont en commun). Les semi-flots 1 et 2 deviennent donc :
A í P1 + P2 + P3b + P4b.
B í P1 + P2 '+ P3b + P4b.
C í P2 + P3a + P4a.
D í P2' + P3a + P4a.
Il nous faut cependant éliminer C et D qui ne vérifient pas la propriété 4. Ce modèle appartient à la classe des réseaux "non-prototypables".
Une solution triviale existe cependant. Il suffirait de considérer le Processus "maître"défini par *P1 + P2 + P3a + P4a* et un Processus "esclave" créé dynamiquement, défini par *P3b* et *P4b*. Nous obtenons la décomposition suivante :
Processus 1 :
Etats_Processus : *P1*, *P2*, *P3a*, *P4a*
Actions : *T1*, *T3* (simples) *T2* (fork)
Une seule instance commençant en *P1* , pas de mot d'état
Processus 2 :
Etats_Processus : *P3b*, *P4b*
Actions : *T4* (simple)
pas d'instance (Processus dynamique), pas de mot d'état, démarrage en *P3b*.
Processus 3 :
Etats_Processus : *P3c*, *P4c*, *P6a*
Actions : T5, *T6* (simples) *T2* (fork), *T8* (join)
aucune instance (Processus dynamique), pas de mot d'état, démarrage en *P3c*.
Processus 4 :
Etats_Processus : *P5*, *P6b*, *P7*

Actions : *T7*, *T9* (simples) *T8* (join)
Une instance démarrant en P5, pas de mot d'état.

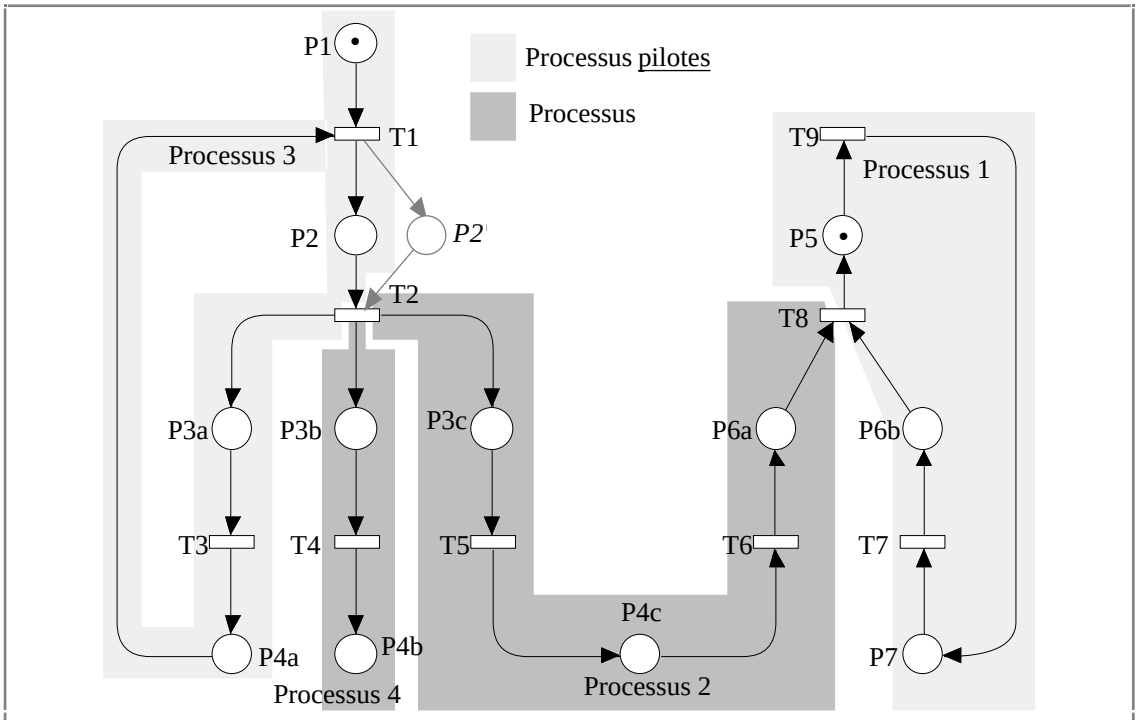


Figure II.22 : Exemple de modèle pour lequel la décomposition en Processus, triviale, ne peut être obtenue simplement à l'aide des seuls semi-flots.

Cette décomposition ne peut cependant être déduite trivialement de la famille génératrice de semi-flots de support minimal (combinaison de places extraites des semi-flots 1 et 2). Il nous faut des connaissances sur la sémantique du modèle.

3.3.2. Vers une interprétation de l'échec de la propriété 3

Considérons, dans un premier temps, les modèles pour lesquels tous les Processus ne vérifiant pas la propriété 3 respectent les contraintes suivantes :

- í t pour laquelle p pre (t) tel que |val (t, p)| > 1, alors :

  ! p2 post (t) et support du même semi-flot tel que |val (p, t)| = |val (t, p2)| = N

- í t pour laquelle p post (t) tel que |val (t, p)| > 1, alors :

  ! p2 pre (t) et support du même semi-flot tel que |val (t, p)| = |val (p2, t)| = N

Cela revient à considérer que N processus instanciés se synchronisent.

En étendant la notion d'Action synchronisée, nous pouvons définir des catégories d'Actions synchronisées permettant l'analyse d'un tel modèle.

***Définition II.14* : Action synchronisée locale, différenciée et hybride**

Une Action synchronisée locale n'implique qu'un seul Processus.
Une Action synchronisée différenciée implique des Processus différent.s
Ce sont les Actions synchronisées déjà évoquées dans Définition II.4.
Une Action synchronisée hybride est à la fois locale et différenciée.

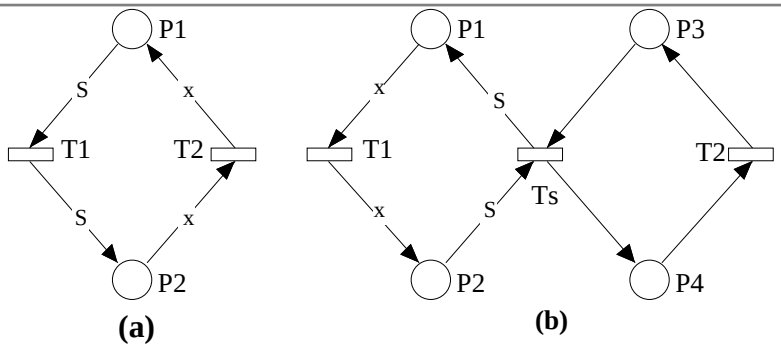


Figure II.23 : Modèles ne respectant pas la propriété 3 mais pour lesquels une décomposition est possible si l'on considère des Actions synchronisées locales ou hybrides.

Exemple II.21 : Etudions les exemples donnés en Figure II.23.

Dans (a), l'échec de la propriété 3 est lié à l'existence d'une synchronisation entre N instances de l'unique Processus du modèle. Chacune des processus instancié se synchronisant doit avoir une couleur différente. Si nous considérons que des Actions synchronisées peuvent avoir lieu entre plusieurs processus instanciés d'un même type, l'interprétation de ce modèle est possible. *T1* est une Action synchronisée locale.

(b) comporte deux processus définis par *P1-P2* et *P3-P4*. L'Action *Ts* est synchronisée. Elle implique d'un côté N processus instanciés du type défini par *P1-P2* et un processus instancié du type défini *P3-P4*. *Ts* est une Action synchronisée hybride.

3.3.3. Nouvelle classification des Actions

Nous avons, d'une part, caractérisé les Actions fork et join. Cela nous permet de décomposer des réseaux de Petri provoquant l'échec des propriétés 1 et 2.

Nous avons également défini les Actions synchronisées locales, différenciées et hybrides. Cela nous permet de décomposer certains réseaux de Petri ne respectant pas la propriété 3. Ces attributs ne sont pas applicables qu'aux seules Actions synchronisées. Nous pouvons également considérer que des Actions fork ou join sont locales, différenciées ou hybrides.

***Définition II.15* : Action fork locale, différenciée et hybride**

Une Action fork locale n'implique qu'un seul Processus.
Une Action fork différenciée implique des Processus différents. Ce sont les Actions fork déjà évoquées dans Définition II.12.
Une Action fork hybride est à la fois locale et différenciée.

***Définition II.16* : Action join locale, différenciée et hybride**

Une Action join locale n'implique qu'un seul Processus.
Une Action join différenciée implique des Processus différents. Ce sont les Actions join déjà évoquées dans Définition II.13.
Un Action join hybride est à la fois locale et différenciée.

Nous obtenons alors la classification suivante :

non gardée	**locale**	**différenciée**	**hybride**
Action simple	**IMPOSSIBLE**	**IMPOSSIBLE**	**IMPOSSIBLE**
Action synchronisée	*Action synchronisée locale*	*Action synchronisée différenciée*	*Action synchronisée hybride*
Action fork	*Action fork locale*	*Action fork différenciée*	*Action fork hybride*
Action join	*Action join locale*	*Action join différenciée*	*Action join hybride*
gardée	**locale**	**différenciée**	**hybride**
Action simple gardée	**IMPOSSIBLE**	**IMPOSSIBLE**	**IMPOSSIBLE**
Action synchronisée gardée	*Action synchronisée locale gardée*	*Action synchronisée différenciée gardée*	*Action synchronisée hybride gardée*
Action fork gardée	*Action fork locale gardée*	*Action fork différenciée gardée*	*Action fork hybride gardée*
Action join gardée	*Action join locale gardée*	*Action join différenciée gardée*	*Action join hybride gardée*

Figure II.24 : Nouvelle classification des Actions.

4. Synthèse

Dans ce chapitre, nous avons défini un algorithme de partitionnement d'un réseau de Petri en Processus, Etats_Processus, Actions et Ressources. Cet algorithme fonctionne pour les "modèles prototypables", qui doivent respecter quatre propriétés.

Les modèles pour lesquels nous ne pouvons appliquer directement l'algorithme se divisent en deux classes : les modèles "transformables" et les modèles "non-prototypables" (Figure II.25).

Le modèle est dit "prototypable" lorsque toutes les propriétés sont vérifiées. Il est dit "transformable" lorsque les propriétés 1, 2 ou 3 ne sont pas respectées. Dans le cas ou l'échec porte sur la propriété 2, il est possible d'obtenir automatiquement un modèle "prototypable" équivalent. Nous ne pouvons que diagnostiquer le non respect des autres propriétés.

Les réseaux de Petri ne vérifiant pas la propriété 4 ne peuvent être analysés.

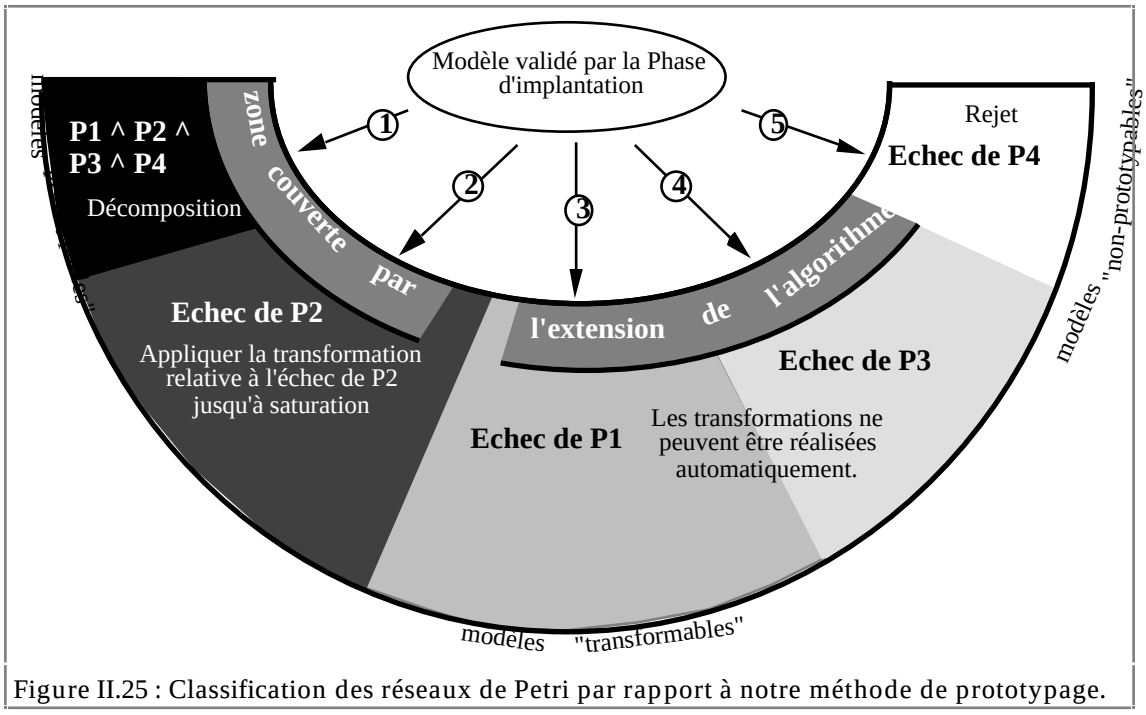


Figure II.25 : Classification des réseaux de Petri par rapport à notre méthode de prototypage.

Nous proposons également d'élargir la classe des "modèles prototypables" en détectant la manipulation dynamique d'instances de Processus. Un certain nombre de modèles pour lesquels les propriétés 1, 2 et 3 ne sont pas respectées deviennent ainsi "prototypables".

Il est probable que l'étude d'autres propriétés (invariants de transitions...) des réseaux de Petri permettra de résoudre ces lacunes. La théorie des réseaux de Petri évolue d'ailleurs dans ce sens [Couvreur 90a, Couvreur 90b, Chehaibar 91, Dutheillet 91, Petrucci 91, Haddad 91].

		Introduction générale
	Chapitre I	: *Prototypage de systèmes parallèles*
	Chapitre II	: *Interprétation sémantique d'un réseau de Petri*
+	Chapitre III	: *Méthodologie de prototypage*
	Chapitre IV	: *Structure du prototype généré*
	Chapitre V	: *Caractéristiques du prototype Ada*
	Chapitre VI	: *Le prototype centralisé*
	Chapitre VII	: *Le prototype réparti*
	Chapitre VIII	: *CPN/TAGADA : L'outil de prototypage*
		Conclusion générale
		Bibliographie
		Annexes & Appendices

Chapitre III :

Méthodologie de Prototypage

1. Introduction 103
2. Notion de composant externe 105
   - 2.1. Abstraction d'un composant externe....105
   - 2.2. Les composants externes dans un modèle....110
   - 2.3. La notion de service....111
   - 2.4. Composants externes réversibles et non-réversibles....112
3. Les étapes du prototypage 115
   - 3.1. Décomposition d'un modèle....116
   - 3.2. Production du prototype....118
4. Conclusion 122

1. Introduction

Ce chapitre décrit l'approche méthodologie permettant d'assurer la traduction de la spécification d'un système complet (modèle fermé) en un prototype exécutable.

L'introduction, dans la section 2, de la notion de *composant externe*, permet de présenter les notions de services et de modules réutilisables. La section 3 détaille les différentes étapes de notre méthodologie.

Notre méthodologie repose sur une étude préalable des éléments de spécification d'un système. Conformément aux principes de construction d'un modèle fermé (Définition I.2), la spécification d'un système se décompose en deux parties :

- le système en lui-même (modèle ouvert),
- l'environnement avec lequel il interagit (spécification d'une abstraction du "monde extérieur").

Naturellement, le prototypage d'un système ne doit s'appliquer que sur le modèle ouvert (Définition I.3). Le prototype obtenu doit s'interfacer correctement avec un environnement qui existe déjà.

Exemple III.1 : On s'intéresse à la réalisation d'un auto-commutateur téléphonique (Figure III.1). Le modèle décrivant les traitements à effectuer doit intégrer le comportement des abonnés. Ceci permet de valider l'auto-commutateur en fonction de toutes les réactions possibles de la part des usagers.
Le prototype du modèle ainsi défini ne doit pas intégrer le comportement de l'abonné qui existe déjà. Il faudra cependant "stimuler" le prototype en fonction des possibilités définies pour l'usager dans le modèle. Des interfaces devront être intégrées de façon à relier le prototype au monde extérieur (les abonnés).

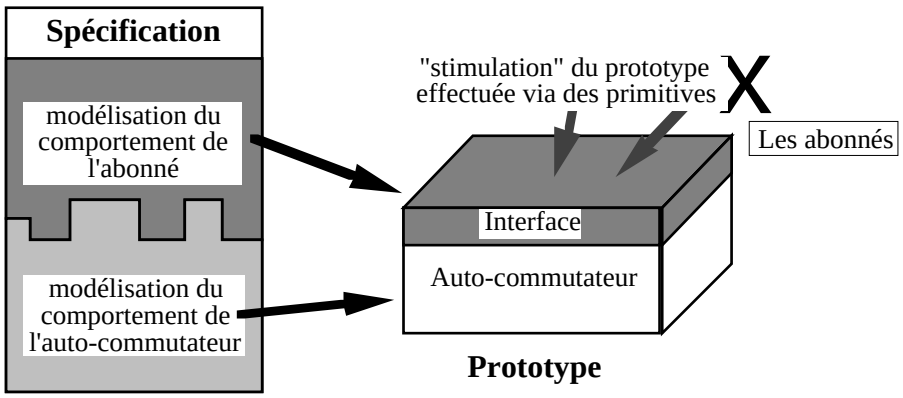


Figure III.1 : Le passage d'un modèle contenant la description de l'environnement d'un système au prototype correspondant.

Lorsque les modèles deviennent trop importants, il est nécessaire de mettre en œuvre une conception modulaire. Sous certaines conditions [Souissi 89, Souissi 90, Chehaibar 91], il est possible de séparer et de valider séparément un ensemble de modèles, exprimés à l'aide de réseaux de Petri, décrivant un système.

L'analogie avec la technique de compilation séparée est évidente. L'utilisation appropriée de cette notion permet de diviser un modèle en modules, éventuellement réutilisables, plus simples à réaliser et à tester.

Dans le cadre des réseaux de Petri, l'utilisation d'une telle technique permet une spécification modulaire : chaque module voit ses homologues comme autant de parties d'environnement. Le prototype sera généré par modules qui seront assemblés dans une phase "d'édition de liens".

Exemple III.2 : Considérons une application divisée en trois modules : M1, M2 et M3 (Figure III.2). M1 est lié au module M2, M3 au module M2 et M2 aux deux autres.
Nous distinguons trois étapes :

1 la conception et la *validation* de chaque module, les autres modules étant considérés comme des "parties d'environnement";
2 un prototype correspondant à chaque module est généré avec des interfaces permettant "d'accéder" aux modules dont il utilise les services;
3 un exécutable est produit par "édition de liens" des trois modules de l'application.

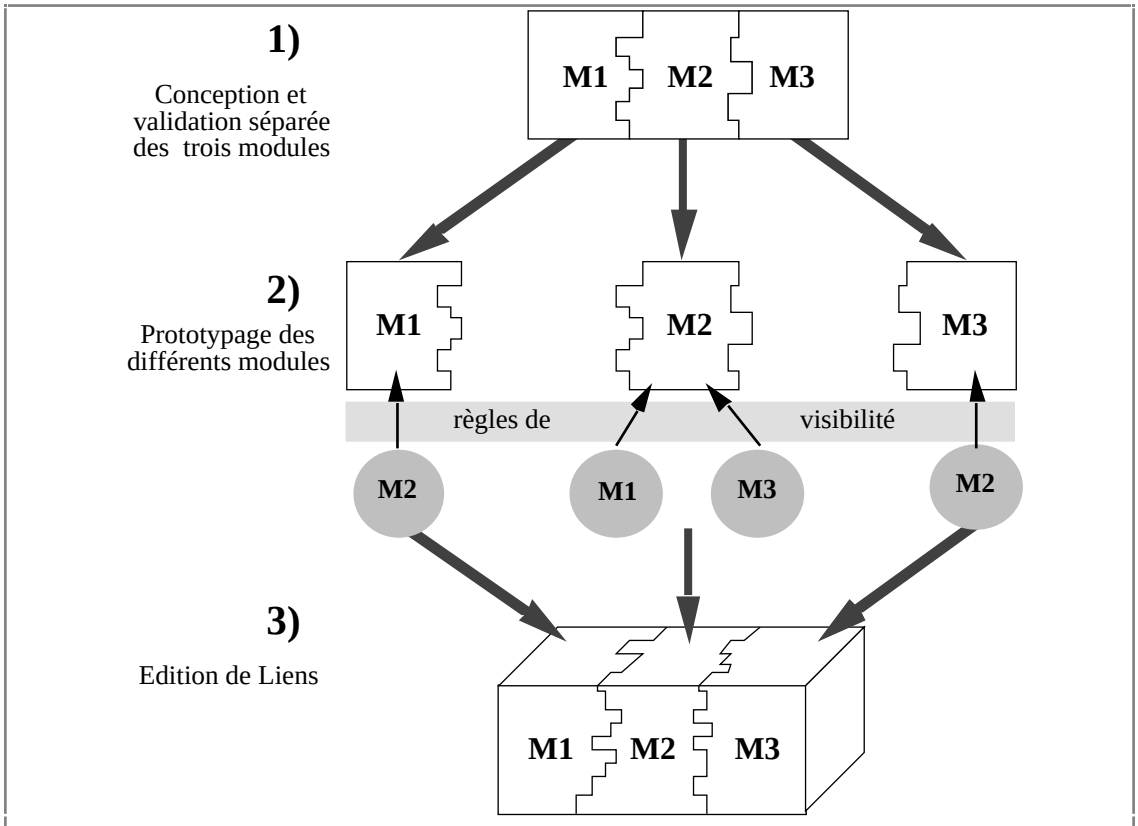


Figure III.2 : Conception modulaire d'une application et obtention du prototype associé.

Il est nécessaire de pouvoir introduire des modèles d'environnement dans la spécification d'un système. Il faut cependant formaliser cette opération afin :

- de permettre une validation cohérente du comportement du système,
- de permettre la génération d'un prototype relié à son environnement.

A cette fin, nous définissons les composants externes que nous décrivons dans la section suivante. Nous nous intéressons ensuite, en section 3, aux étapes du prototypage automatique d'un système spécifié à l'aide de réseaux de Petri.

2. Notion de composant externe

Le chapitre précédent a décrit la technique de décomposition en G-objets d'un modèle ouvert. Nous présentons maintenant une technique adaptée à la représentation de son environnement. A cette fin, nous introduisons la notion de composant externe.

***Définition III.1 :* Composant externe**

Un composant externe représente une portion de l'environnement qui agit sur le modèle ouvert.

Nous avons repris le principe des unités de modélisation présenté dans [Bachatène 92]. Elles sont caractérisées par deux types de services :

- les services offerts, disponibles pour d'autres modules;
- les services requis, qui doivent être offerts par d'autres modules.

Nous appliquons cette démarche pour la représentation de l'environnement. Ainsi, un composant externe constitue une unité de modélisation spécialisée dans la représentation de l'environnement.

Par construction, un composant externe est réutilisable. Il s'intègre naturellement dans une bibliothèque de composants externes correspondant à des "morceaux d'environnement types" souvent utilisés (clavier, écran, système de gestion de fichiers...).

La notion de composant externe est une solution aux problèmes évoqués dans Exemple III.1 et Exemple III.2. Elle autorise une approche objet [Booch 86, Booch 87, Stroustrup 87, Coad 92].

Exemple III.3 : On s'intéresse à la réalisation d'un auto-commutateur téléphonique déjà évoquée dans Exemple III.1. Le modèle décrivant les traitements à effectuer est relié à un composant externe : l'abonné.
L'abonné peut "stimuler" l'auto-commutateur (décrocher le téléphone, composer un numéro...) qui lui transmet des informations (état de la ligne du correspondant...). Le prototypage de l'auto-commutateur doit exclure le composant externe qui est intégré par le biais de primitives de manipulation.

Exemple III.4 : Considérons l'application divisée en modules déjà évoquée dans Exemple III.2. Chaque module voit ses homologues comme autant de composants externes :

1 M2 est l'unique composant externe utilisé par M1 et M3. Ce composant externe offre des services qui sont utilisés par M1 et M3.
2 Deux composants externes représentant M1 et M3 sont utilisés par le modèle de M2.

Les trois "morceaux de prototype" ainsi obtenus sont associés dans une phase d'édition de liens en vue d'obtenir un exécutable.

2.1. ABSTRACTION D'UN COMPOSANT EXTERNE

Un composant externe est défini à l'aide d'une *abstraction*.

***Définition III.2 :* Abstraction d'un composant externe**

L'abstraction d'un composant externe regroupe l'ensemble des informations permettant de le décrire. Elle se compose de deux parties :

- la **formalisation**, regroupant les informations formelles permettant la validation et l'identification du composant;
- la **description interne** regroupant les données nécessaires à l'écriture du prototype.

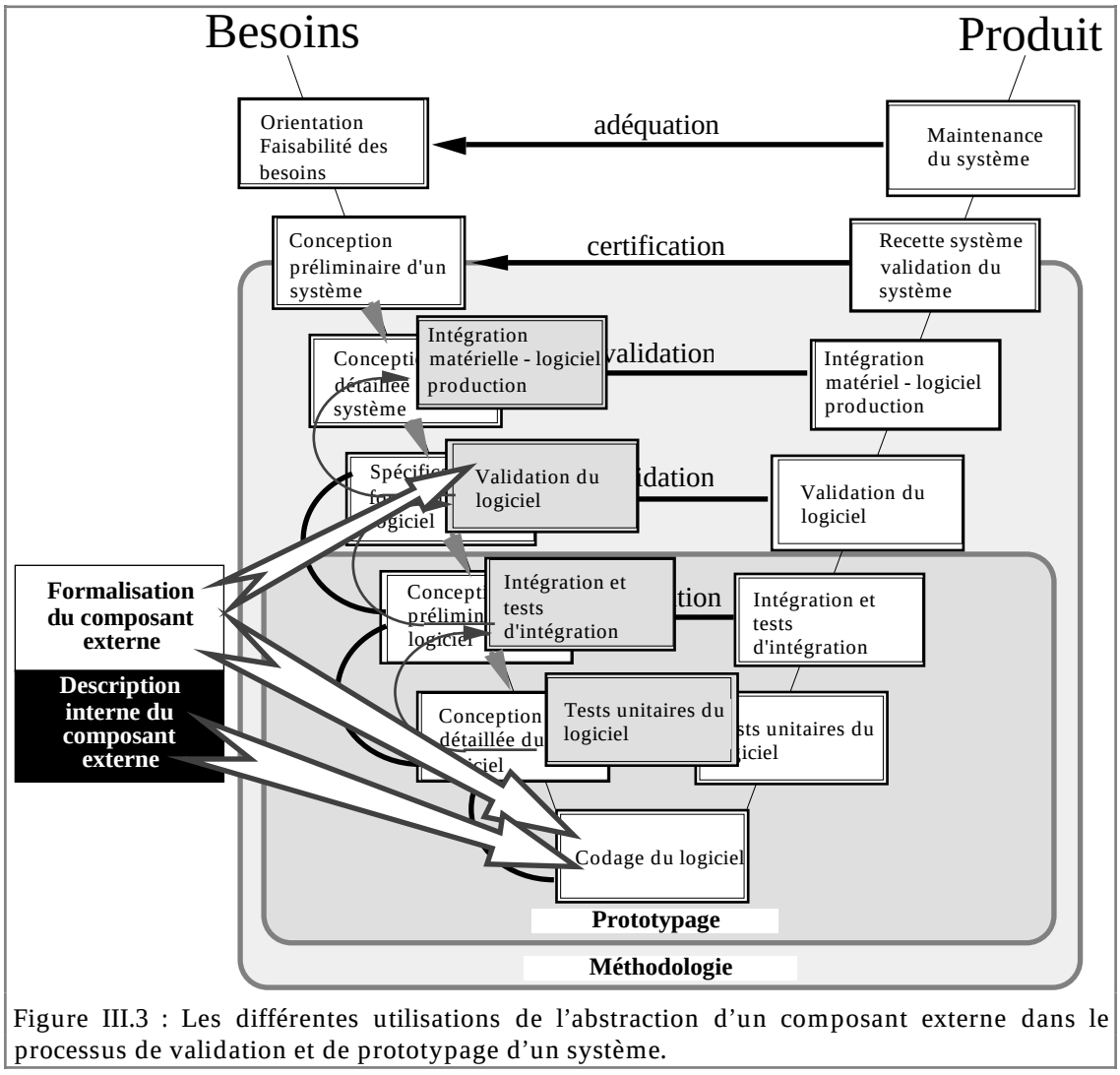


Figure III.3 : Les différentes utilisations de l'abstraction d'un composant externe dans le processus de validation et de prototypage d'un système.

Les abstractions de composants externes interviennent à plusieurs niveaux dans le processus de validation et de prototypage (Figure III.3) :

- lorsqu'il s'agit de valider le modèle fermé;
- lorsqu'il faut caractériser les composants externes afin d'identifier le modèle ouvert;
- lorsqu'il faut interfacer le squelette du prototype, afin d'identifier les primitives de manipulation à utiliser.

Les catégories d'informations utilisées sont complètement différentes selon qu'il s'agisse de spécifier les composants externes, de les reconnaître, ou de les intégrer au prototype. Dans les deux premiers cas, les renseignements sont introduits de manière formelle (formalisation). Lors de l'écriture du prototype,

il faut connaître les primitives de manipulation du composant (description interne).

2.1.1. Formalisation d'un composant externe

La formalisation d'un composant externe est donnée sous la forme d'un réseau de Petri : elle décrit son comportement et ses interfaces. Elle permet de valider le modèle complet puis de localiser les composants externes pour les séparer de la spécification d'un système.

La formalisation d'un composant externe est reliée au modèle du système par des places d'interface[4] en entrée ou en sortie (Définition III.3). La production d'une marque dans une place interface en entrée est équivalente à l'appel d'un service donné, réalisé par le composant. Les résultats sont transmis via des marques qui seront consommées par le modèle du système, dans les places d'interface en sortie.

Définition III.3 : **Places d'interface en entrée et en sortie**

Les places d'interface permettent de relier un composant externe au modèle ouvert d'un système.

Les places d'interface en entrée sont dédiées à la communication modèle ouvert ß composant. Coté modèle ouvert, il n'existe que des arcs entrants. Coté composant, seuls des arcs sortants sont autorisés.

Les places d'interface en sortie sont dédiées à la communication composant ß modèle ouvert. Coté modèle ouvert, il n'existe que des arcs sortants. Coté composant, seuls des arcs entrants sont autorisés.

L'ensemble des places d'interface avec les composants externes est noté I. Nous avons :

- $P = P_{mo} \quad P_c \quad I$ avec $P_{mo} \cap P_c =$, $P_{mo} \cap I =$ et $I \cap P_c =$, où P_{mo} et P_c représentent respectivement l'ensemble des places du modèle ouvert et l'ensemble des places des abstractions qui ne sont pas des interfaces.
- $I = I_e \quad I_s$ avec $I_e \cap I_s =$ où I_e et I_s représentent respectivement l'ensemble des places d'interface en entrée et en sortie.

Les places d'interface représentent la partie *visible* d'un composant externe [Ada 83, Booch 87], c'est-à-dire ce qui peut être utilisé par d'autres sous-ensembles du système.

2.1.2. Description interne d'un composant externe

La description interne du composant externe comporte l'ensemble des informations qui ne peuvent être représentées dans la formalisation, c'est-à-dire :

- Les contraintes de placement du composant externe : elles sont dépendantes d'une architecture.

---

4 Le type de la place d'interface (entrée ou sortie) est désigné par rapport au composant externe.

- Le nom des primitives et celui des paramètres qui leur sont associés. Ces informations sont optionnelles car, comme le format de ces primitives est déduit de la formalisation du composant, des valeurs par défaut peuvent être construites.
- Le corps des primitives associées aux places d'interface : elles sont dépendantes du langage.

Ces informations peuvent être décrites à l'aide de formalismes propres à un outil de génération de code. Elles ne participent pas à la validation comportementale du modèle fermé. Elles représentent la partie *privée* [Ada 83, Booch 87] du composant.

Les contraintes de placement se rapportent à une architecture cible. Elles référencent des éléments contenus dans une bibliothèque décrivant des architectures.

Le corps des primitives de manipulation peut être textuellement exprimé dans le langage cible ou être référencé dans une bibliothèque prédéfinie. L'unité est alors considérée comme visible par le prototype et assemblée au niveau de l'édition de liens.

La format des primitives associées aux places d'interface respecte des règles précises.

Définition III.4 : **Primitives associées aux places d'interface**

Les primitives associées aux places d'interface des composants externes permettent de manipuler un "marquage virtuel", c'est-à-dire autoriser la production ou la consommation de marques.
Pour les primitives d'interface en entrée, seule la production de marques est possible : la primitive associée est de type procédure et ne comporte que des paramètres en entrée (passage par valeur).
Pour les primitives d'interface en sortie, seule la consommation de marques est possible : la primitive associée est de type procédure et ne comporte que des paramètres en sortie (passage par référence).
Le nombre et le type de ces paramètres sont déduits du domaine de couleurs de la place d'interface.

Exemple III.5 : Considérons un composant externe de type clavier respectant les contraintes suivantes :

- L'utilisateur frappe une touche lorsqu'il le désire, que celle-ci soit mémorisée ou non n'a pas d'importance.
- La commande GET, permettant la récupération d'une touche pressée par un utilisateur, doit être activée. Cette commande est bloquante.
- Il n'y a pas de gestion partagée du clavier. Si plusieurs processus attendent une touche à un moment donné, il est impossible de déterminer vers lequel sera dirigée la touche frappée par un utilisateur.

FORMALISATION DU COMPOSANT EXTERNE :

Le réseau de Petri présenté en Figure III.4 correspond à la formalisation du clavier dont nous avons donné la spécification.

La place *Je_veux_une_touche* est une place d'interface en entrée. Elle est reliée à la spécification d'un système par des arcs entrants uniquement. La production d'une marque dans cette place correspond à l'appel de la primitive permettant d'activer la

commande GET. Il n'y a pas de paramètre à une telle primitive puisque les marques déposées ne sont pas colorées.

La place *Touche_pressée* est une place d'interface en sortie. Elle est reliée à la spécification de l'application par des arcs sortants uniquement. Consommer une marque correspond à récupérer le code de la touche pressée, ce code étant déterminé par la couleur de la marque ainsi consommée.

DESCRIPTION INTERNE DU COMPOSANT EXTERNE:

***Contraintes de placement sur une architecture* :**

Il est possible, sur un réseau de machines, que le clavier manipulé par une application soit lié à une unité donnée. Cela a une influence sur le placement des tâches extraites du modèle qui doivent effectuer des accès au clavier.

***Description des primitives de manipulation* :**

A partir de la formalisation du clavier donnée en Figure III.4, il est possible de déduire les primitives d'interface suivantes :

- "produire_dans_Je_veux_une_touche" qui ne comporte aucun paramètre. Le concepteur du composant externe peut appeler cette primitive "initialisation_du_get".
- "consommer_dans_Touche_pressée" qui comporte un paramètre en sortie : "consommer_dans_Touche_pressée_p_1". Il se peut que l'utilisateur souhaite nommer la primitive permettant de ramener le caractère pressé "récupérer_caractère" et son paramètre "caractère_frappé".

Enfin, le corps des primitives "initialisation_du_get" et "récupérer_caractère" doit être défini pour obtenir un prototype exécutable.

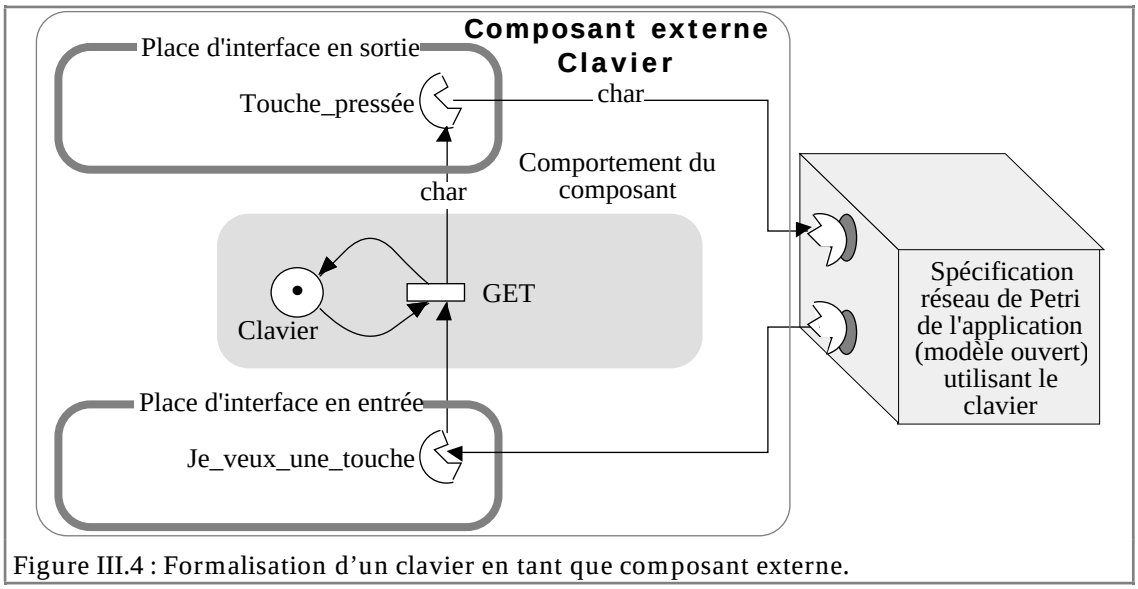


Figure III.4 : Formalisation d'un clavier en tant que composant externe.

Seule la formalisation d'un composant externe est réutilisable. La description interne dépend de critères d'implémentation (langage cible, architecture cible...). Si l'on souhaite réutiliser, pour une nouvelle architecture, ou pour un nouveau langage cible, un composant externe déjà défini, seule la description interne est à modifier. Ainsi, chaque élément de la bibliothèque de composants externes est caractérisé par une formalisation unique et éventuellement, par plusieurs descriptions internes.

2.2. LES COMPOSANTS EXTERNES DANS UN MODÈLE

Les composants externes sont reliés au modèle ouvert par l'intermédiaire des places d'interface. Il est possible que certaines places d'interface d'un

composant externe soient “pendantes”. Cela signifie que les services ou résultats qui leur sont associés ne sont pas utilisés par le système.

Exemple III.6 : Nous donnons en Figure III.5 un exemple de système relié à un clavier. Le modèle comporte donc un composant externe :
Le réseau de Petri présenté modélise un producteur-consommateur. Un processus récupère des touches frappées sur un clavier et les transmet à tour de rôle (via les transitions *Donner_a_1* et *Donner_a_2*), à deux processus consommateurs qui leur appliquent un traitement (transitions *Traitement_1* et *Traitement_2*). Au bout d’un certain temps (défini par le nombre de marques contenues dans *Max*), le producteur se termine après avoir ordonné aux consommateurs d’en faire autant (via la place *Terminer*).

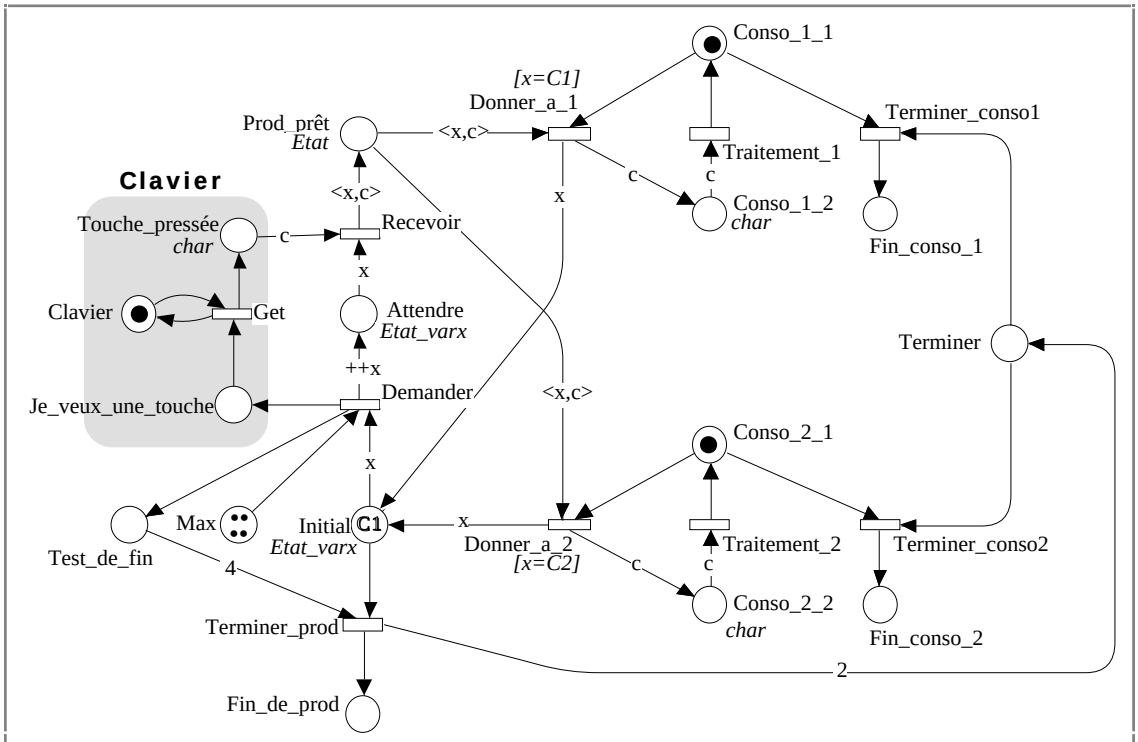


Figure III.5 : Exemple de modèle comportant un composant.

Il est également possible d’intégrer un composant externe via ses places d’interface uniquement. Cela permet de limiter la taille d’un modèle, en particulier lorsque les composants externes sont nombreux et leur formalisation volumineuse. Cependant, aucune validation formelle de l’ensemble du système n’est possible.

2.3. LA NOTION DE SERVICE

En terme de conception objet, des *méthodes* sont associées aux classes [Booch 86, Stroustrup 87, Coad 92]. Ces méthodes définissent les services disponibles pour le composant.

Les composants externes permettent de formaliser l’utilisation d’unités de modélisation [Bachatene 92] pour décrire l’environnement d’un système (Figure III.6). Ainsi, il y a équivalence :

- entre services offerts et places d’interface en entrée;
- entre services requis et places d’interface en sortie.

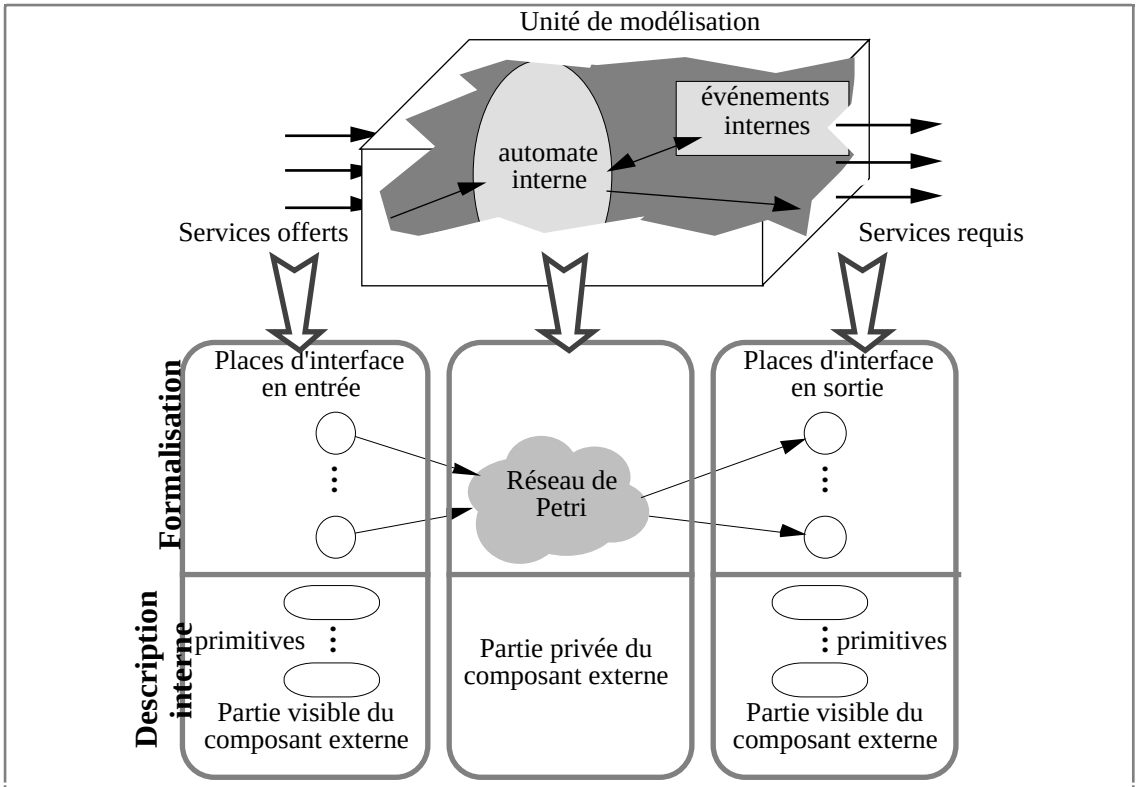


Figure III.6 : Equivalences entre une unité de modélisation et un composant externe.

Il existe, pour les systèmes parallèles, deux types de services :

- Les services synchrones : ils correspondent à une synchronisation entre serveur et client et sont toujours bloquants.
- Les services asynchrones : le client, après avoir effectué sa demande, continue son exécution. Si le service fournit un résultat, deux cas de figure sont envisageables :
  - le client utilise le résultat avant qu'il ne soit produit, il est alors suspendu;
  - le client utilise le résultat après qu'il a été calculé, son exécution n'est pas affectée.

Les seuls services que nous offrons sont de type asynchrone (communication par places). Cependant, il est possible, à l'aide de places d'interface, de modéliser les deux types de services.

Pour représenter un service de type synchrone, il faut le décomposer en deux temps : activation et récupération des résultats.

Exemple III.7 : Le modèle de la Figure III.7 présente un exemple de réalisation d'un service synchrone à l'aide de places d'interface. Le client effectue une demande de service (*lancer service*) puis se bloque en attente du résultat (*fin service*).
Le composant externe clavier défini dans Exemple III.5 possède un unique service de type synchrone, permettant de récupérer le code d'une touche frappée par un utilisateur.
Le client utilisant le clavier, en Figure III.5, réalise le service en deux temps :
- activation : au niveau de la transition *Demander*.
- récupération des résultats : au niveau de la transition *Recevoir*.

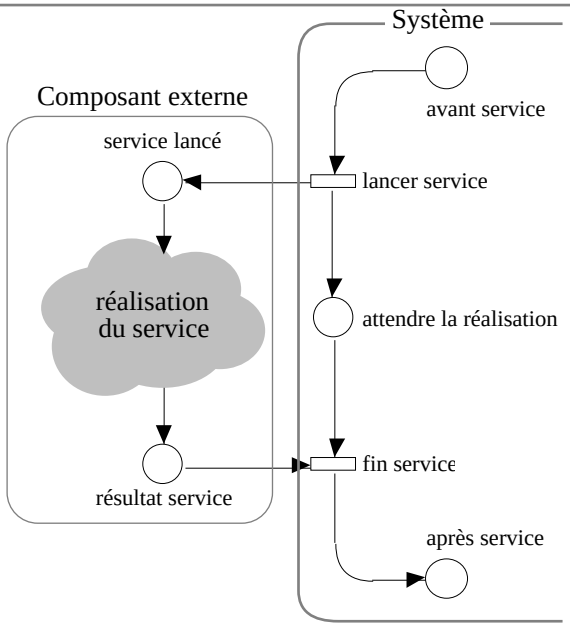


Figure III.7 : Utilisation des places d'interface pour réaliser un service synchrone.

2.4. COMPOSANTS EXTERNES RÉVERSIBLES ET NON-RÉVERSIBLES

Par construction, un composant, décrit de manière externe, a un comportement propre. L'interactivité des services offerts (traitements immédiats ou mémorisés) ne dépend que de lui. Cette caractéristique nous contraint à distinguer deux classes de composants externes : les composants externes *réversibles* et *non-réversibles*. L'exemple III.8 illustre les problèmes potentiels.

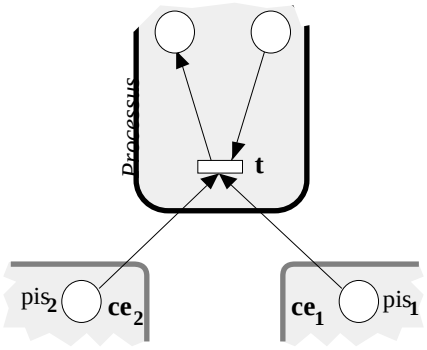


Figure III.8 : Exemple de transition effectuant une demande de résultats provenant de composants externes différents.

Exemple III.8 : Soit une transition t, ayant en précondition deux places d'interface en sortie, pis_1 et pis_2, liées respectivement aux composants externes ce_1 et ce_2 (Figure III.8). Pour exécuter la transition, il faut tenter de réaliser la précondition associée.

Une telle précondition ne pose aucun problème en terme de validation du modèle. Il n'en est pas de même lorsque nous considérons l'exécution d'un prototype basée sur de tels principes.

Comme pis_1 et pis_2 appartiennent à deux composants externes distincts, il est impossible de les évaluer simultanément. Les dispositifs matériels qu'ils représentent ne sont pas contrôlés par le prototype et les primitives associées à pis_1 et à pis_2 doivent être appelées successivement.

Considérons que le marquage de pis_1 soit adéquat mais pas celui de pis_2 . Que faire de ce qui a été consommé au niveau de pis_1 ? Ce problème est lié à "l'exécution du réseau de Petri". Il est impossible à résoudre sans effectuer des hypothèses sur le mode de dialogue avec le composant.
Ce protocole de dialogue implique que l'on puisse "reproduire" une marque indûment consommée. Cela est impossible dans certains cas (Exemple III.9).

Ainsi, il est nécessaire de différencier deux types de composants externes : les composants externes *non-réversibles* et *réversibles*.

Définition III.5 : **Composant externe non-réversible**

Un composant externe non-réversible représente une portion de l'environnement sur laquelle aucune supposition ne peut être effectuée. Nous avons :

- IN_e I_e et IN_s I_s où IN_e et IN_s représentent respectivement les places d'interface en entrée et en sortie des composants externes non-réversibles.

Les modèles ouverts accédant à des composants externes non-réversibles doivent respecter la contrainte suivante :

- t T_{mo}, p_1, p_2 pré (t), p1 IN_s p2 IN_s où T_{mo} représente les transitions du modèle ouvert (une transition du modèle ouvert ne peut référencer plus d'une place d'interface en sortie avec un composant externe non-réversible).

Ils représentent des dispositifs externes sur lesquels aucune supposition sur l'interaction avec le modèle ouvert ne peut être effectuée.

Définition III.6 : **Composant externe réversible**

Un composant externe réversible représente une portion de l'environnement pour laquelle il est considéré que toutes les places d'interface en sortie correspondent à des "zones de stockage de données" permettant une communication asynchrone entre le composant externe réversible et le modèle ouvert. Nous avons :

- IR_e I_e et IR_s I_s où IR_e et IR_s représentent respectivement les places d'interface en entrée et en sortie des composants externes réversibles. Bien sûr, $IN_e \cap IR_e =$ et $IN_s \cap IR_s =$.

Les références à des places d'interface en sortie, pour les composants externes réversibles, ne sont soumises à aucune restriction.
Ils représentent des dispositifs externes pour lesquels il y a réversibilité des services offerts.

Nous avons défini des règles régissant la spécification des primitives associées aux places d'interface des composants externes (Définition III.4). De telles règles peuvent être reprises tell quel pour les composants externes non-réversibles. En revanche, il faut les étendre pour les composants externes réversibles.

Définition III.7 **Extension des règles de définition des primitives associées aux places d'interface des composants externes réversibles**

Deux primitives sont associées aux places d'interface en sortie des composants externes réversibles :

- la première permet de consulter le "marquage" (les données) associé à la place d'interface. C'est une procédure ne comportant que des paramètres en sortie (passage par référence).
- la seconde permet de mettre à jour le "marquage" associé à la place d'interface. C'est une procédure ne comportant que des paramètres en entrée (passage par valeur).

Dans les deux cas, le format des primitives est obtenu à l'aide du domaine de couleurs de la place d'interface considérée.

D'une manière générale, les composants externes non-réversibles sont plus aptes à modéliser un environnement ou un dispositif matériel (Exemple III.1). Les composants externes réversibles sont utilisés dans la conception séparée de morceaux d'application (Exemple III.2) ou représentent une couche de traitement associée à un composant externe non-réversible (Exemple III.9).

Exemple III.9 : Considérons un composant externe de type fichier séquentiel. Si un article de ce fichier a été inconsidérément consommé, il est impossible de le "reproduire" car une telle action implique un retour en arrière qui est incompatible avec la définition d'un fichier séquentiel. Il s'agit d'un composant externe *non-réversible.*

En programmation, une telle opération est cependant possible au prix d'un dispositif logiciel encapsulant la notion de fichier séquentiel (on réinvente alors les fichiers à accès direct!). En spécification, c'est la même chose. Il faut modéliser cette couche de traitement supplémentaire sur la base d'un composant externe *non-réversible* aux possibilités plus restreintes. L'ensemble ainsi défini peut lui-même devenir un composant externe *réversible*.

3. Les étapes du prototypage

Le prototypage d'un modèle ouvert est une opération complexe qui doit être décomposée en plusieurs étapes. La plupart des études se sont focalisées sur les techniques d'implémentation des réseaux de Petri [Silva 79, Silva 80, Silva 82, Valette 82, Nelson 83, Valette 83, Thuriot 85, Bruno 86, Colom 86, Murata 86, Hauschildt 87, Taubner 87]. L'approche méthodologique proposée dans [Paludetto 91] est particulièrement adaptée aux systèmes temps réel. L'originalité de l'approche que nous proposons consiste en l'intégration progressive d'informations concernant le système et son environnement.

Notre méthode de prototypage se décompose en deux étapes (Figure III.9) :

- la décomposition du modèle : elle est indépendante du langage d'implémentation du prototype puisqu'elle ne travaille que sur des réseaux de Petri;
- la production du prototype : il existe un générateur par langage cible possible. Elle intègre des informations dépendantes du langage et de l'architecture cible choisis;

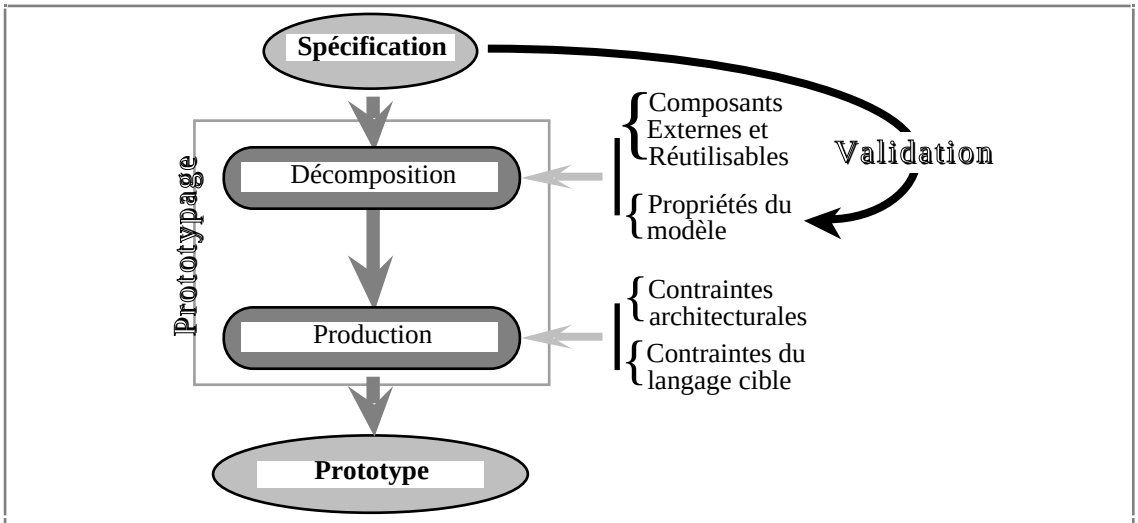


Figure III.9 : Etapes du prototypage.

Pendant l'étape de décomposition, il faut identifier les composants externes et caractériser les G-objets. Ce traitement est effectué sur la base :

- de la description des composants externes,
- des résultats issus de la théorie des réseaux de Petri.

Ensuite, il faut produire le prototype. Pour cela, il faut intégrer :

- les contraintes architecturales (description de l'architecture, contraintes de placement des composants externes et des G-objets),
- les contraintes du langage cible.

3.1. Décomposition d'un modèle

La décomposition en G-objets (Chapitre II) ne concerne que le modèle ouvert. Il faut préalablement séparer ce dernier de la représentation de son environnement.

La décomposition d'un modèle s'effectue en deux phases (Figure III.10) :

- La phase d'*Identification* permet de vérifier l'utilisation des *composants externes* dans la spécification puis de les séparer du modèle ouvert. Elle produit un réseau de Petri expurgé des composants externes du modèle donné en entrée.
- La phase de *Caractérisation* assure la décomposition du réseau résultat de la phase d'Identification en G-objets.

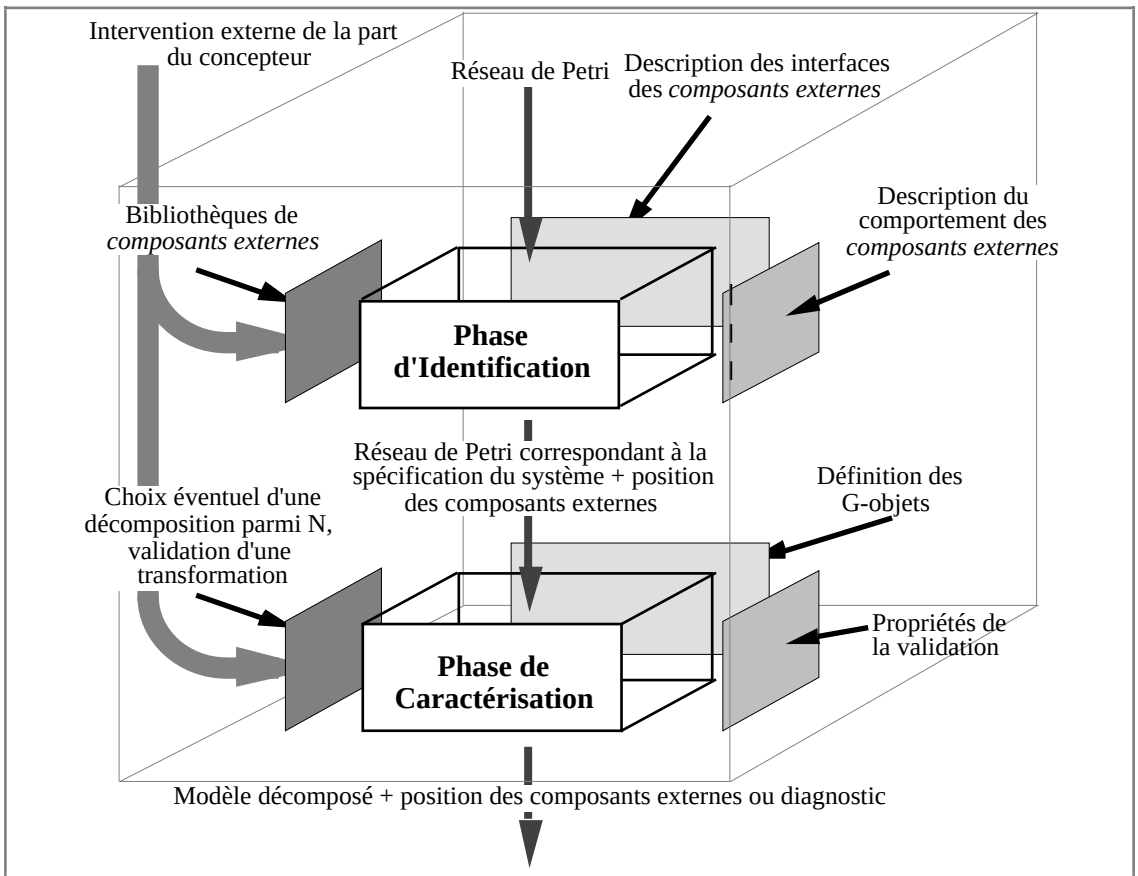


Figure III.10 : Les différentes étapes de la décomposition d'un réseau de Petri en G-objets.

Le concepteur du système signale les places d'interface reliant les composants externes au modèle ouvert. La phase d'Identification utilise la formalisation des composants externes ainsi désignés afin de vérifier qu'ils sont conformes à la description qui en est faite. Si tel n'est pas le cas, le processus de prototypage est interrompu.

A partir du résultat fourni par la phase d'Identification, la phase de Caractérisation fournit, soit une décomposition du modèle en G-objets, soit un diagnostic si le modèle n'est pas décomposable. Dans ce cas, le processus de prototypage est interrompu.

La décomposition du modèle est effectuée d'après l'algorithme décrit dans le Chapitre II. Les places d'interface avec les composants externes sont, à ce stade, considérées comme des Ressources du modèle.

Si le modèle ne comporte qu'une seule décomposition possible, son concepteur n'intervient pas. S'il comporte plusieurs décompositions valides, il doit indiquer sa préférence. La suite du prototypage peut alors être appliquée.

En cas d'échec de la décomposition, plusieurs cas sont envisageables :

- Le modèle peut être automatiquement transformé en un modèle "prototypable". Après acceptation des transformations par le concepteur du modèle, il devient possible d'obtenir une ou plusieurs décompositions. Dans le cas contraire, le modèle est rejeté.
- Le modèle n'est "pas prototypable", ou les transformations qui lui sont applicables ne sont pas automatisables. La phase de Caractérisation fournit un diagnostic. La génération du prototype ne peut pas être effectuée.

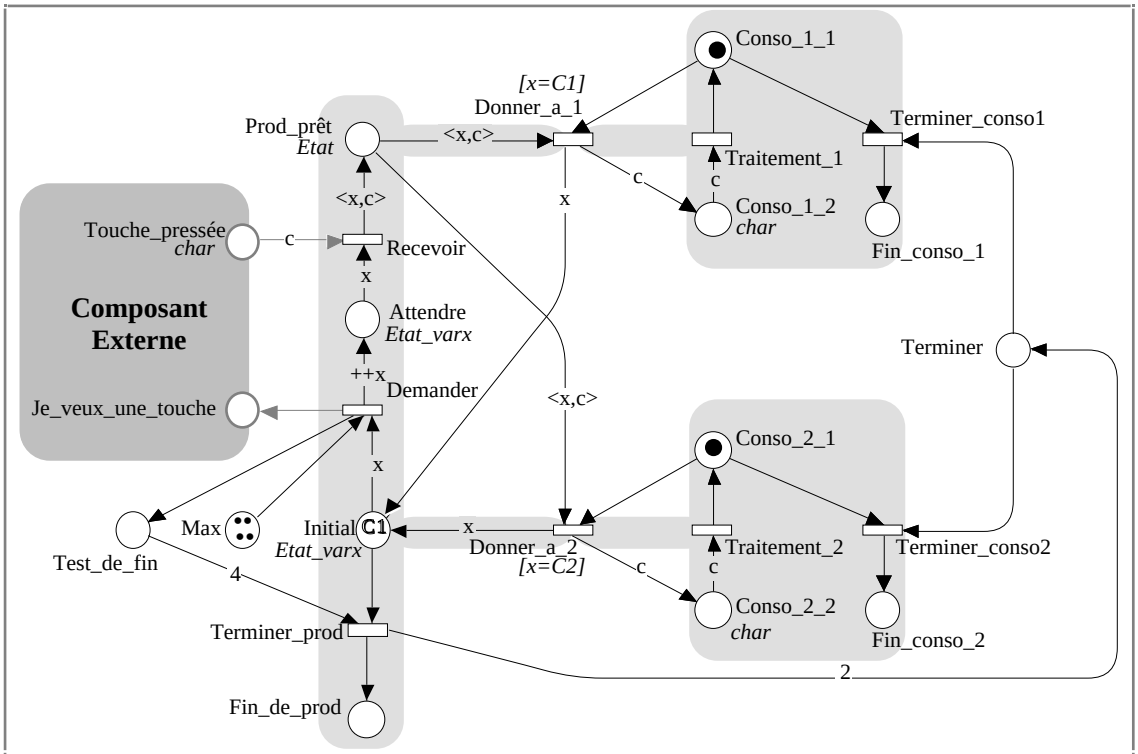


Figure III.11: Décomposition en Processus du modèle de la Figure III.5 après suppression des composants externes. Seule la position des places d'interface est mémorisée.

Exemple III.10 : Considérons à nouveau le modèle de la Figure III.5.

PHASE D'IDENTIFICATION :

Le concepteur du modèle indique que les places *Je_veux_une_touche* et *Touche_pressée* sont des interfaces avec un clavier. Le composant externe peut être caractérisé à l'aide des informations contenues dans la bibliothèque et le modèle décrivant le système identifié. La phase d'Identification rend le réseau de Petri donné en Figure III.11.

PHASE DE CARACTÉRISATION :

La phase de Caractérisation, sur la base de l'algorithme décrit dans le Chapitre II, trouve une unique décomposition du modèle en trois Processus, que nous nommons *Prod*, *Cons1* et *Cons2*. Les autres objets sont récapitulés dans le tableau ci-dessous :

Objets	Etats_Processus			Actions			Ressources	Processus		
	simp.	alt.	de term	simp.	sync	gard.		Prod	Cons1	Cons2
Prod_Prêt		7						7		
Attendre	7							7		
Initial		7						7		
Fin_de_prod			7					7		
Conso_1_1		7							7	
Conso_1_2	7								7	
Fin_conso_1			7						7	
Conso_2_1		7								7
Conso_2_2	7									7
Fin_conso_2			7							7
Max							7			
Test_de_fin							7			
Terminer							7			
Terminer_prod				7		7		7		
Demander				7		7		7		
Recevoir				7		7		7		
Terminer_conso1				7		7			7	
Terminer_conso2				7		7				7
Traitement_1				7					7	
Traitement_2				7						7
Donner_a_1					7	7		7	7	
Donner_a_2					7	7		7		7

Le processus *Prod* possède un mot d'état : <x, c>. Il est instancié une seule fois à l'Etat_Processus *Initial* avec la valeur <C1, "pas de valeur significative">. Les processus *Cons1* et *Cons2* ont également un mot d'état réduit à la seule variable c. *Cons1* et *Cons2* sont instanciés une seule fois, respectivement aux Etats_Processus *Conso_1_1* et *Conso_2_1*. Aucune valeur par défaut n'est affectée aux mots d'état.
Les relations des places d'interface (*Je_veux_une_touche* et *Touche_pressée*) avec les transitions du modèle sont transmises aux étapes suivantes.

3.2. PRODUCTION DU PROTOTYPE

Cette deuxième étape du processus de prototypage gère le placement des G-objets et la production d'un prototype dans un langage cible donné.

Lorsque l'on réalise des applications réparties sur plusieurs processeurs, il est intéressant de pouvoir étudier différentes stratégies de placement. Deux solutions sont envisageables :

- Le concepteur du modèle effectue lui-même différents placements et analyse les performances d'exécution afin d'en trouver un optimal. C'est l'approche choisie dans PARALLEL-PROTO [Burns 91].
- L'outil de prototypage fait une proposition de placement sur la base de résultats de l'évaluation du modèle (connexité...) [Sinclair 87, Lo 88, Billionnet 89]. L'état actuel des recherches dans ce domaine permet d'obtenir des heuristiques. Le concepteur d'un modèle doit cependant pouvoir inférer sur le placement des tâches de son application.

Afin de rendre l'outil de prototypage plus efficace, il nous paraît intéressant de proposer un placement des tâches le composant sur une architecture donnée. L'étude d'un placement ne se justifie évidemment que pour des architectures multi-processeurs.

Le calcul du placement s'appuie sur une connaissance de la structure du prototype pour un langage cible donné (coût et performances des mécanismes de communication). Il faut également connaître la configuration de l'architecture cible. C'est un résultat qui est fourni à la phase de génération du prototype.

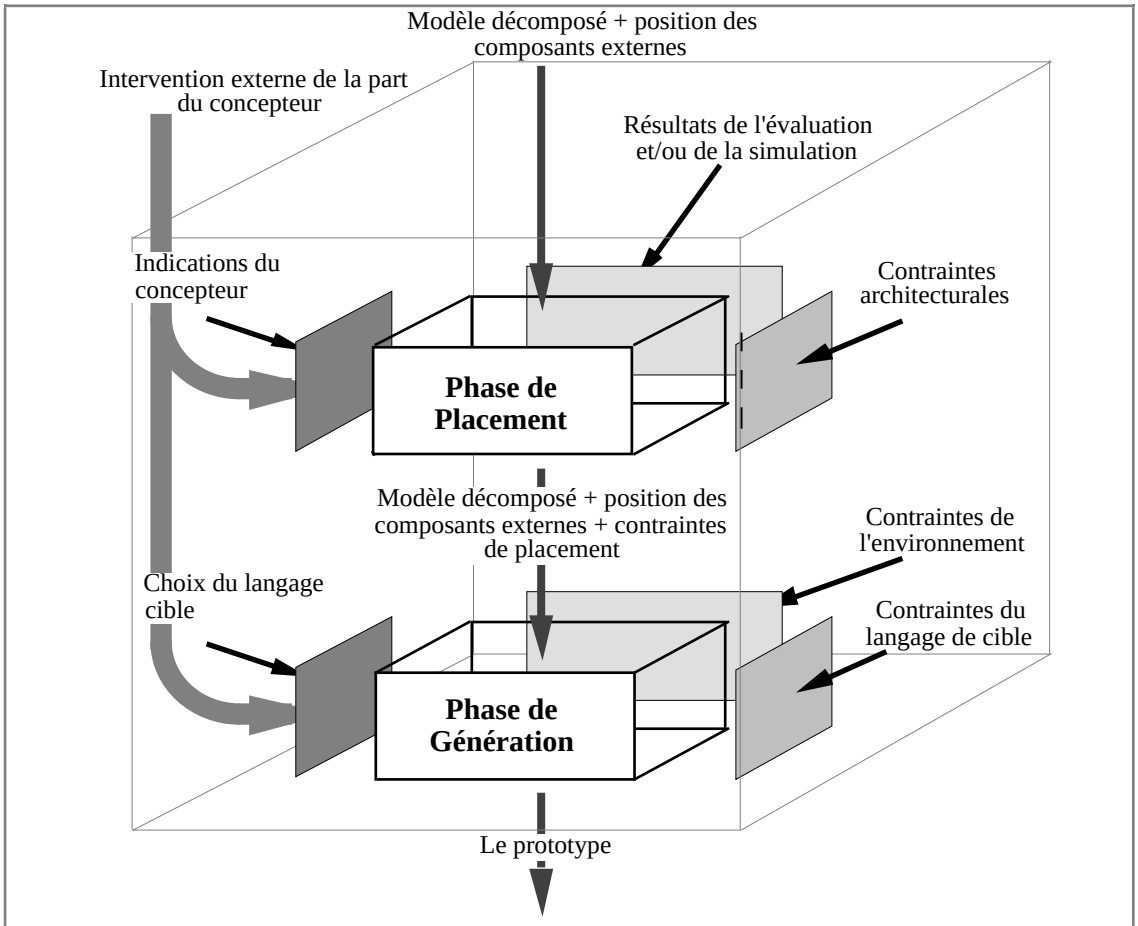


Figure III.12 : Les différentes étapes de production d'un prototype à partir de la décomposition du réseau de Petri le spécifiant.

Nous avons décomposé la génération d'un prototype en deux étapes (Figure III.12) :

- la phase de *Placement* propose un placement des tâches de l'application sur une architecture donnée;
- la phase de *Génération* génère le prototype.

A partir de la décomposition d'un réseau de Petri en G-objets, c'est-à-dire en termes de processus instanciés (contenant des Etats_Processus et des Actions) et de liens de communication (Ressources, Actions synchronisées), la phase de Placement effectue une proposition de placement. La décomposition du modèle lui est fournie par la phase de Caractérisation. Il faut prendre en compte :

- Les propriétés du modèle (type de communications, densité des communications...) : elles sont dépendantes du langage cible et des choix d'implémentation effectués par la phase de Génération.
- La description de l'architecture (nombre de processeurs, topologie...) : elle est décrite par le concepteur du modèle. Il est intéressant de gérer des bibliothèques d'architectures.
- Les contraintes de placement : certaines sont indiquées par le concepteur du système qui peut forcer le placement de certains objets; d'autres sont liées à la description interne des composants externes manipulés par le prototype.

La phase de Placement échoue si des contradictions apparaissent :

- entre les indications du concepteur du système et la description de l'architecture ou les contraintes de placement des composants externes,
- entre les contraintes de placement des composants externes et la description de l'architecture.

A partir de la décomposition du modèle en G-objets et d'une proposition de placement, la phase de Génération crée un programme source. Elle doit tenir compte des contraintes du langage, des contraintes liées à l'environnement imposé par ce langage (système d'exploitation...) et intégrer les composants externes par le biais d'appels de primitives.

Exemple III.11 : Considérons à nouveau le modèle de la Figure III.5 et sa décomposition (Exemple III.10).

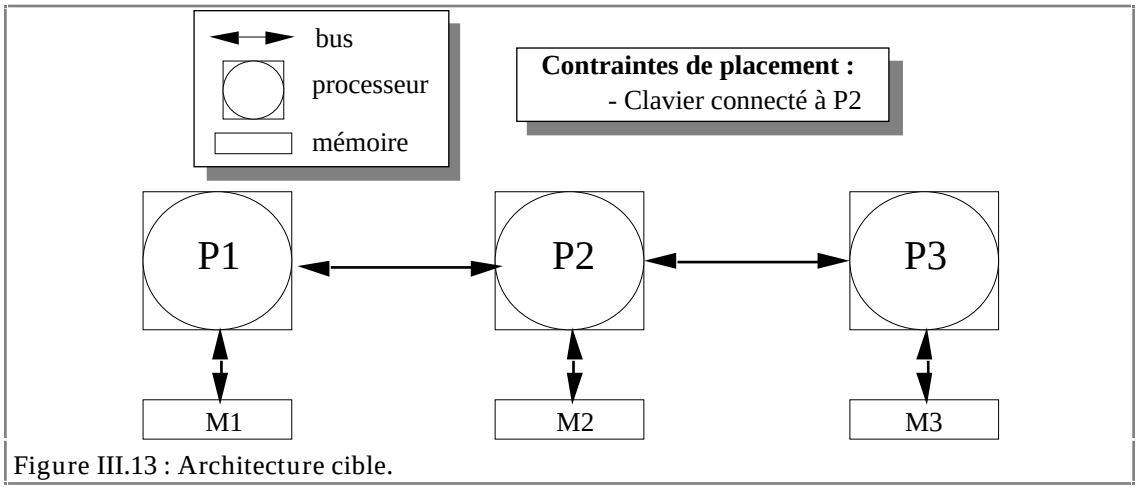


Figure III.13 : Architecture cible.

PHASE DE PLACEMENT :

Considérons que l'application spécifiée doive s'exécuter sur l'architecture décrite en Figure III.13 et qu'aucune contrainte de placement ne soit spécifiée dans l'abstraction d'un clavier. La machine cible possède trois processeurs avec mémoire locale reliés deux à deux. Le composant externe non-réversible Clavier est relié au processeur P2.

Si les communications entre processeurs sont performantes, la phase de Placement proposera de placer *Prod* sur P2, *Cons1* et *Cons2* étant affectés aux processeurs P1 et P3. Dans le cas contraire, elle pourra décider de placer les trois processus sur un seul processeur : P1.

PHASE DE GÉNÉRATION :

Considérons que la phase de Placement ait proposé un placement sur trois processeurs. La phase de Génération produira trois programmes différents, destinés à être exécutés

sur les trois processeurs de l'architecture (Figure III.14). Seul le programme implémentant le *Prod* sera relié au clavier via des primitives de manipulation.

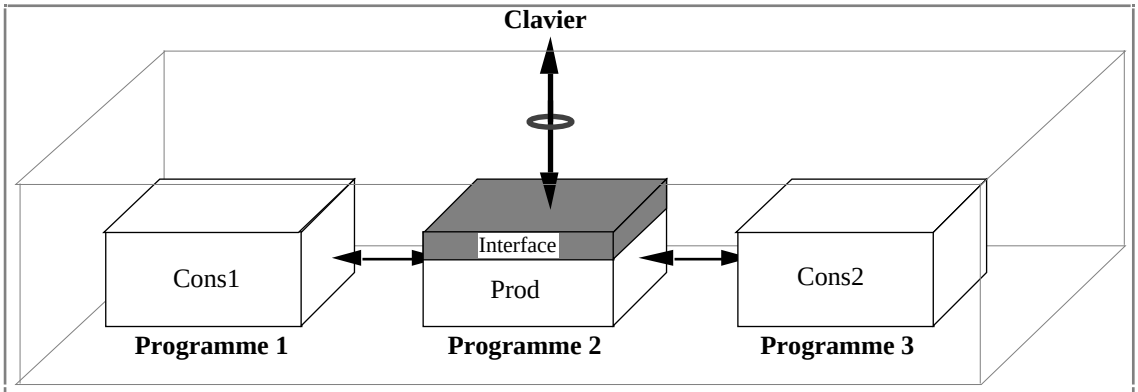


Figure III.14 : Organisation possible du prototype associé au modèle de la Figure III.5, pour l'architecture décrite en Figure III.13.

4. Conclusion

La notion de composant externe, que nous avons définie, permet d'établir des règles de composition d'un modèle ouvert avec des modules prédéfinis. Il existe deux types de composants externes :

- Les composants externes non-réversibles qui représentent des dispositifs externes sur lesquels aucune supposition n'est faite. Leur utilisation est soumise à de fortes contraintes.
- Les composants externes réversibles qui représentent des unités déjà prototypées ou des dispositifs répondant à des règles de comportement particulières. Les règles régissant leur intégration dans la spécification d'un système sont beaucoup plus souples. Ces composants permettent, entre autres, de spécifier et de valider modulairement un système.

Considérant les composants externes (qui ne font pas partie de la spécification d'un système), l'algorithme de décomposition décrit dans le Chapitre II, les problèmes liés aux contraintes de l'architecture et du langage cible, nous avons divisé le processus de prototypage en quatre étapes successives :

- la phase d'Identification, chargée de gérer les composants externes;
- la phase de Caractérisation, chargée de décomposer le modèle en G-objets;
- la phase de Placement, chargée de proposer un placement adéquat des composantes du prototype sur une architecture donnée;
- la phase de Génération, chargée de générer le squelette du prototype.

Chaque étape intègre une catégorie d'informations différente. L'échec de l'une d'entre elles bloque le processus.

Il est important que l'utilisateur puisse intervenir sur le déroulement du processus à l'aide de directives. Ainsi, certaines étapes, comme la phase de Caractérisation, sont interactives (s'il faut effectuer un choix entre plusieurs décompositions possibles).

Nous avons présenté, dans ce chapitre, une application concrète : un composant externe y est défini puis intégré dans un modèle auquel les différentes étapes du prototypage sont appliquées.

La Figure III.15 représente l'organisation des traitements nécessaires pour générer un prototype.

Les deux premières étapes que nous avons définies ne portent que sur le réseau de Petri spécifiant le système. Elles sont donc indépendantes du langage choisi pour le prototype.

Les deux autres étapes sont dépendantes d'une architecture et d'un langage cibles. Si pour un langage cible donné, la phase de Placement gère l'ensemble des configurations architecturales possibles, ces deux étapes ne dépendent alors que du langage cible. Naturellement, différents langages, comparable

dans leur mise en œuvre du parallélisme, peuvent nécessiter des techniques de placement similaires mais ceci constitue un cas particulier.

Dans ce cas, si l'on a déjà réalisé un prototype dans un langage donné et que l'on veut en obtenir un autre dans un langage différent, seules les phases de Placement et de Génération seront à effectuer.

Cela conditionne l'architecture d'un générateur de code suivant les principes évoqués dans ce chapitre. Il faut diviser le processus en deux parties : la décomposition du réseau de Petri et la production du prototype.

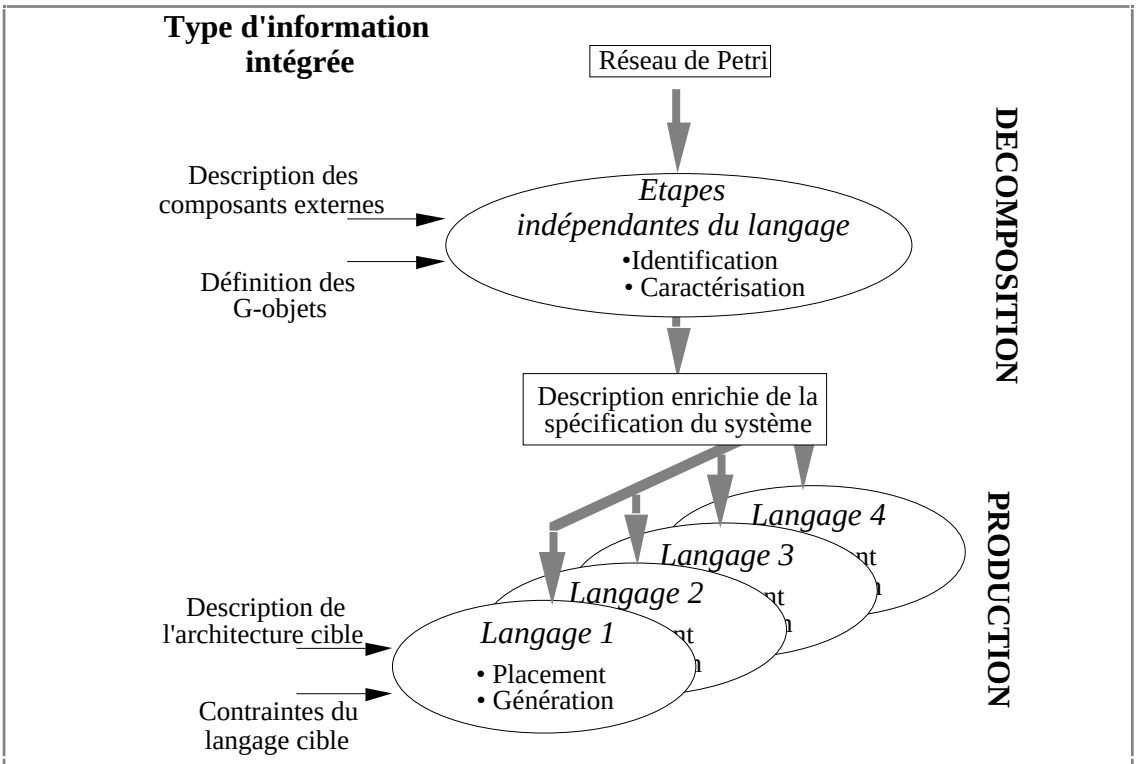


Figure III.15 : Enchaînement des étapes du prototypage.

		Introduction générale
+	Chapitre I	: *Prototypage de systèmes parallèles*
	Chapitre II	: *Interprétation sémantique d'un réseau de Petri*
	Chapitre III	: *Méthodologie de prototypage*
	Chapitre IV	: *Structure du prototype généré*
	Chapitre V	: *Caractéristiques du prototype Ada*
	Chapitre VI	: *Le prototype centralisé*
	Chapitre VII	: *Le prototype réparti*
	Chapitre VIII	: *CPN/TAGADA : L'outil de prototypage*
		Conclusion générale
		Bibliographie
		Annexes & Appendices

Chapitre IV :

Structure du prototype généré

1. Introduction 127
   - 1.1. Données en entrée de la phase de génération 128
   - 1.2. Le prototype obtenu....129
   - 1.3. Les différentes composantes du prototype....130
2. Les modules-processus 134
   - 2.1. Problèmes liés à l'activation d'une Action....135
   - 2.2. Etat_Processus....139
   - 2.3. Actions....140
   - 2.4. Comportement d'un Processus 141
   - 2.5. Services requis....143
   - 2.6. Services offerts 149
   - 2.7. Synthèse 151
3. Le module de gestion des Ressources 154
   - 3.1. Gestion des interfaces avec l'environnement 155
   - 3.2. Services offerts 160
   - 3.3. Services requis....165
   - 3.4. Synthèse des interfaces 167
   - 3.5. Architectures possibles du module....168
4. Le module de gestion des Actions synchronisées 172
   - 4.1. Problèmes liés à l'activation d'une Action synchronisée....172
   - 4.2. Services offerts 176
   - 4.3. Services requis....177
   - 4.4. Synthèse des interfaces 178
   - 4.5. Architecture du module 179
5. Le module de contrôle 182
   - 5.1. Services offerts 182

5.2. Services requis 184
5.3. Synthèse des interfaces.....185
5.4. Architectures possibles pour le module de contrôle.....186
6. Conclusion 189

1. Introduction

Ce chapitre décrit l'architecture fonctionnelle des prototypes que nous générons automatiquement.

Compte tenu de nos hypothèses portant sur les Actions et le domaine d'application visé, les langages cibles que nous considérons sont procéduraux et autorisent la *manipulation du parallélisme*.

Le langage doit être procédural dans la mesure où nous invoquons une procédure à chaque traitement associé à une Action du modèle. Cette procédure est définie de manière externe par le concepteur du système. On notera que, de manière récursive, une telle procédure aurait elle-même pu être spécifiée à l'aide d'un modèle formel puis prototypée. La limite intuitive du procédé est atteinte lorsque la procédure ne manipule que des données locales, sans interaction avec d'autres activités du système.

La manipulation du parallélisme est nécessaire dans la mesure où le prototype généré contiendra, en plus des éléments de code directement issus des G-objets, un ensemble de tâches de services assurant sa mise en fonctionnement.

L'objet de ce chapitre est donc de détailler les différents modules composant un prototype. Pour chacun de ces modules, nous nous attachons à décrire précisément, conformément à la notion de composant, les services offerts et requis. Les stratégies mises en œuvre sont discutées. Un schéma synthétise l'organisation des différents modules du prototype :

- les modules-processus, supportant l'exécution des Processus du modèle ouvert;
- le module de gestion des Ressources, réalisant les services liés aux données partagées entre les Processus;
- le module de gestion des Actions synchronisées, mettant en œuvre les techniques de rendez-vous entre Processus;
- le module de contrôle qui gère l'initialisation et la terminaison du prototype.

A partir d'un cahier des charges, nous obtenons trois types d'informations (Figure IV.1) : le réseau de Petri modélisant le comportement du système, la description des traitements de l'utilisateur (associés aux Actions) ainsi que celle de l'environnement (composants externes réversibles et non-réversibles).

Ces trois types d'informations nous permettent d'obtenir le squelette de contrôle du prototype.

Définition IV.1 : **Squelette de contrôle**

> Les modules du prototype sont directement issus de la décomposition en G-objets du modèle ouvert. Ils supportent l'exécution de l'ensemble des activités concurrentes, les fonctionnalités liées aux données partagées, l'initialisation et la terminaison.

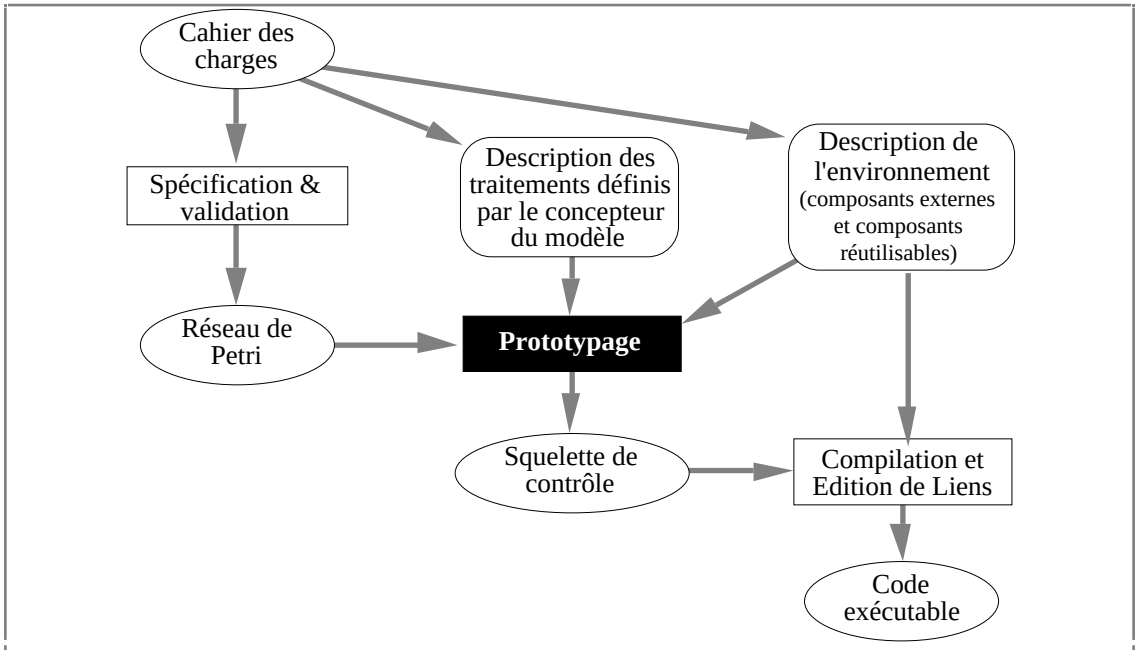


Figure IV.1 : L'intégration des différentes informations dans le processus de prototypage.

Le prototype en lui-même est obtenu par compilation et édition de liens du squelette de contrôle et des unités fournies par le concepteur du système. L'édition de liens doit être effectuée avec des bibliothèques contenant les primitives de manipulation associées à l'environnement du modèle.

1.1. DONNÉES EN ENTRÉE DE LA PHASE DE GÉNÉRATION

Cette section formalise le processus de production d'un prototype. Nous définissons ce que nous avons en entrée, ce que nous souhaitons obtenir comme résultat et les différentes composantes d'information dont nous avons besoin.

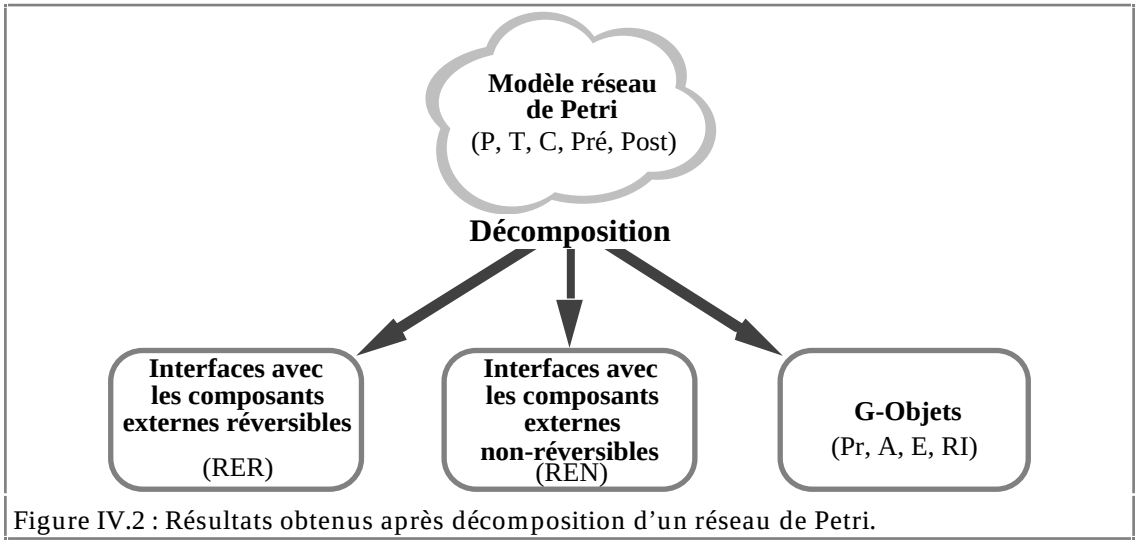


Figure IV.2 : Résultats obtenus après décomposition d'un réseau de Petri.

Le résultat obtenu après décomposition du modèle se divise en deux parties (Figure IV.2) :

- l'ensemble des G-objets : c'est-à-dire le résultat de la décomposition du modèle;

- la liste des interfaces avec les composants externes réversibles et non-réversibles.

Les relations entre le modèle et les composants externes (réversibles ou non-réversibles), sont réalisées au moyen d'ensembles de places d'interface notés RER (composants externes réversibles) ou REN (composants externes non-réversibles).

Chaque ensemble de places d'interface (RER et REN) est divisé en deux parties : les places d'interface en entrée et les places d'interface en sortie.

Définition IV.2 : **Partitionnement des places d'interface des composants externes réversibles et non-réversibles**

Soit REN l'ensemble des places d'interface connectant le modèle du système aux composants externes non-réversibles qu'il manipule. Nous avons

$REN = REN_e \quad REN_s$ avec $REN_e \cap REN_s =$ où :

- REN_e est l'ensemble des places d'interface en entrée des composants externes non-réversibles;
- REN_s est l'ensemble des places d'interface en sortie des composants externes non-réversibles.

Soit RER l'ensemble des places d'interface connectant le modèle du système aux composants externes réversibles qu'il manipule. Nous avons

$RER = RER_e \quad RER_s$ avec $RER_e \cap RER_s =$ où :

- RER_e est l'ensemble des places d'interface en entrée des composants externes réversibles;
- RER_s est l'ensemble des places d'interface en sortie des composants externes réversibles.

Définition IV.3 : **L'ensemble des G-objets**

Soit G l'ensemble des G-objets du modèle analysé. G est caractérisé par le quadruplet (Pr, A, E, RI) :

- Pr est l'ensemble des Processus du modèle;
- A est l'ensemble des Actions du modèle;
- E est l'ensemble des Etats_Processus du modèle;
- RI est l'ensemble des Ressources du modèle.

1.2. LE PROTOTYPE OBTENU

Le prototype obtenu est constitué d'un ensemble de modules coopérant (Figure IV.3). Chaque module est une entité séquentielle; l'exécution des modules est potentiellement concurrente.

Le comportement de chaque module respecte des règles qui peuvent être définies à l'aide de machines à états. Ce comportement doit tenir compte d'événements internes et externes.

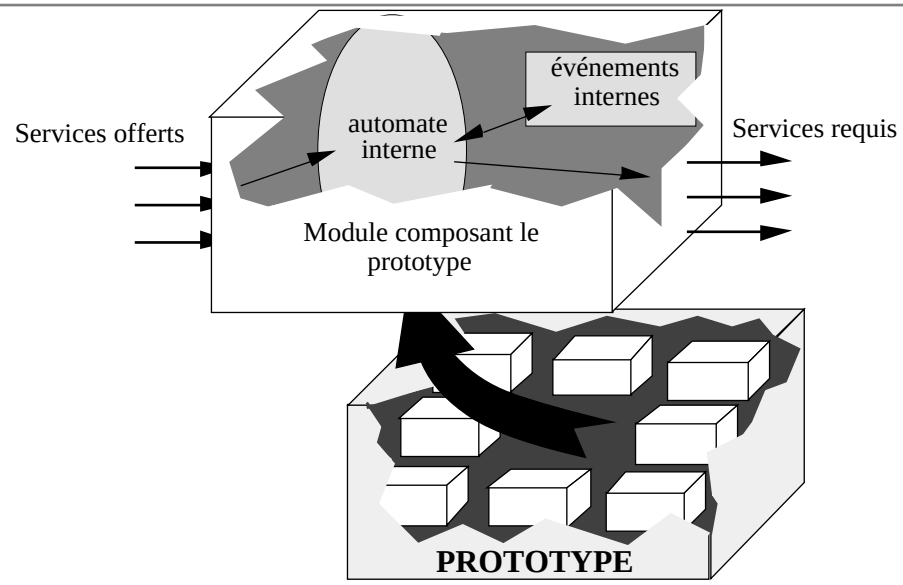


Figure IV.3 : Structure du prototype obtenu.

Les liens entre un module et ses homologues sont réalisés au moyen de *services offerts* et de *services requis* [Bachatene 92]. Les services offerts sont utilisés par les autres parties du prototype pour stimuler le module (événements externes). Ce dernier peut avoir besoin de services requis, fournis par d'autres modules.

1.3. LES DIFFÉRENTES COMPOSANTES DU PROTOTYPE

Le prototype se décompose en trois parties : le système, les composants externes réversibles et les composants externes non-réversibles (Figure IV.4). Il existe deux types de modules : *internes* et *externes*.

Les modules *internes* sont déduits de la décomposition du réseau de Petri. Ils sont complètement inclus dans le prototype (services, automates d'états finis).

Les modules *externes* sont liés aux composants externes réversibles ou non-réversibles. Seule une interface avec les services est générée : la partie privée de ces modules, provenant de la description interne, est intégrée au prototype durant l'édition de liens.

Les modules internes coopèrent, communiquent et interagissent avec les modules non-réversibles : ils *sont* le système modélisé. Ce système se divise en deux parties : le *squelette* et les *traitements* associés aux Actions (Figure IV.4).

Le squelette réalise les différentes fonctionnalités déterminées, au niveau de l'analyse, par les G-objets. Son rôle est de lancer les actions associées aux transitions du réseau de Petri, conformément à la spécification.

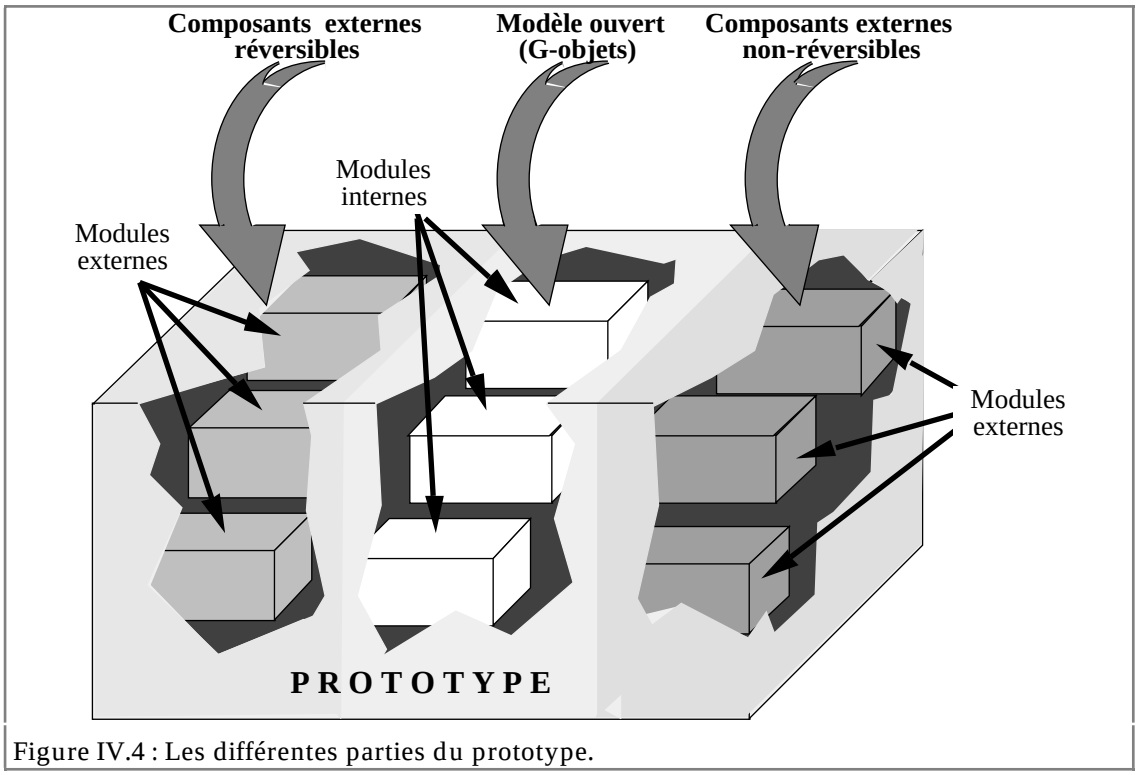


Figure IV.4 : Les différentes parties du prototype.

Les traitements associés aux transitions sont fournis par le concepteur du modèle. Ils seront compilés avec le squelette.

***Définition IV.4 :* Les différents types d'Actions**

Soit A l'ensemble des G-objets de type Action. A est divisé en deux sous-ensembles disjoints A = ASI ASY, avec ASI ∩ ASY = où :

- ASI est l'ensemble des Actions simples du modèle,
- ASY est l'ensemble des Actions synchronisées du modèle.

Le squelette est généré automatiquement à partir de la décomposition en G-objets. Chaque type de G-objet caractérise une classe de fonctionnalités qui doit être implémentée :

- Les Ressources (ensemble RI) du modèle correspondent à des liens de communication asynchrones. Nous choisissons de les gérer dans une unité dédiée.
- Les Actions synchronisées du modèle (ensemble ASY) définissent des mécanismes de communication synchrones. Nous choisissons de les gérer dans une unité dédiée.
- Les Etats_Processus (ensemble E) et les Actions simples (ensemble ASI) sont inclus dans des Processus (ensemble Pr). Un Processus définit une entité de traitement séquentielle parallélisable. Chaque Processus est réalisé sous la forme d'un module processus.

Deux implémentations des interfaces avec les composants externes réversibles et non-réversibles sont envisageables :

- l'appel des primitives associées est effectué par les modules-processus concernés;

- l'appel des primitives associées est effectué par le module de gestion des Ressources.

Une place d'interface avec un composant, que ce soit en entrée ou en sortie, représente un lien de communication avec le monde extérieur, c'est-à-dire une Ressource ayant des propriétés particulières. Nous distinguons ainsi les *Ressources internes* des *Ressources externes*.

Définition IV.5 : **Ressources internes**

Les Ressources internes sont celles définies dans la décomposition du réseau de Petri correspondant au système, c'est-à-dire les éléments de RI.

Définition IV.6 : **Ressources externes**

Les Ressources externes sont les places d'interface des composants externes réversibles ou non-réversibles, c'est-à-dire les éléments de REN et RER. Il existe deux types de Ressources externes : les Ressources externes en sortie (éléments de REN_s RER_s) et les Ressources externes en entrée (éléments de REN_e RER_e).

La gestion par les modules processus concernés des relations du squelette de contrôle avec les composants externes pose des problèmes. En effet, bien qu'autorisant une exécution performante, une telle gestion possède les inconvénients suivants :

- Le placement des module-processus, dans le cadre d'une architecture multi-processeurs, sera soumis aux contraintes, parfois contradictoires, des composants externes auquel ils accèdent.
- En cas d'échec du dialogue avec un composant, derrière une alternative, les retours en arrière (à l'Etat_Processus alternatif précondition de l'Action) sont complexes à gérer. Il faut rétablir l'état antérieur au niveau des Ressources éventuellement consommées.

C'est pourquoi l'unité de gestion des Ressources assure le dialogue avec les modules externes du système.

Les composants externes réversibles et non-réversibles sont connectés au modèle du système à l'aide de places d'interface auxquelles correspond l'appel d'une primitive. Les modules-processus se comportent comme des clients : ils ignorent si la Ressource à laquelle ils accèdent correspond à un dispositif externe au système ou non.

Chaque module est implémenté à l'aide de tâches. Cela permet de retrouver le parallélisme potentiel détecté durant la phase de décomposition.

Le contrôle d'une application parallèle est fréquemment réparti sur l'ensemble des processus quie la composent [Mullender 89].

Cependant, il existe dans le prototype tel que nous l'avons conçu, deux catégories de modules (Figure IV.5) :

- Les modules-processus, associés aux Processus issus de la décomposition du modèle. Ils exécutent les traitements modélisés dans le réseau de Petri. Ils sont *actifs*.

- Les modules de service, associés à la gestion des Ressources et des Actions synchronisées. Ils réalisent, sur demande des modules-processus, les communications dans le prototype. Ils sont *passifs*.

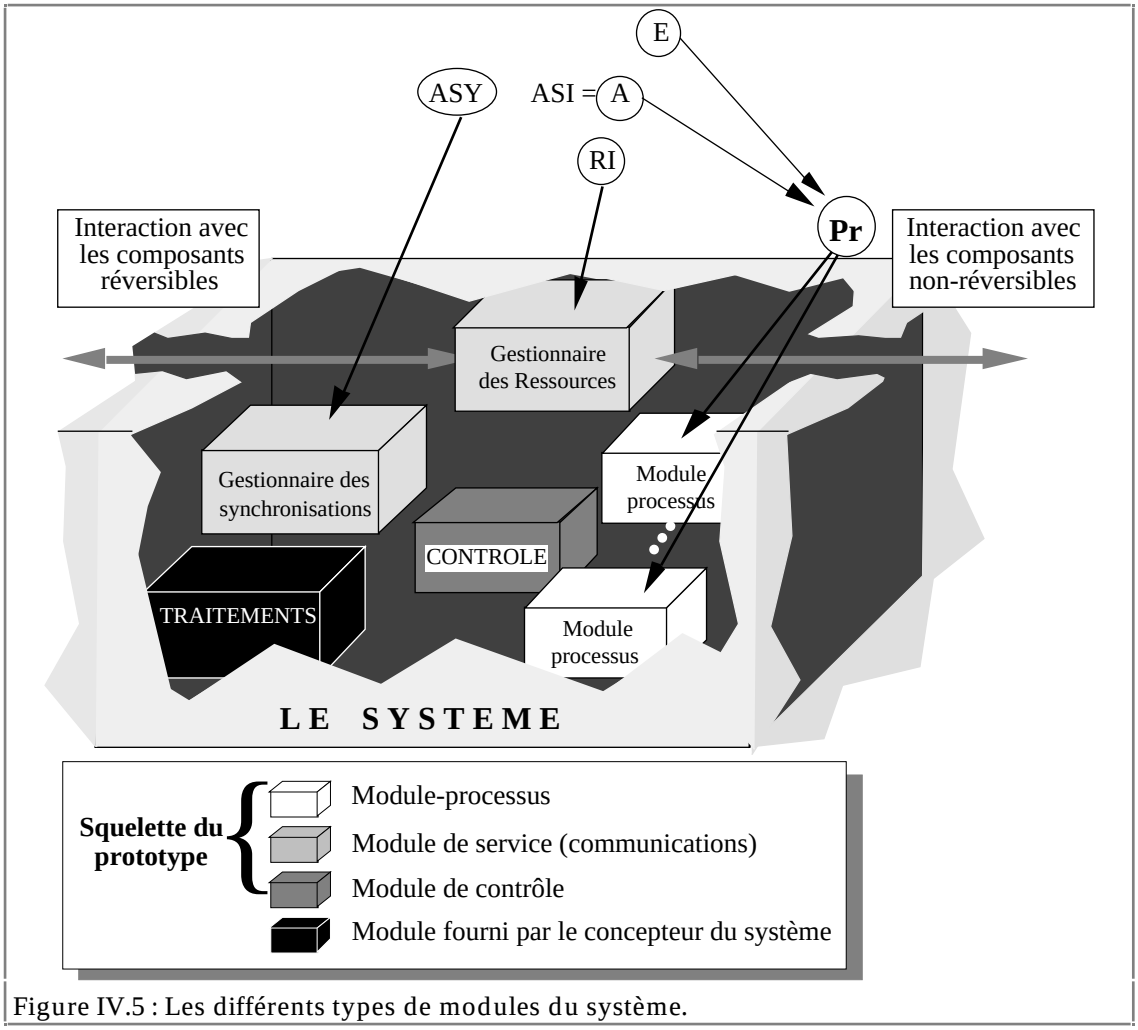


Figure IV.5 : Les différents types de modules du système.

Aucun module n'a de vision complète de l'état du fonctionnement de l'application [Raynal 87]. Le contrôle du prototype est donc confié à une unité dédiée qui intervient :

- A l'initialisation du prototype : elle initialise l'ensemble des modules en leur transmettant leurs paramètres de démarrage.
- Dans la gestion des cas d'exception : elle provoque la fin prématurée du prototype.
- A la terminaison du prototype : elle est chargée de comptabiliser les instances des processus qui disparaissent. Lorsque les modules-processus ont achevé leur exécution, elle termine le prototype.

Dans les sections suivantes, nous allons identifier les problèmes liés à la réalisation des différents modules composant le squelette. Nous proposons un ensemble de solutions d'implémentation.

2. Les modules-processus

Dans cette section, nous décrivons l'organisation des modules-processus. Un module-processus réalise le comportement d'un processus du réseau de Petri.

Considérons les relations entre un module-processus et les autres types de modules composant le prototype (Figure IV.6).

Deux modules-processus ne communiquent jamais directement entre eux mais uniquement par l'intermédiaire d'un module de service. Cela est lié à la façon dont nous avons décidé de décomposer le réseau de Petri en entités séquentielles communicantes.

Le module de contrôle initialise les différentes instances du processus. Chaque instance signale la fin de son exécution, qu'elle soit normale ou anormale.

Les liens avec le module de gestion des Ressources n'existent que si le processus comporte au moins une Action avec précondition ressource. Il en est de même avec le module de gestion des Actions synchronisées : la relation n'existe que si le processus se synchronise au moins une fois.

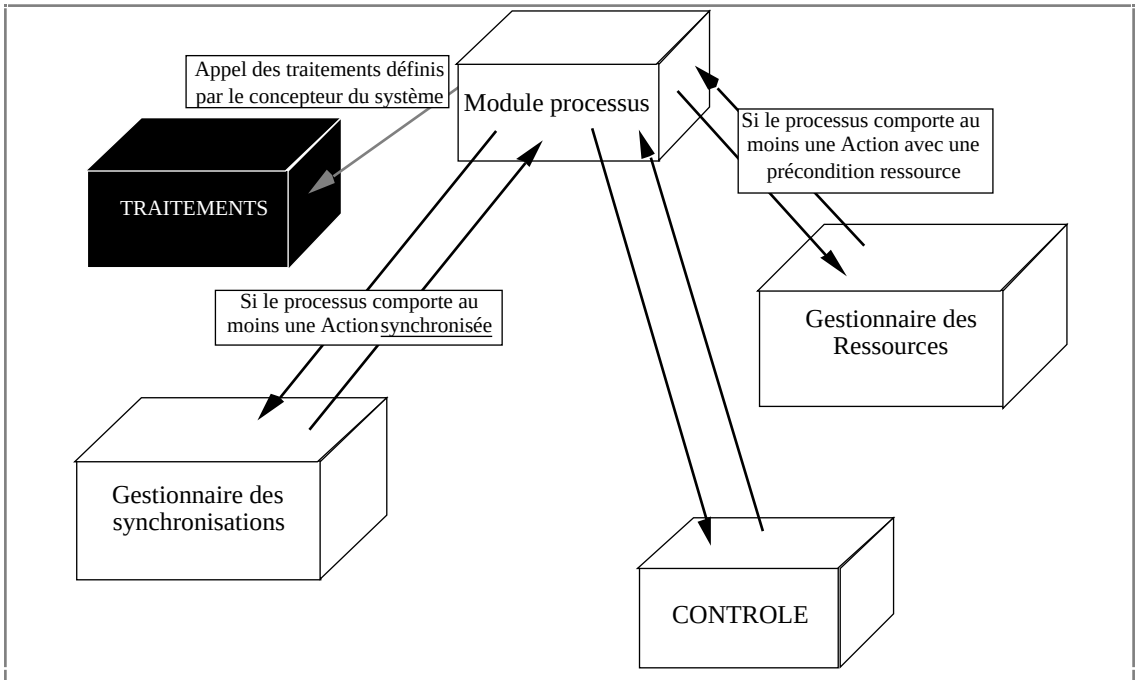


Figure IV.6 : Relation d'un module-processus avec les autres types de modules dans le prototype.

Aucun service n'est requis par le module définissant les traitements associés aux Actions puisqu'aucun dialogue n'a lieu. Ses relations avec les modules-processus sont unidirectionnelles.

Un Processus décrit un modèle de machine à états. Conformément aux propriétés 1, 2, 3 et 4 définies dans le Chapitre II, chaque Processus respecte les contraintes suivantes:

***Définition IV.7 :* Contraintes sur un Processus**

Chaque processus est un élément de Pr. C'est un graphe d'états finis dont les sommets appartiennent à E A. Nous avons

- Pr = $_{i=1..|Pr|}$ (A_i E_i), avec A = $_{i=1..|Pr|}$ A_i, E = $_{i=1..|Pr|}$ E_i et i, j 1.. |Pr|, $A_i \cap A_j$ = .
- pr Pr, pr = { $_{k=1..K}$ {p_k}}, avec, k 1..K, p_k A où p_k E.
- Les éléments p_k de pr sont reliés entre eux par des arcs orientés et valués conformément à ce qui est défini dans le réseau de Petri.

Des traitements sont associés aux sommets du graphe d'états finis. Nous distinguons deux types de sommets : ceux qui correspondent aux Actions et ceux qui correspondent aux Etats_Processus. Nous introduisons les définitions suivantes :

***Définition IV.8 :* L'ensemble des Actions**

L'ensemble des Actions de pr Pr, noté α, est défini comme suit :

- α (pr) = { $_{k=1..K}$ {p_k} pr | p_k A}.

***Définition IV.9 :* L'ensemble des Etats_Processus**

L'ensemble des Etats_Processus de pr Pr, noté ε, est défini comme suit :

- ε (pr) = { $_{k=1..K}$ {p_k} pr | p_k E}.

2.1. PROBLÈMES LIÉS À L'ACTIVATION D'UNE ACTION

Le concepteur d'un système associe des procédures aux transitions du réseau de Petri. Le graphe d'états définissant un Processus doit effectuer les appels conformément à ce qui est défini dans le modèle. Pour cela, il nous faut évaluer la précondition associée à la transition. La réalisation d'une Action se déroule en trois étapes :

- Evaluation de la précondition : cette étape est bloquante jusqu'à ce que l'Action soit activable. Cette évaluation n'a lieu que si l'Action est gardée.
- Réalisation de l'Action : il s'agit de l'appel de la procédure définissant les traitements associés.
- Réalisation du marquage postcondition s'il y a lieu.

2.1.1. Evaluation d'une précondition

Dans les réseaux de Petri Bien Formés, la précondition d'une transition est définie comme un ensemble de contraintes appliquées aux places précondition [Chiola 90, Dutheillet 91]. Afin d'être activable, le marquage des places précondition doit vérifier ces contraintes. Nous définissons formellement une précondition de la façon suivante :

***Définition IV.10 :* Précondition d'une Action Π (a)**

Soit Π (a) la précondition de l'action :

- Π (a) = (χ (a), { $_{k=1..K}$ {p_k} | p_k E R REN_s RER_s et un arc reliant p_k à a}) où χ (a) définit le prédicat associé à la transition, c'est-à-dire une expression booléenne définissant des contraintes sur l'ensemble des places précondition.

Les mécanismes d'évaluation d'une précondition sont comparables à ceux des requêtes de type bases de données [Gardarin 86]. Cela pose des problèmes d'optimisation des requêtes [Bellahsene 82].

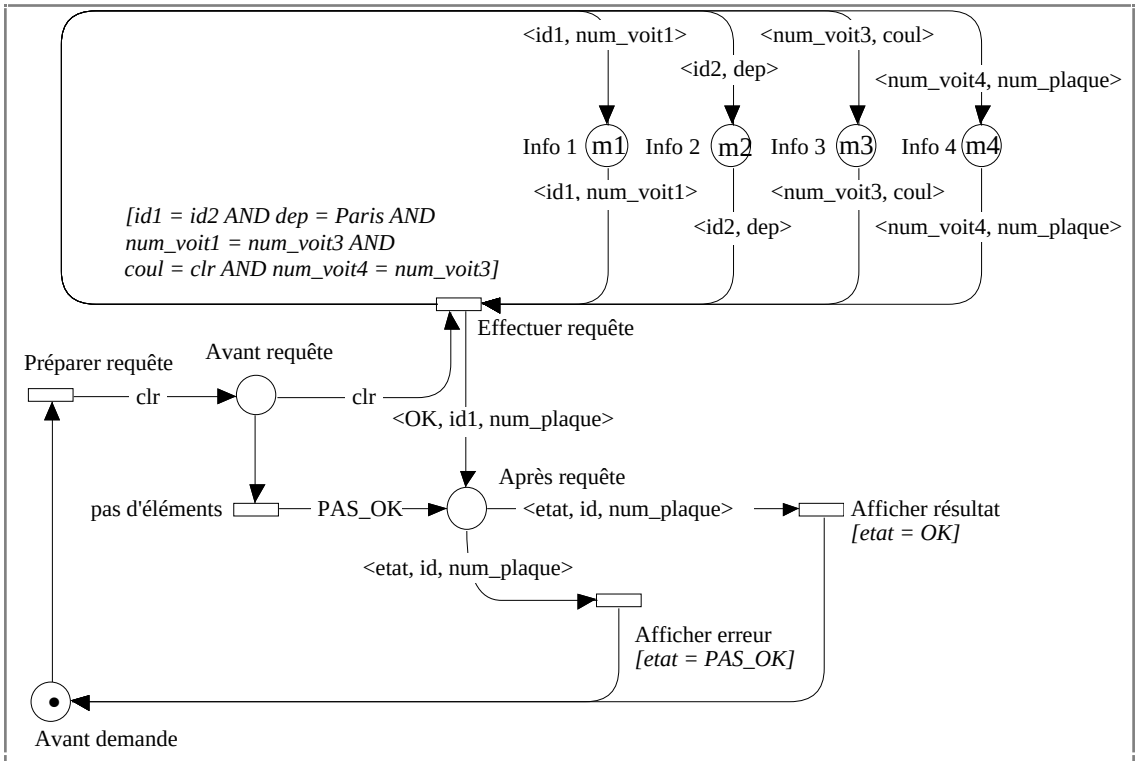


Figure IV.7 : Modèle simple associé à la réalisation d'une requête dans une base de données.

Exemple IV.1 : Considérons le problème lié à la réalisation d'une requête dans une base de données. La requête est : "récupérer le numéro d'immatriculation et l'identité d'un propriétaire d'une voiture d'une couleur donnée, dans la région parisienne". La couleur est définie à la préparation de la requête qui est effectuée dans une base composée de quatre éléments :

- le premier permet d'associer un numéro de voiture à son propriétaire;
- le deuxième permet d'associer un individu au département qu'il habite;
- le troisième permet d'associer une couleur à un numéro d'identification de voiture;
- le dernier permet d'associer un numéro d'identification d'une voiture à un numéro de plaque d'immatriculation.

Nous pouvons modéliser un tel système par le modèle donné en Figure IV.7. Ce modèle comporte un unique Processus défini par les Etats_Processus *Avant demande*, *Avant requête* et *Après requête*.

Les différentes composantes de la base de données sont définies par les places *Info 1*, *Info 2*, *Info 3* et *Info 4*. Les données contenues dans la base sont matérialisées par les marquages m1, m2, m3 et m4.

Si la transition *Effectuer requête* est activée, c'est qu'il existe dans la base de données au moins un article correspondant à la requête effectuée.

La précondition associée à l'Action *Afficher résultat* est définie par le couple ("le champ *etat* d'une marque de la place *Après requête* vaut OK", {"après requête"}).

Nous différencions, en fonction du type de place précondition de l'Action, deux catégories de préconditions : *processus* et *ressource*.

***Définition IV.11 :* Précondition processus**

Soit Π_p (a) la précondition processus de l'action a : Π_p (a) = (χ_p (a), { $_{k=1..K}$ {p_k} | p_k E et un arc reliant p_k à a} où χ_p (a) définit le prédicat associé à la transition concernant les places p | p E ∩ Π_p (a). χ_p (a) ne doit faire aucune référence à des marques (ou composantes de marques) contenues dans des Ressources.

***Définition IV.12 :* Précondition dégénérée**

Une précondition processus sans aucune contrainte sur les Etats_Processus en entrée est toujours vérifiée : nous la qualifions de *dégénérée*.

Pour les Actions simples, la précondition processus est réduite à une seule place tandis que pour les Actions synchronisées, elle en comporte plusieurs.

Une précondition processus postcondition d'un Etat_Processus simple doit être dégénérée. Dans le cas contraire, l'exécution du prototype peut se bloquer : en cas de non respect de la précondition processus, il est impossible de revenir à un point de choix.

***Définition IV.13 :* Précondition ressource**

Soit Π_r (a) la précondition ressource de l'action a : Π_r (a) = (χ_r (a), { $_{k=1..K}$ {p_k} | p_k R REN_s RER_s et un arc reliant p_k à a} où χ_r (a) définit le prédicat associé à la transition concernant les places p | p (R REN_s RER_s) ∩ Π_p (a). χ_r (a) fait référence à des constantes ainsi qu'à des marques (ou composantes de marques) contenues dans des Etats_Processus ou des Ressources.

Exemple IV.2 : Si nous considérons le modèle de la Figure IV.7, nous avons :

- Π (*Préparer requête*) = Π_p (*Préparer requête*) = ("n'importe quelle marque", {Avant demande}). Ici, la précondition est réduite à sa plus simple expression : une précondition processus dégénérée.
- Π (*Afficher erreur*) = Π_p (*Afficher erreur*) = ("couleur du champ *etat* d'une marque contenue dans *Après requête* vaut PAS_OK", {Après requête }). La précondition est réduite à une précondition processus non dégénérée.
- Π (*Effectuer requête*) = Π_a (*Effectuer requête*) = ("couleur du champ *id1* d'une marque contenue dans *Info 1* vaut couleur du champ *id2* d'une marque contenue dans *Info 2* **ET** couleur du champ *dep* d'une marque contenue dans *Info 2* vaut "Paris" **ET** couleur du champ *num_voit1* d'une marque contenue dans *Info 1* vaut couleur du champ *num_voit3* d'une marque contenue dans *Info 3* **ET** couleur du champ *coul* d'une marque contenue dans *Info 3* vaut une marque contenue dans *Avant requête* **ET** couleur du champ *num_voit4* d'une marque contenue dans *Info 4* vaut couleur du champ *num_voit3* d'une marque contenue dans *Info 3*", {Info 1, Info 2, Info 3, Info 4}). La précondition se ramène à une précondition ressource. En effet, la référence à un champ du mot d'état (la couleur de la marque contenue dans *Avant requête*) est vue comme une comparaison à une constante dont la valeur change dynamiquement en fonction de l'évolution du système.

L'évaluation d'une précondition processus n'implique pas les mêmes mécanismes que celle d'une précondition ressource. Dans un cas, elle est effectuée au niveau du processus, dans l'autre, elle est réalisée par le module de gestion des Ressources.

2.1.2. Réalisation du marquage postcondition

Nous définissons la postcondition d'une Action comme suit :

Définition IV.14 : **Postcondition**

Soit Θ (a) la postcondition de l'Action a, Θ (a) = { $_{k=1..K}$ {p_k, v_k} | p_k E R REN$_e$ RER$_e$ et un arc reliant a à p_k | Val (a, p_k) = v_k}.

Comme pour les préconditions, la postcondition d'une Action se divise en deux ensembles : la postcondition ressource et la postcondition processus.

Définition IV.15 : **Postcondition processus**

Soit Θ_p (a) la postcondition processus de l'Action a, Θ_p (a) = { $_{k=1..K}$ {p_k, v_k} | p_k E et un arc reliant a à p_k | Val (a, p_k) = v_k}.

La postcondition processus d'une Action <u>simple</u> référence toujours une seule place, tandis que celle d'une Action <u>synchronisée</u> en référence plusieurs. Nous avons toujours l'égalité :

$|\Pi_p (a)| = |\Theta_p (a)|$.

Définition IV.16 : **Postcondition ressource**

Soit Θ_r (a) la postcondition processus de l'Action a, Θ_r (a) = { $_{k=1..K}$ {p_k, v_k} | p_k R REN$_e$ RER$_e$ et un arc reliant a à p_k | Val (a, p_k) = v_k}.

La réalisation d'une postcondition ressource est effectuée par le module de gestion des Ressources. La postcondition processus ne concerne que le processus.

<u>Exemple IV.3</u> : Considérons la postcondition de l'Action *Effectuer requête* dans le modèle de la Figure IV.7 :

- Θ (*Effectuer requête*) = {(Info 1, "marque venant de *Info 1*"), (Info 2, "marque venant de *Info 2*"), (Info 3, "marque venant de *Info 3*"), (Info 4, "marque venant de *Info 4*"), (Après requête, "concaténation de : constante OK, champ *Id1* provenant de la marque consommée dans *Info 1*, champ *num_plaque* provenant de la marque consommée dans *Info 4*")}.

Cette Action comporte une postcondition processus Θ_p et une postcondition ressource Θ_r :

- Θ_p (E*ffectuer requête*) = {(Après requête, "concaténation de : constante OK, champ *Id1* provenant de la marque consommée dans *Info 1*, champ *num_plaque* provenant de la marque consommée dans *Info 4*")},
- Θ_r (*Effectuer requête*) = {(Info 1, "marque venant de *Info 1*"), (Info 2, "marque venant de *Info 2*"), (Info 3, "marque venant de *Info 3*"), (Info 4, "marque venant de *Info 4*")}.

2.2. ETAT_PROCESSUS

Il existe trois types d'Etats_Processus : <u>simples</u>, <u>alternatifs</u> ou <u>de terminaison</u>.

Définition IV.17 : **Les sous-ensembles d'Etats_Processus**

Nous avons $\varepsilon(pr) = \varepsilon_S(pr) \quad \varepsilon_T(pr) \quad \varepsilon_A(pr)$ avec :

- $\varepsilon_S(pr)$ est l'ensemble des Etats_Processus simples,
- $\varepsilon_T(pr)$ est l'ensemble des Etats_Processus de terminaison,
- $\varepsilon_A(pr)$ est l'ensemble des Etats_Processus alternatifs.

Conformément à la définition des Etats_Processus, donnée dans le Chapitre II :

- $\varepsilon_S(pr) \cap \varepsilon_T(pr) =$, $\varepsilon_S(pr) \cap \varepsilon_A(pr) =$ et $\varepsilon_T(pr) \cap \varepsilon_A(pr) =$.

Les Etats_Processus simples correspondent à une liaison entre deux Actions; aucun traitement ne peut leur être associé.

Les Etats_Processus de terminaison correspondent à un point d'arrêt dans le graphe d'états définissant le comportement du Processus. Nous intégrons les traitements liés à la terminaison du processus au niveau de l'Action précondition de l'Etat_Processus de terminaison.

Les Etats_Processus alternatifs correspondent à un point de choix dans le déroulement du Processus. Un traitement doit donc leur être associé.

Nous caractérisons de la sorte les Etats_Processus *neutres*, ainsi que la règle de traduction qui leur est applicable.

Définition IV.18 : **Etats_Processus neutres**

Nous avons $\varepsilon(pr) = \varepsilon_N(pr) \quad \varepsilon_A(pr)$ avec $\varepsilon_N(pr) = \varepsilon_S(pr) \quad \varepsilon_T(pr)$. $\varepsilon_N(pr)$ est l'ensemble des Etats_Processus neutres, $\varepsilon_A(pr)$ celui des Etats_Processus alternatifs.

Pour réaliser un Etat_Processus alternatif, plusieurs politiques sont envisageables :

- évaluation en parallèle des Actions potentielles [Hauschildt 87, Taubner 87] : on provoque l'évaluation des préconditions des N Actions postcondition de l'Etat_Processus en vue de choisir la première qui se réalise:
- choix a priori d'une Action de la postcondition de l'Etat_Processus [Cousin 88b, Kordon 89, Kordon 90].

Le choix de la première politique a les conséquences suivantes :

- Si, parmi les Actions postcondition de l'Etat_Processus, certaines sont synchronisées ou possèdent une précondition ressource, le Processus du modèle se comporte comme un client par rapport aux modules de service. Son comportement est passif; de ce point de vue, la politique est performante.
- Cependant, une disparité dans les préconditions des Actions postcondition d'un Etat_Processus peut en favoriser une. Cette politique n'est pas équitable, le prototype risque d'avoir un comportement pathologique.

Le choix de la seconde politique a les conséquences suivantes :

- Le choix effectué peut être mauvais : l'Action choisie n'est pas activable. En conséquence, il faut pouvoir revenir au point de choix afin d'éviter le blocage du prototype; le Processus n'a pas un comportement passif. De ce point de vue, cette politique est moins performante.
- Cependant, le choix d'une Action ne dépend que de la fonction de tirage. Les phénomènes comportementaux ne sont liés qu'au choix d'une politique. Le concepteur du modèle peut ainsi influer sur le comportement du prototype.

2.3. ACTIONS

Les Actions, telles qu'elles ont été définies, se divisent en deux classes sémantiquement différentes : les Actions simples qui sont exécutées pour le compte d'un processus instancié et les Actions synchronisées qui sont exécutées pour le compte de différents processus instanciés.

***Définition IV.19 :* L'ensemble des Actions**

L'ensemble α (pr) des Actions de pr Pr est partitionné de la façon suivante : $\alpha(pr) = \alpha_s(pr) \quad \alpha_p(pr)$ avec :

- $\alpha_s(pr) \cap \alpha_p(pr) =$,
- $\alpha_s(pr) = \{ \quad_{k=1..K} \{p_k\} \quad pr \mid p_k \quad \alpha(pr)$ et $p_k \quad ASY\}$,
- $\alpha_p(pr) = \{ \quad_{k=1..K} \{p_k\} \quad pr \mid p_k \quad \alpha(pr)$ et $p_k \quad ASI\}$.

2.3.1. Actions simples

La précondition d'une Action simple est toujours vérifiée. En conséquence, si l'Etat Actif d'un Processus correspond à une Action simple, il faut appeler la procédure associée puis générer le marquage postcondition s'il y a lieu.

2.3.2. Actions simples gardées

Une Action simple gardée possède une précondition qu'il faut évaluer afin de déterminer si elle est activable.

Comme l'évaluation de la précondition ressource implique des mécanismes externes, elle ne doit avoir lieu que si la précondition processus est vérifiée. Pendant l'évaluation d'une précondition ressource, le Processus est suspendu.

Si l'Action est postcondition d'un Etat_Processus simple, l'échec de l'évaluation d'une précondition processus pose un problème. Il est impossible de revenir en arrière afin d'exécuter une autre séquence de transition, le prototype se bloque. Le réseau de Petri décrivant le système n'est pas vivant. Ainsi, la précondition processus d'une Action simple gardée située derrière un Etat_Processus simple doit être dégénérée. Elle se résume donc à une précondition ressource : le Processus n'est réveillé que lorsqu'elle se réalise.

Si l'Action est postcondition d'un Etat_Processus alternatif, les traitements à produire dépendent de la politique choisie pour les Etats_processus alternatifs :

- Evaluation en parallèle des Actions potentielles : la précondition a déjà été évaluée au niveau du code associé à l'Etat_Processus alternatif. Le code associé à l'Action simple gardée se ramène à celui d'une Action simple.
- Choix a priori d'une Action potentielle : l'évaluation est effectuée au niveau du code associé à l'Action. Cependant, en cas d'échec prolongé, il faut pouvoir revenir au point de choix représenté par l'Etat_Processus alternatif : une attente perpétuelle de la réalisation d'une précondition peut générer des interblocages.

2.3.3. Actions synchronisées

La réalisation d'une Action synchronisée implique plusieurs processus : il s'agit d'un rendez-vous dont la gestion est confiée à un module dédié. Ce module évalue la précondition, lance l'Action dès que possible (son exécution n'a lieu qu'une seule fois, quel que soit le nombre de participants) puis gère la production du marquage postcondition.

Les Processus se comportent donc comme des clients attendant la réalisation d'un service.

Si l'Action est postcondition d'un Etat_Processus simple, le Processus n'est réveillé que lorsque la précondition ressource se réalise.

Si l'Action est postcondition d'un Etat_Processus alternatif, les traitements à produire dépendent de la politique choisie pour les Etats_processus alternatifs :

- Evaluation en parallèle des Actions potentielles : la demande de service est effectuée au niveau du code associé à l'Etat_Processus alternatif. En cas de succès, il faut passer à l'état processus successeur de l'Action. Aucun code n'est à générer pour l'Action synchronisée.
- Choix a priori d'une Action potentielle : l'évaluation est effectuée au niveau du code associé à l'Action. Le comportement à adopter en cas d'échec prolongé est similaire à celui décrit pour les Actions simples gardées : il faut revenir au niveau de l'Etat_Processus alternatif.

2.4. COMPORTEMENT D'UN PROCESSUS

Le comportement d'un processus est caractérisé par une machine à états : le graphe d'états actifs (GEA).

Définition IV.20 : **Etat actif d'un processus**

Sont considérés comme états actifs d'un processus l'ensemble des G-objets auxquels nous associons des traitements, c'est-à-dire :
- les Actions du modèle,
- les Etats_Processus alternatifs.

Définition IV.21 : **Graphe d'états actifs d'un processus**

Le graphe d'états actifs (GEA) d'un Processus relie les états actifs d'un processus entre eux. Il est obtenu à partir du graphe définissant un processus. GEA est défini comme suit :

- pr Pr, p pr, p GEA (pr) si et seulement si p ε_A (pr) A

La transformation permettant, à partir de Pr, d'obtenir GEA est définie comme suit . Soit p pr :

- si p ε_N (pr), alors q pré (p), Post (q) = Post (p),
- si p ε_N (pr), alors laisser tel quel.

Le graphe d'états actifs est directement implémenté dans le corps du modèle de tâche réalisant un processus.

Le compteur ordinal d'une instance de processus est représenté par son état actif courant. Le mot d'état est implémenté sous la forme d'une variable locale respectant le format donné par la structure des marques circulant dans les Etats_Processus.

Définition IV.22 : **Mot d'état d'un Processus**

Le mot d'état d'un processus, ψ (pr), est une variable composée. Chacun des champs de cette variable correspond à une référence à une marque (ou composante de marque) circulant dans les Etats_Processus du Processus. Le mot d'état est calculé et homogénéisé (les éléments qui le constituent sont définis en extension). Il suffit qu'une marque ou composante de marque soit référencée une fois pour apparaître dans le mot d'état. Nous avons :

Soit pr Pr, ψ (pr) = { $_{k=1..K}$ $\{v_k\}$ | il existe dans pr un arc connectant p et q (p α (pr) et q ε (pr) ou p ε (pr) et q α (pr)) | v_k val (p, q)}.

Exemple IV.4 : Considérons le modèle décrit dans Exemple III.6 et sa décomposition en processus (Figure IV.8).

Nous donnons en Figure IV.9 les graphes d'états actifs de ce modèle. Les places *Attendre*, *Conso1_2* et *Conso2_2* sont des Etats_Processus simples qui sont agglomérés à leurs prédécesseurs. Les places *Fin_prod*, *Fin_conso_1* et *Fin_conso_2* sont des Etats_Processus de terminaison; le traitement associé à ces terminaisons est intégré au niveau des transitions *Terminer_prod*, *Terminer_conso1* et *Terminer_conso2*.

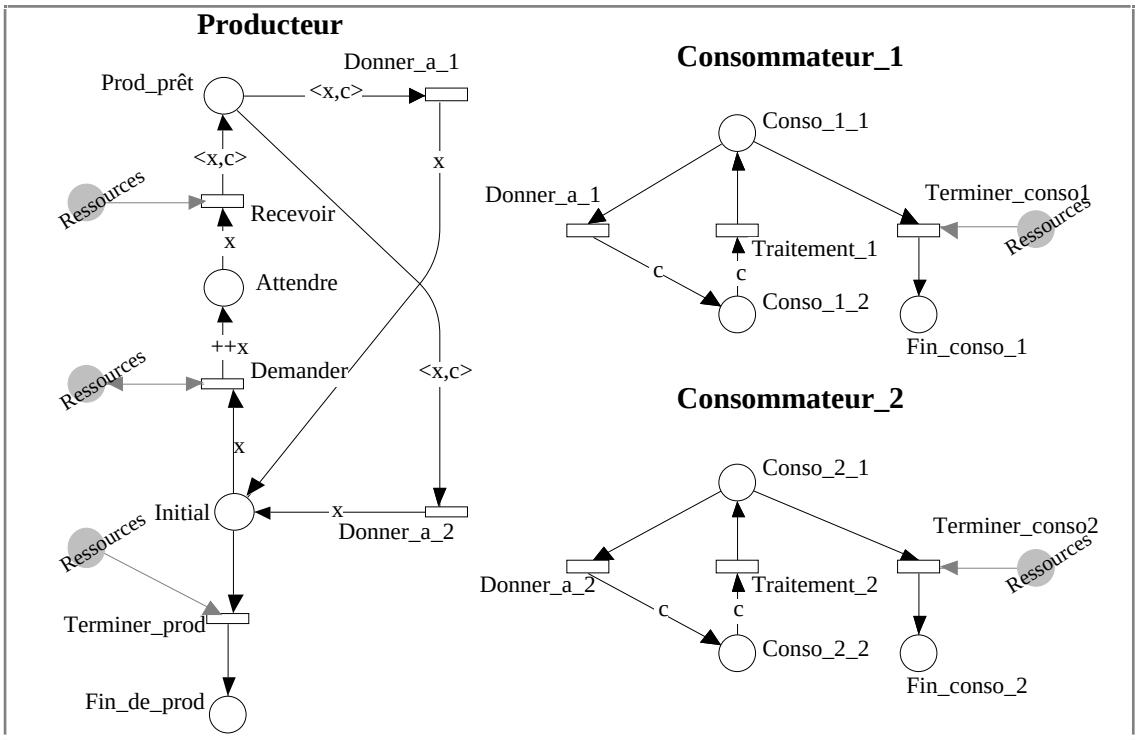


Figure IV.8 : Les Processus du modèle défini dans Exemple III.6.

La variable décrivant le mot d'état associé à *Producteur* est un article à deux champs (x et c). Les variables décrivant les mots d'états associés à *Consommateur_1* et *Consommateur_2* ne sont pas composées.

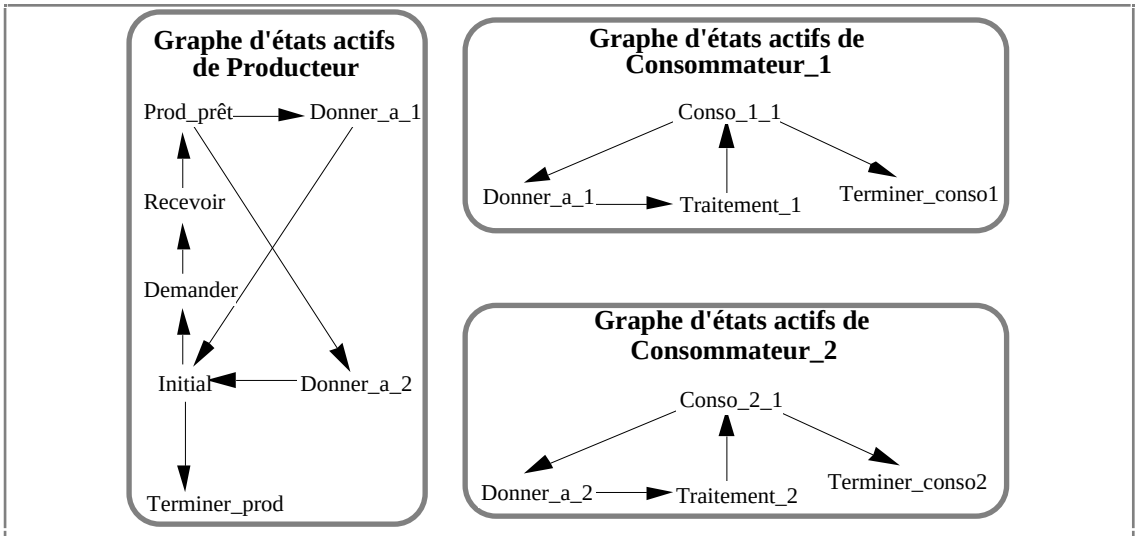


Figure IV.9 : Graphes d'états actifs associés à *Producteur*, *Consommateur_1* et *Consommateur_2*.

2.5. SERVICES REQUIS

Dans cette section, nous détaillons les services requis par les modules-processus : ils doivent être offerts par les modules de service (gestion des Ressources et des Actions synchronisées), le module de contrôle du prototype ou le module de traitement (Figure IV.10).

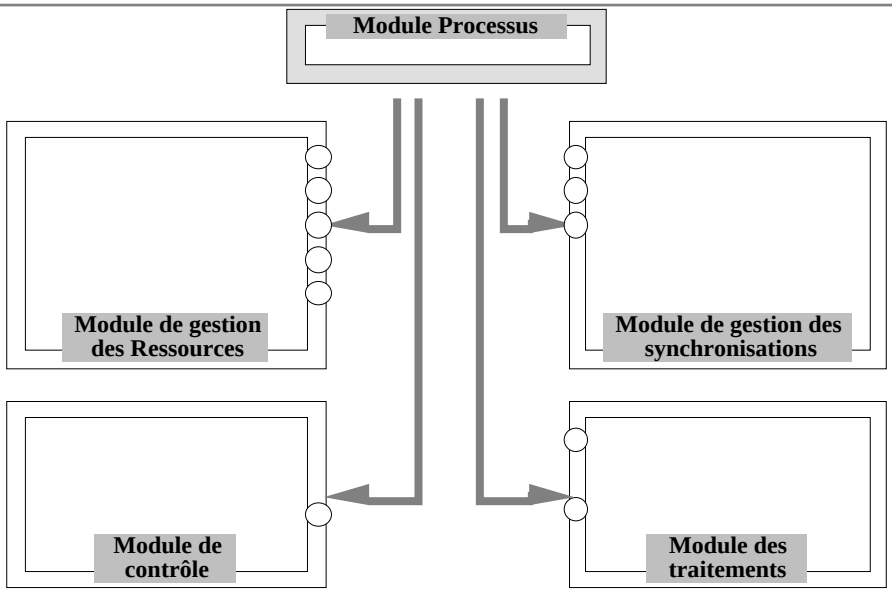


Figure IV.10 : Catégories de services requis par les modules-processus.

2.5.1. Services requis aux modules de service

Les services requis aux modules de service sont de deux types : bloquants et passants.

Les services bloquants sont :

- l'évaluation d'une précondition ressource, demandée au gestionnaire des Ressources;
- l'évaluation d'une Action synchronisée, demandée au gestionnaire des synchronisations.

La réalisation de ces services ne pose aucun problème dans les séquences d'Actions. Le client effectue une demande et attend sa réalisation. Le protocole est de type RPC [Nelson 81, Birell 84, Weihl 89]. Si un blocage se produit, il est dû à un modèle incorrect et, par conséquent, mal validé.

Le problème doit être envisagé différemment lorsque l'Action demandant l'évaluation d'un tel service se situe derrière un point de choix (elle est postcondition d'un Etat_Processus alternatif).

Nous avons évoqué deux algorithmes possibles pour l'implémentation des Etats_Processus alternatifs. Le choix de l'une ou l'autre des politiques a des conséquences sur les services requis par un module-processus. Ces services sont fournis par les modules de service (gestion des Ressources et gestion des Actions synchronisées).

Evaluation en parallèle des Actions postcondition

Le processus demande l'évaluation en parallèle de plusieurs préconditions en s'adressant au module de gestion des synchronisations et/ou de gestion des Ressources puis attend une réponse positive. La première Action réalisable sera choisie.

Une fois le choix effectué, il faut annuler les autres évaluations qui ont été lancées. Le module de service qui a répondu le premier doit avoir l'accord du client pour rendre effectif le service qui lui a été demandé : seule une Action doit être choisie.

Propriété attendue : Le protocole défini ne doit en aucun cas provoquer un blocage du système. Un client doit être servi dans des délais bornés.

Le protocole de dialogue est de type RPC [Nelson 81, Birell 84, Weihl 89] à ceci près que plusieurs clients sont contactés (Figure IV.11). A gauche, nous avons modélisé le comportement d'un Processus au niveau d'un Etat_processus alternatif. A droite, nous avons modélisé le comportement des modules de service. Les transitions partagées correspondent aux échanges, réalisés au moyen de primitives.

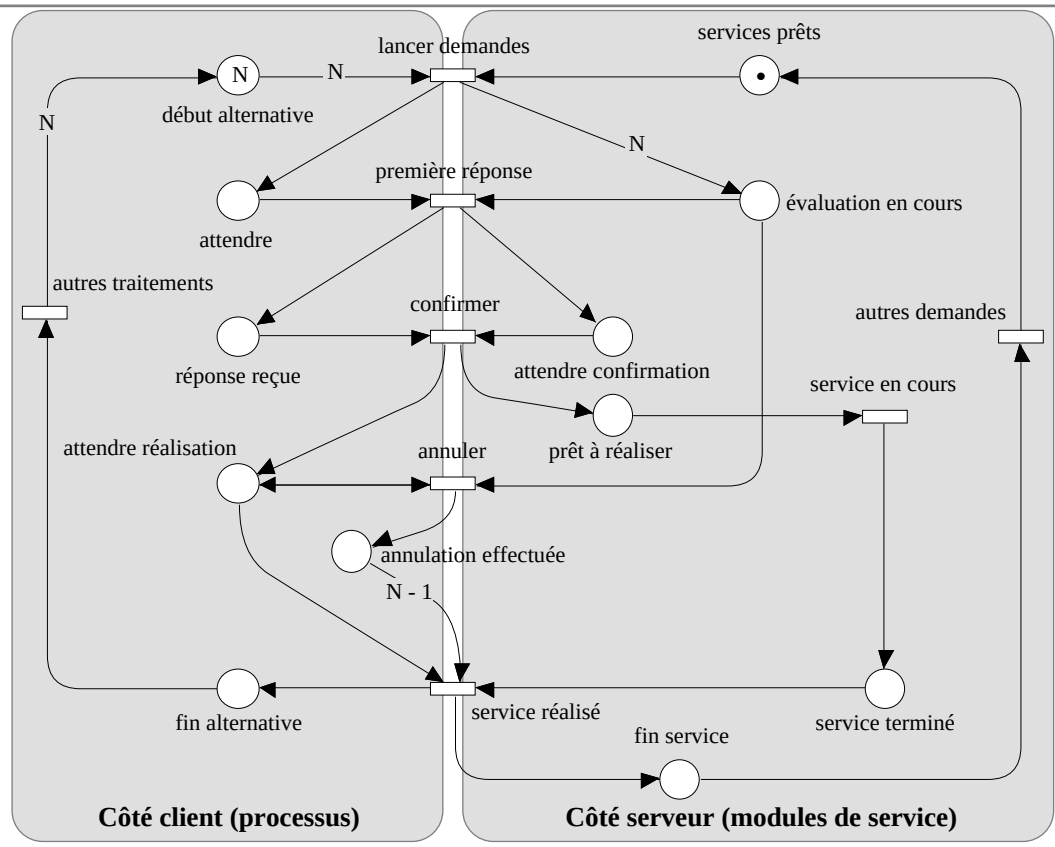


Figure IV.11 : Modèle du protocole de dialogue entre un module-processus et les modules de service dans le cadre de l'évaluation en parallèle des préconditions de toutes les Actions postcondition d'un Etat_Processus alternatif.

A l'état initial, nous avons, côté client, N Actions potentiellement réalisables (N marques). Les modules de service sont prêts à accepter les demandes.

Dans un premier temps, le client lance les évaluations puis se met en attente d'une réponse. La première réponse obtenue détermine l'Action choisie. Il confirme sa demande auprès du serveur puis annule toutes les autres évaluations, qu'elles soient réalisées ou non. Enfin, le client attend la réalisation effective du service qu'il a demandé.

A la fin du traitement associé à l'Etat_Processus alternatif, une Action a été choisie.

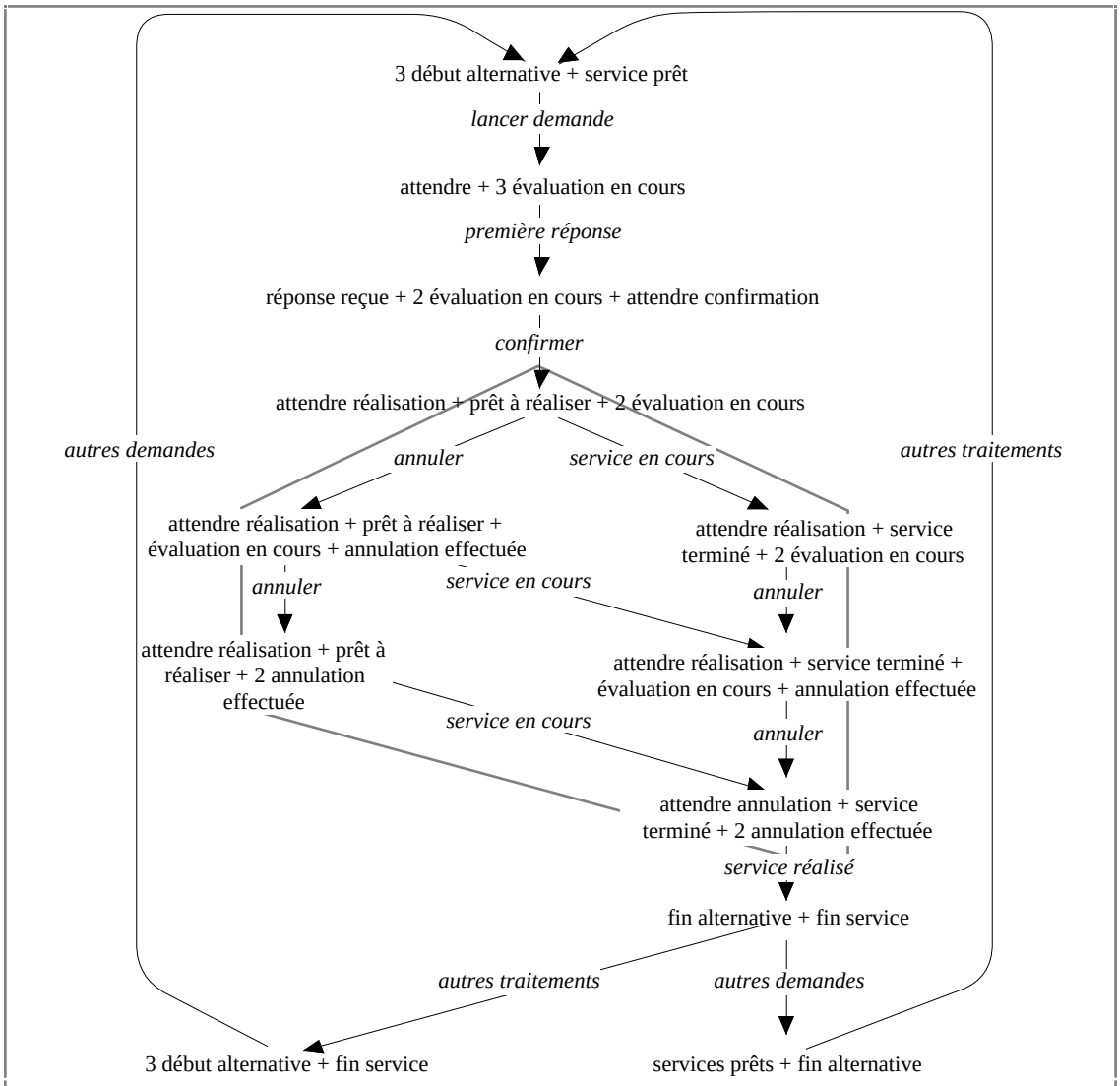


Figure IV.12 : Graphe des marquages accessibles du modèle de la Figure IV.11 pour N = 3.

Validation : Le graphe des marquages accessibles pour N = 3 (trois Actions sont postcondition d'un Etat_Processus alternatif) est donné en Figure IV.12.
Si nous faisons varier la constante N, seule la partie centrale change; elle se décline en fonction du nombre de combinaisons entres les N - 1 annulations et la réalisation du service choisi.
Le marquage initial constitue un état d'accueil pour ce modèle puisque tous les chemins du graphe font évoluer le modèle dans cette direction. Le réseau de Petri modélisant le protocole est vivant :
- le serveur ne se bloquera jamais,
- les clients seront toujours servis.

Les propriétés attendues pour ce protocole sont vérifiées; il est donc valide.

Les services nécessaires à la réalisation du protocole correspondent aux transitions de dialogue entre serveur et client. Nous nous intéressons ici aux services requis, c'est-à-dire :

- `LANCE_EVALUATION` : il permet à un client soit de demander l'évaluation d'une précondition ressource (service offert par le gestionnaire des

Ressources), soit de signaler qu'il est prêt à effectuer une Action synchronisée donnée (service offert par le gestionnaire des synchronisations);

- `ANNULE` : il permet à un client d'annuler une demande;
- `CONFIRME` : il permet à un client de confirmer une demande.

Choix a priori d'une Action postcondition

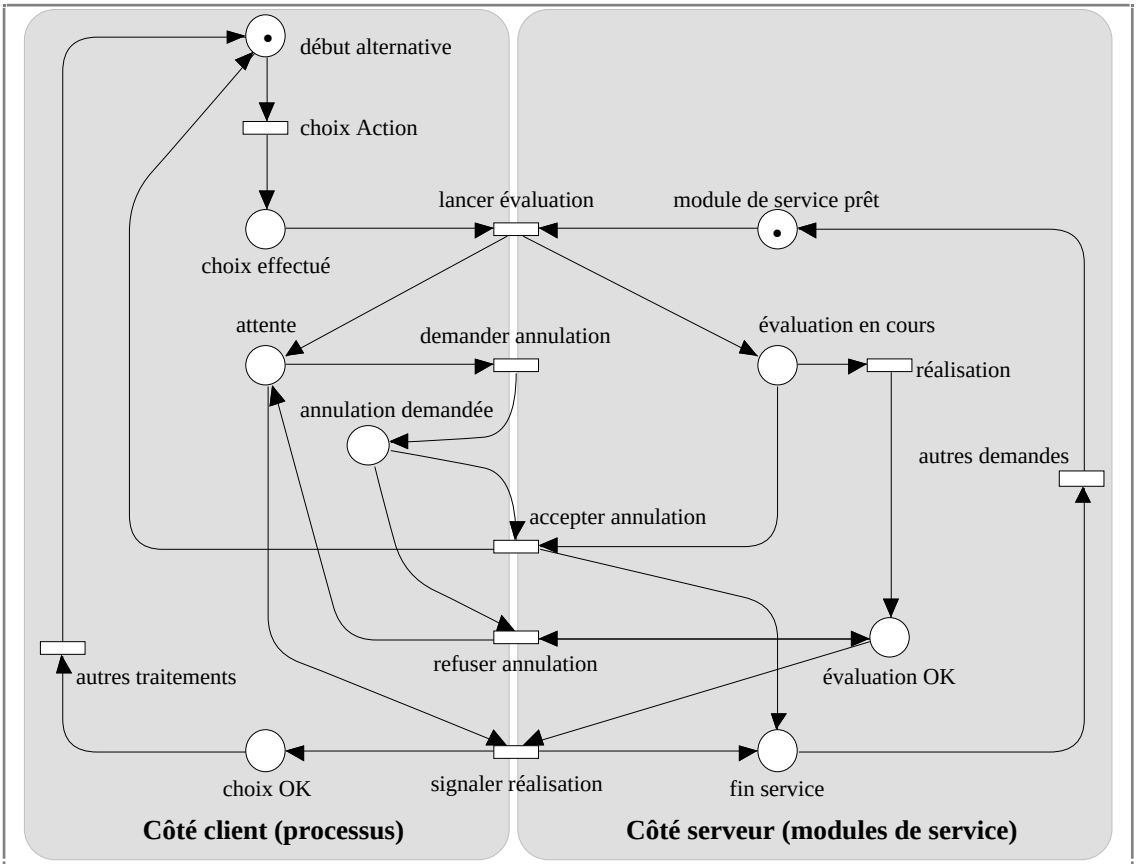


Figure IV.13 : Modèle du protocole de dialogue entre un module-processus et les modules de service dans le cadre du choix a priori d'une Action postcondition d'un Etat_Processus alternatif.

Contrairement à la politique précédente, un Processus est complètement maître du choix d'une Action postcondition d'un Etat_Processus alternatif. Comme l'Action choisie *a priori* peut ne pas être activable, il faut pouvoir revenir au point de choix et tenter une autre Action.

Propriété attendue : La décision d'annuler une demande doit être acceptée par le module de service : les croisements de messages potentiels (le serveur signale que le service est résolu alors que le client demande une annulation) ne doivent pas provoquer un blocage du client.

Le protocole de dialogue est de type RPC avec, dans certains cas, une possibilité d'annulation [Nelson 81, Birell 84, Weihl 89] (Figure IV.13). A gauche, nous avons modélisé le comportement d'un Processus au niveau d'un Etat_processus alternatif. A droite, nous avons modélisé le comportement des modules de service. Les transitions partagées correspondent aux échanges, réalisés au moyen de primitives.

Dans un premier temps, le processus choisit une Action dans la postcondition de l'Etat_Processus. Il effectue une demande d'évaluation au module de service concerné (gestion des Actions synchronisées ou gestion des Ressources) puis se met en attente d'une temporisation.

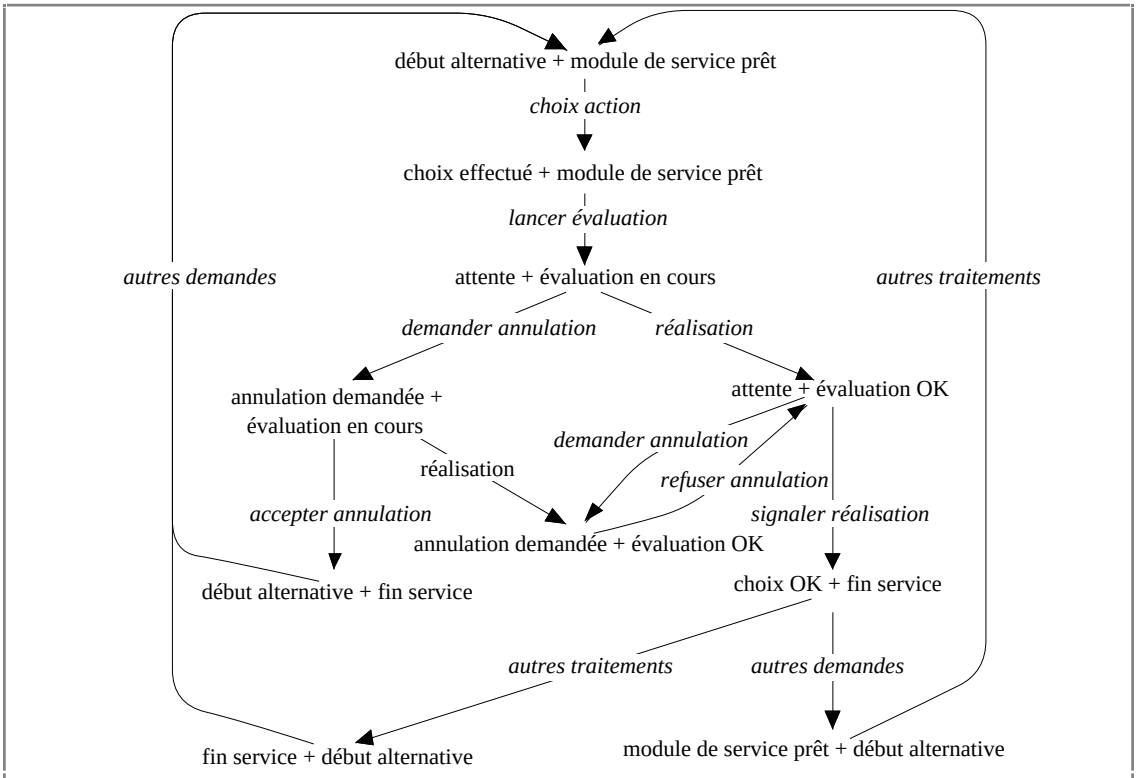


Figure IV.14 : Graphe des marquages accessibles du modèle de la Figure IV.13.

Si le module de service répond par l'affirmative avant expiration de la temporisation, l'Action est choisie et le processus reprend son exécution. Dans le cas contraire, le processus tente d'annuler la demande qu'il a effectuée. Cette annulation est soumise à approbation du module de service qui ne la refuse que si la demande a déjà été traitée avec succès.

Si l'annulation est acceptée, le processus effectue un autre tirage. Dans le cas contraire, il se remet en attente : la réponse est inéluctable.

Validation : Le graphe des marquages accessibles est donné en Figure IV.14.

Le marquage initial constitue un état d'accueil pour ce modèle puisque tous les chemins du graphe font évoluer le modèle dans cette direction. Le réseau de Petri modélisant le protocole est vivant.

Le serveur et les clients ne se bloqueront jamais, même s'il y a croisement des demandes d'annulation et des réponses du module-serveur. Les propriétés attendues pour ce protocole sont vérifiées; il est donc valide.

Les services nécessaires à la réalisation du protocole correspondent aux transitions de dialogue entre serveur et client. Nous nous intéressons ici aux services requis, c'est-à-dire :

- `LANCE_EVALUATION` : il permet à un client soit de demander l'évaluation d'une précondition ressource (service offert par le gestionnaire des Ressources), soit de signaler qu'il est prêt à effectuer une Action

synchronisée donnée (service offert par le gestionnaire des synchronisations);

- ANNULE : il permet à un client d'annuler une demande.

Production d'une postcondition ressource

Le gestionnaire des Ressources doit offrir un service passant - la production d'une postcondition ressource - appelé PRODUIRE.

Manipulation du résultat obtenu

Enfin, le gestionnaire des Ressources doit offrir un ensemble de services permettant la manipulation du résultat d'une consommation. Cela peut être nécessaire pour construire une postcondition ressource ou modifier le mot d'état du Processus. Ces services sont regroupés dans MANIPULER_RES_PRECOND.

2.5.2. Services requis au module de contrôle

Le prototype est piloté par le module de contrôle qui doit offrir un service INSTANCE_TERMINEE permettant à un module-processus de signaler qu'il termine son exécution.

Un paramètre doit permettre d'indiquer si l'instance se termine normalement ou à la suite d'une erreur survenue dans l'exécution d'une primitive du module des traitements.

2.5.3. Services requis au module des traitements

Le module de traitement doit fournir l'ensemble des primitives définissant les traitements associés aux transitions du réseau de Petri. Le service correspondant est utilisé lorsqu'une Action simple est activée.

2.6. SERVICES OFFERTS

Dans cette section, nous détaillons les services offerts par les modules-processus : ils appartiennent à deux catégories (Figure IV.15) :

- les services liés au dialogue avec les modules de service,
- les services liés au contrôle du prototype.

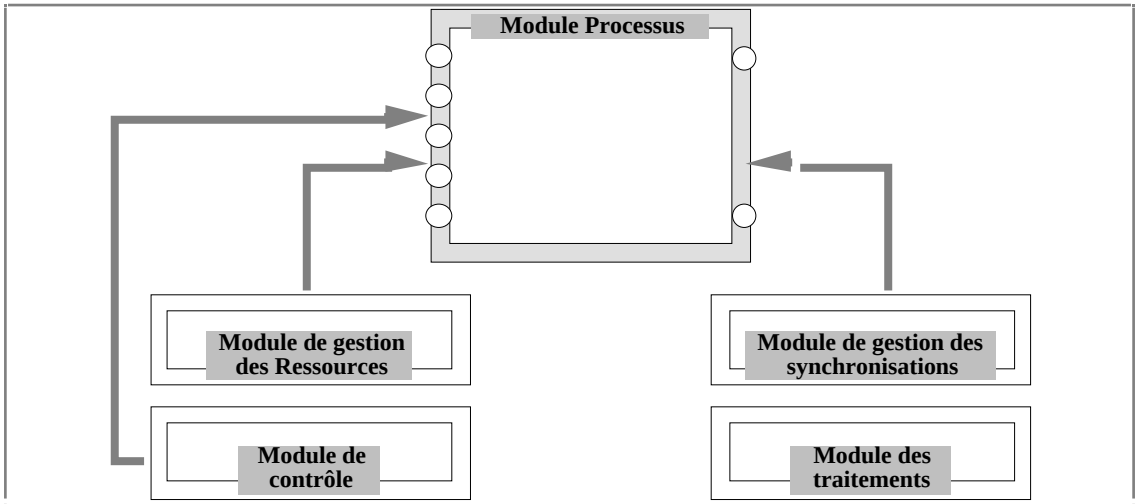


Figure IV.15 : Catégories de services offerts par les modules-processus.

2.6.1. Services offerts aux modules de service

Dans le cadre du dialogue avec les modules de service, quelle que soit la politique d'implémentation des Etats_Processus alternatifs, les modules-processus doivent offrir des services de type `REPONSE`, permettant aux modules de service de transmettre des résultats (Figures IV.11 et IV.13). Ces services doivent pouvoir être utilisés dans les contextes suivants :

- évaluation en parallèle des Actions potentielles :
  - pour signaler qu'une Action est activable;
  - pour signaler que le service confirmé a été réalisé.
- choix a priori d'une Action potentielle :
  - pour signaler qu'une précondition ressource est réalisée;
  - pour signaler qu'une Action synchronisée a été activée;
  - pour permettre à un serveur d'accepter ou de refuser l'annulation d'une évaluation de précondition ou d'une tentative de synchronisation.

Nous verrons, dans la section 4, que le gestionnaire des Actions synchronisées peut avoir besoin de manipuler le mot d'état d'un Processus. Ce mot d'état étant une structure privée, il faut fournir les primitives permettant d'écrire, ou de lire, une composante du mot d'état. Ces primitives sont regroupées sous la dénomination `MANIPULER_MOT_D_ETAT`.

2.6.2. Services offerts au module de contrôle

A l'initialisation de l'application, le module de contrôle doit créer les différentes instances du processus, puis les initialiser une par une. En cas d'exception, il provoque la terminaison prématurée de l'ensemble des instances du processus. Les services offerts sont donc :

- `CREE_LES_INSTANCES` : pour créer dynamiquement un ensemble d'instances;
- `INITIALISE_UNE_INSTANCE` : pour initialiser une instance, c'est-à-dire lui transmettre son mot d'état initial et son Etat_processus initial;

- DETRUIT_LES_INSTANCES : pour détruire l'ensemble des instances encore actives, quel que soit leur état (en attente d'un service...).

Nous verrons, dans la section 5, qu'à des fins d'initialisation, le module de contrôle doit manipuler le mot d'état du Processus. Le service MANIPULER_MOT_D_ETAT est donc également offert au module de contrôle.

2.7. SYNTHÈSE

Nous récapitulons dans cette section les grandes lignes de notre technique de réalisation des modules-processus. Pour cela, il faut considérer deux aspects :

- les interfaces : elles sont définies par les impératifs de coopération avec les autres modules du prototype;
- le comportement : il est défini par la structure du processus, c'est-à-dire l'ordonnancement des G-objets qui le composent.

2.7.1. Les interfaces du module

La Figure IV.16 récapitule les services requis et offerts par les modules-processus.

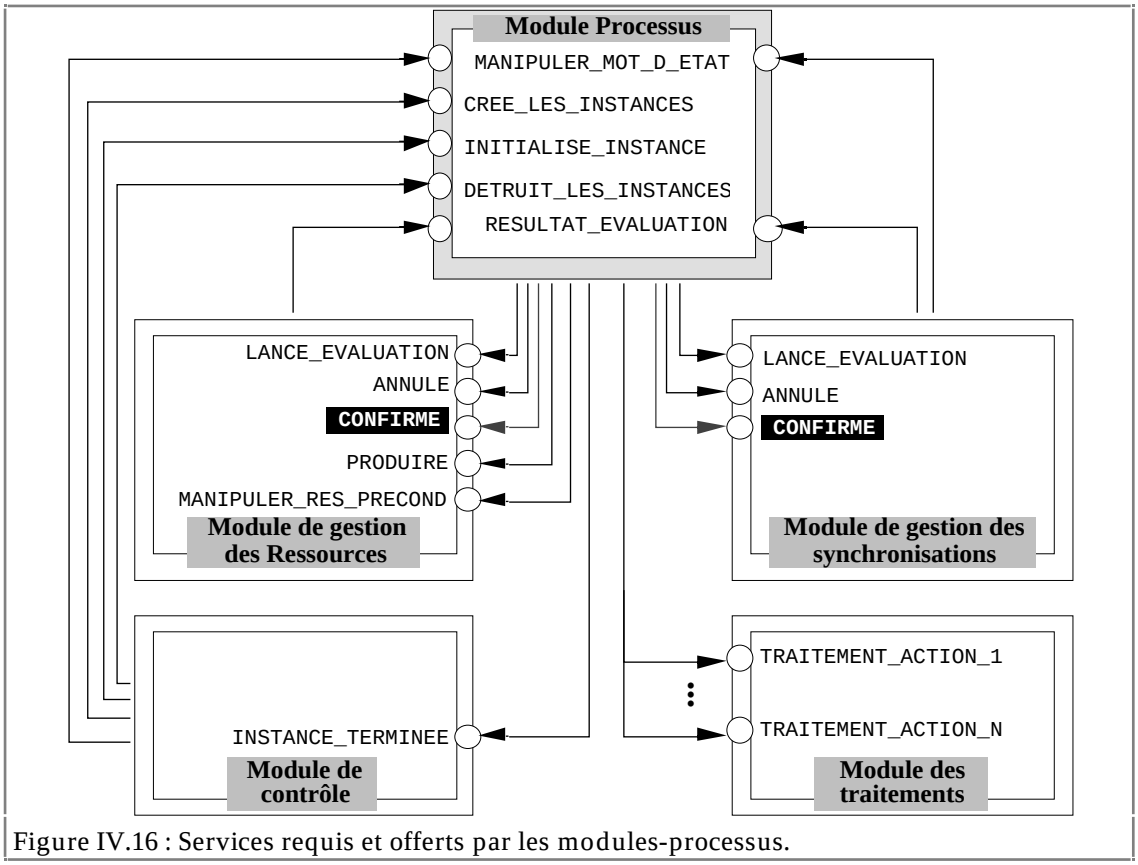


Figure IV.16 : Services requis et offerts par les modules-processus.

Certains services permettent aux modules-processus d'échanger des informations avec leur homologues, soit via le gestionnaire des Ressources, soit via le gestionnaire des synchronisations. D'autres sont dédiés au contrôle du prototype. Le service MANIPULER_MOT_D_ETAT permet aux autres modules de travailler sur une structure propre à chaque module-processus.

Aucun service n'est offert au module de traitement, ce dernier a un rôle complètement passif puisqu'il contient les traitements associés aux Actions.

Les services liés au dialogue entre les modules-processus et les modules de service dépendent de la politique choisie pour l'implémentation des Etats_Processus alternatifs, le service `CONFIRME` (indiqué en "inverse") n'existant que s'il y a évaluation en parallèle des Actions situées derrière un point de choix.

2.7.2. Génération du comportement

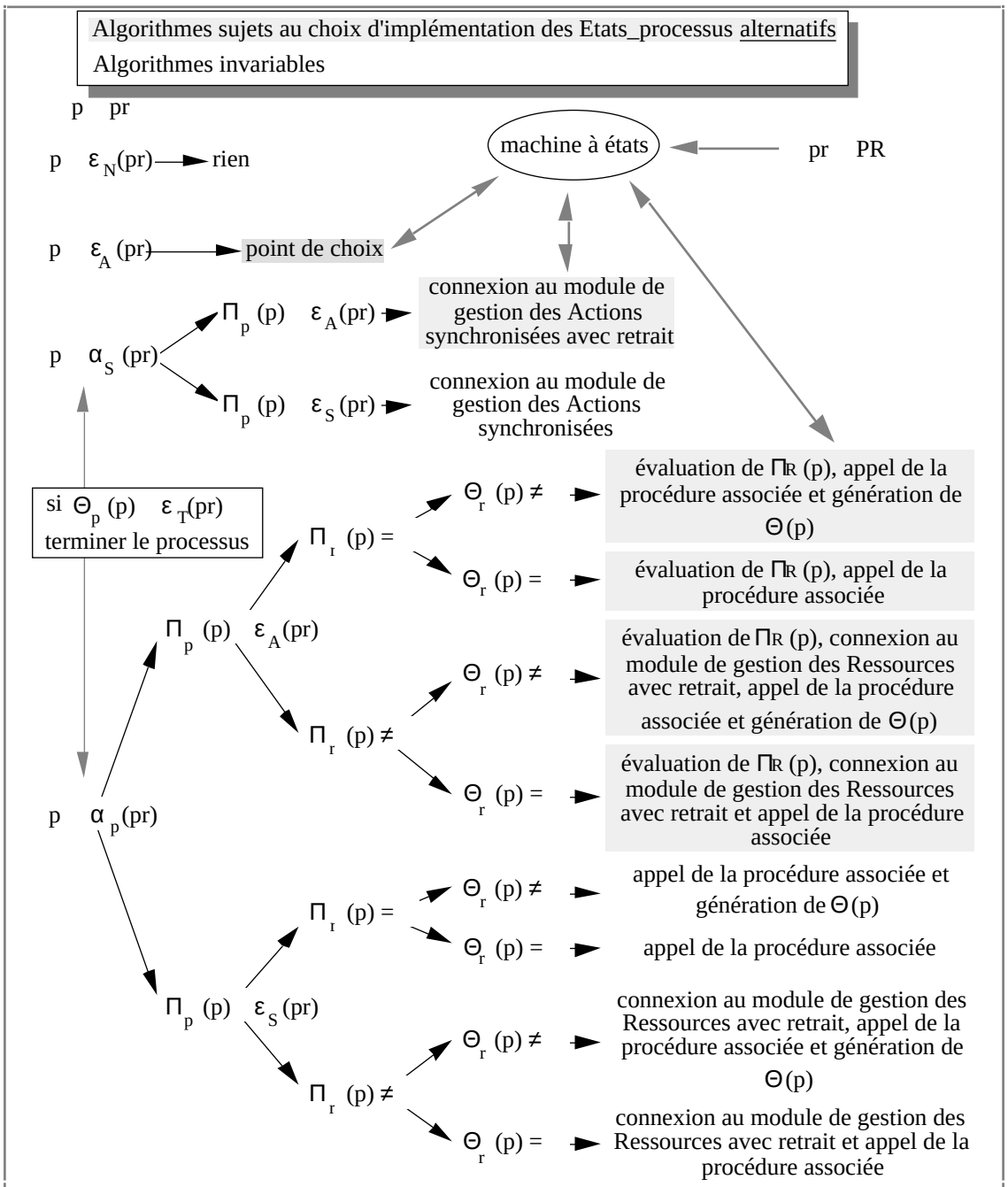


Figure IV.17 : Schéma d'association de structures de contrôle type à chaque G-objet.

Nous récapitulons dans cette section les différents types d'algorithmes générés pour chaque G-objet.

Chaque processus est réalisé sous la forme d'un modèle de tâche implémentant une machine à états. Ce modèle de tâche sera dynamiquement instancié en fonction du nombre de processus instanciés.

A chaque G-objet composant le processus est associé un traitement type que nous décrivons brièvement en Figure IV.17. Les algorithmes en eux-mêmes sont donnés en Annexe A.

3. Le module de gestion des Ressources

Le module de gestion des Ressources gère les communications asynchrones dans le prototype. Il réalise des services pour le compte de clients :

- les modules-processus : au cours de leur exécution, s'ils doivent évaluer une précondition ou produire un marquage postcondition;
- le module de gestion des Actions synchronisées : il doit pouvoir évaluer une précondition (pour les Actions synchronisées gardées) ou produire un marquage postcondition;
- le module de contrôle du prototype : il doit produire un marquage initial au début de l'exécution.

La Figure IV.18 décrit les relations entre le gestionnaire des Ressources et les autres modules composant le prototype. Il existe un lien avec chaque module-processus si ce dernier contient au moins une Action ayant, soit une précondition ressource, soit une postcondition ressource.

De même, il existe un lien avec le module de gestion des Actions synchronisées s'il existe au moins une Action synchronisée ayant une précondition ou une postcondition ressource.

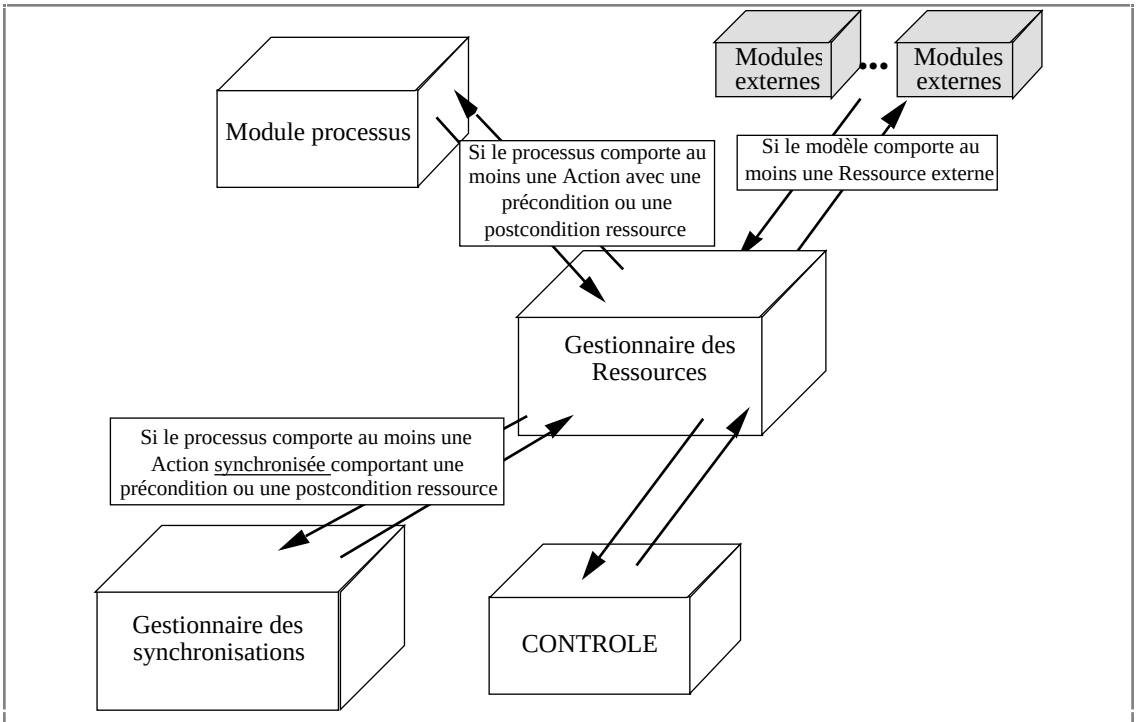


Figure IV.18 : Relations du gestionnaire des Ressources avec les autres types de modules dans le prototype.

Le module de contrôle est un client particulier : il gère le marquage initial du prototype, à l'initialisation du module de gestion des Ressources. Lorsque les modules processus ont achevé leur exécution (il n'existe plus de processus instanciés), il doit également provoquer le terminaison de la gestion des Ressources.

Enfin, si le modèle comporte des Ressources externes, le module de gestion des Ressources est lié aux modules externes correspondants. Ces derniers sont issus des composants externes (réversibles ou non-réversibles).

Après avoir abordé les problèmes liés à la communication avec les modules externes, nous détaillons les services offerts et requis par le gestionnaire des Ressources. Nous discutons ensuite des architectures possibles du module.

3.1. GESTION DES INTERFACES AVEC L'ENVIRONNEMENT

Nous détaillons, dans cette section, les problèmes liés à la gestion des interfaces avec l'environnement du prototype, c'est-à-dire la manipulation des Ressources externes liées aux composants externes réversibles ou non-réversibles.

Les Ressources externes sont manipulées par le biais de primitives d'accès, qui doivent être définies pour chacun des composants externes concernés dans le langage cible. Comme nous considérons un prototype parallèle, une première contrainte peut être définie sur ces primitives :

Définition IV.23 : **Contrainte sur les primitives associées aux Ressources externes**

> Toutes les primitives de manipulation associées à des Ressources externes, liées à des composants externes réversibles ou non-réversibles, sont réentrantes.

En fonction du flot d'utilisation des Ressources externes (en entrée, en sortie) liées à des composants externes réversibles ou non-réversibles, des contraintes supplémentaires doivent être définies.

Dans un premier temps, nous décrivons la production de marques dans les Ressources externes en entrée. La consommation de marques depuis une Ressource externe en sortie met en œuvre des mécanismes différents, selon qu'elle est liée à un composant externe réversible (elle est incluse dans RER_s) ou non-réversible (elle est incluse dans REN_s).

3.1.1. Production dans une Ressource externe en entrée

Lorsqu'un module-client produit une marque dans une Ressource externe (qui est forcément une interface en entrée), le module de gestion des Ressources effectue l'appel de la primitive associée dans le module externe qui active un dispositif dont le squelette ignore la structure (Figure IV.19).

Nous définissons les contraintes suivantes sur les primitives associées aux Ressources externes en entrée.

***Définition IV.24 :* Contraintes sur les primitives associées aux Ressources externes en entrée**

Une primitive associée à une Ressource externe en entrée est :

- Une procédure : si la Ressource contient des marques colorées, elle comporte un ou plusieurs paramètres en mode *in*[5] dont les types sont définis par la classe ou le domaine de couleurs de la place.
  Si la Ressource contient des marques non colorées, elle ne comporte aucun paramètre.
- Passante : elle doit rendre la main le plus rapidement possible au module de gestion des Ressources afin de préserver le parallélisme du prototype.

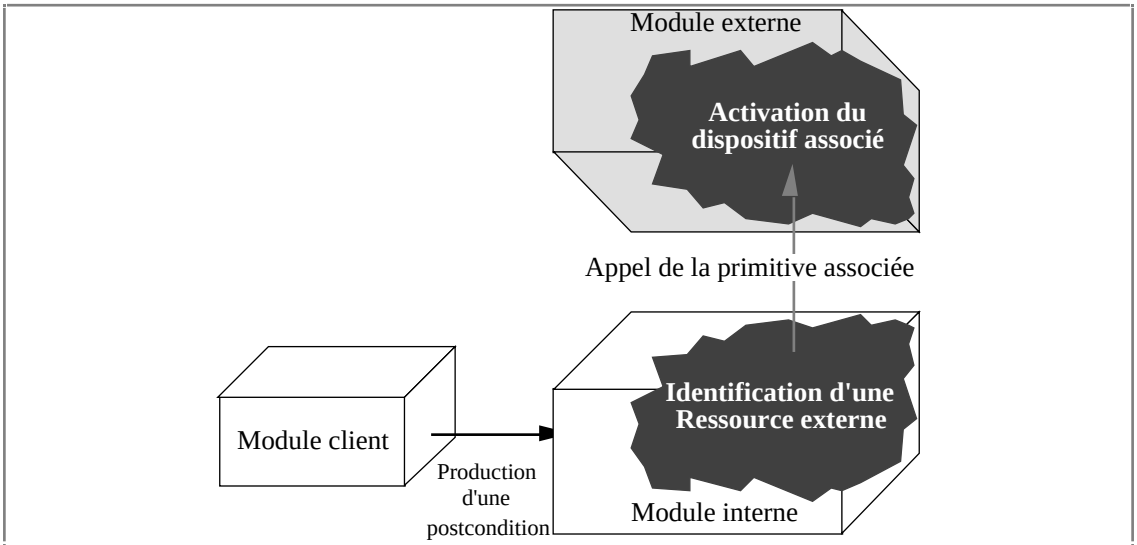


Figure IV.19 : Production dans une place Ressource externe.

3.1.2. Consommation dans une Ressource externe liée à un composant externe non-réversible

Le gestionnaire des Ressources identifie l'accès aux Ressources externes et appelle la primitive associée (Figure IV.20). Cette primitive doit rendre un résultat quoi qu'il arrive : l'évaluation d'une précondition ne doit pas rester en suspens car cela risque d'engendrer des blocages. Ce résultat est donc :

- une marque : la précondition référençant la Ressource externe est vérifiée, le client est réveillé;
- pas de marque : la précondition référençant la Ressource externe échoue.

Les contraintes suivantes doivent être respectées :

***Définition IV.25 :* Contraintes sur les primitives correspondant aux Ressources externes en sortie associées aux composants externes non-réversibles**

La primitive associée à une Ressource externe en sortie liée à un composant externe non-réversible est :

5 Au sens Ada du terme, c'est-à-dire des paramètres par valeur.

- Une procédure : si la Ressource contient des marques colorées, elle comporte un ou plusieurs paramètres en mode *out*[6] dont les types sont définis par la classe ou le domaine de couleurs associé. Un paramètre supplémentaire, de type booléen, indique si une marque a pu être consommée.
  Si la Ressource n'a pas de domaine de couleurs, l'unique paramètre consiste en ce booléen.
- Passante : elle doit rendre la main le plus rapidement possible au module de gestion des Ressources afin de ne pas bloquer l'évaluation d'une précondition. Si aucun résultat ne peut être obtenu, une convention "pas de valeur disponible" doit être utilisée.

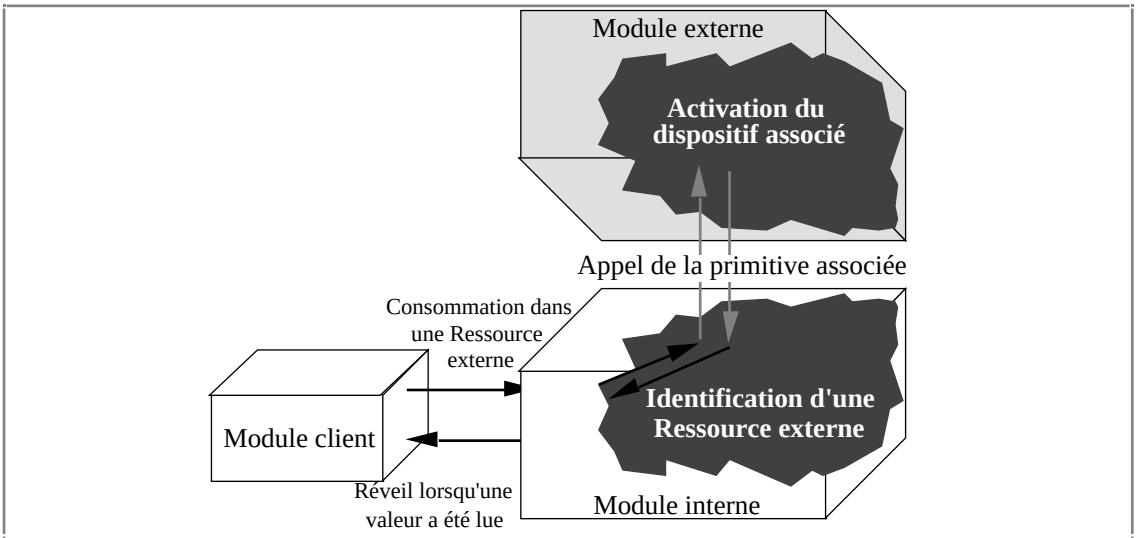


Figure IV.20 : Consommation d'une marque dans une Ressource externe en sortie d'un composant externe non-réversible.

3.1.3. Consommation dans une Ressource externe liée à un composant externe réversible

Le module de gestion des Ressources identifie la Ressource externe et appelle la primitive associée (Figure IV.21). Un dialogue doit permettre au gestionnaire des Ressources de connaître le marquage de la Ressource externe. Ce dialogue s'effectue en deux phases :

- Demande de la liste des "marques" contenues dans la Ressource. Les marques ainsi transmises sont réservées.
- Restitution des "marques" qui n'ont pas été utilisées après analyse du contenu par le gestionnaire des Ressources.

Le module de gestion des Ressources peut donc, le temps de l'évaluation d'une précondition, posséder une copie du marquage de plusieurs Ressources externes liées à des composants externes réversibles. Deux primitives sont associées à ces Ressources externes en sortie.

6 Au sens Ada du terme, c'est-à-dire des paramètres en "write only".

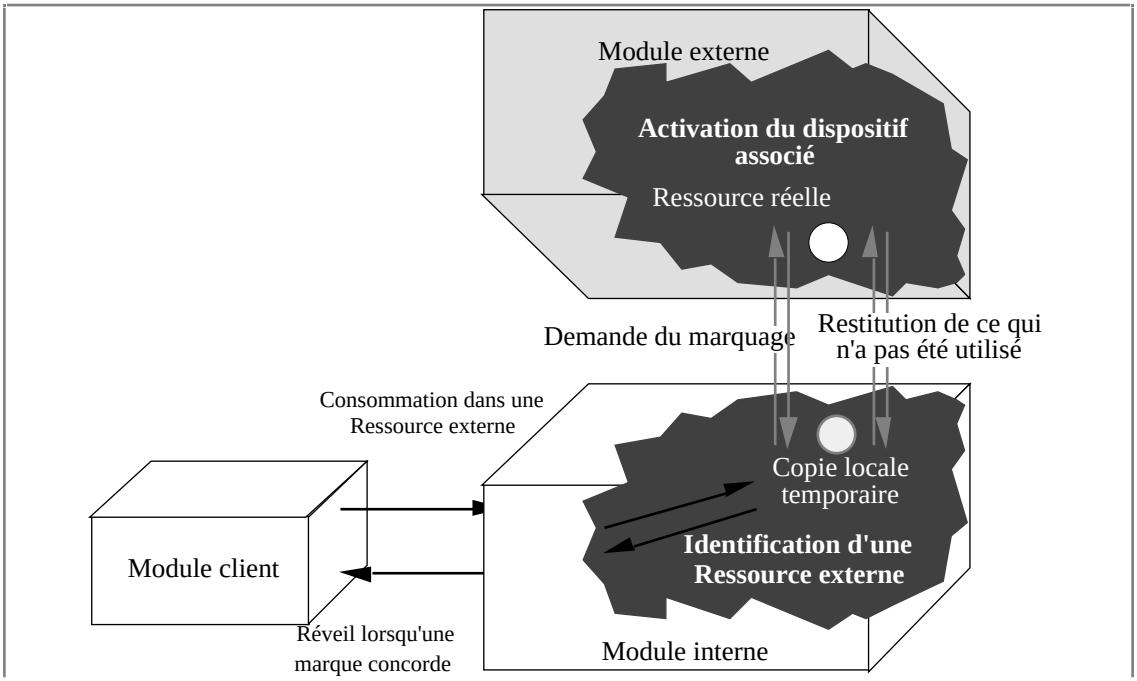


Figure IV.21 : Consommation dans une Ressource externe en sortie associée à un composant externe réversible.

***Définition IV.26 :* Contraintes sur les primitives correspondant aux Ressources externes en sortie associées aux composants externes réversibles**

Deux primitives sont associées à chaque Ressource externe en sortie liée à un composant externe réversible :

- La première est une procédure. Si la Ressource contient des marques colorées, elle comporte un paramètre en mode *out*[7] ramenant le marquage contenu dans la Ressource externe.
  Si la Ressource contient des marques non colorées, un compteur indique le nombre de marques disponibles.
  Cette procédure doit être passante, afin de ne pas bloquer l'évaluation d'une précondition.
- La seconde est également une procédure. Elle comporte un paramètre en mode *in*[8] permettant, soit de "restituer" les marques qui n'ont pas été utilisées (Ressource colorée), soit d'indiquer le nombre de marques non consommées (Ressource non colorée).
  Cette procédure doit être passante afin de ne pas perturber l'exécution du prototype.

3.1.4. Traitement des événements positifs

L'évaluation d'une précondition ressource est un service bloquant. Il faut suspendre l'exécution du client jusqu'à sa réalisation. La production d'un marquage postcondition peut réveiller une précondition dont l'évaluation a auparavant échoué : nous appelons cela un *événement positif*.

7 Au sens Ada du terme, c'est-à-dire un paramètre en "write only".

8 Au sens Ada du terme, c'est-à-dire un paramètre passé par valeur.

Définition IV.27 : **Evénement positif**

Tout enrichissement du marquage d'une Ressource est considéré comme un événement positif :

- pour les Ressources internes, la production d'une postcondition ressource est un événement positif;
- pour les Ressources externes en sortie associées à des composants externes réversibles, la "production", dans le composant externe, d'un "marquage", ou la restitution par un gestionnaire de Ressources de marques non utilisées constituent des événements positifs.

En cas d'échec de leur évaluation, les préconditions ressources, qu'elles comportent des références à des Ressources externes (selon les règles définies au Chapitre III) ou non, sont suspendues jusqu'à ce qu'un événement positif ou une annulation survienne.

La gestion des événements positifs doit être disponible, pour les Ressources internes comme pour les Ressources externes. Dans le premier cas, ils sont gérés de façon interne; dans le second cas, ils le sont au moyen d'un service particulier pouvant être réalisé de deux manières différentes :

- Il est offert par le module de gestion des Ressources. En cas d'événement positif, il est spécifié que le composant externe doit faire appel à ce service. L'implémentation s'apparente à celle d'une interruption logicielle [Krakowiak 85].
- Il est offert par chaque module externe. Dans ce cas, le module de gestion des Ressources interroge régulièrement les modules pour lesquels il attend des demandes.

La première solution est la plus intéressante du point de vue de l'exécution; la gestion des Ressources externes devient similaire à celle des périphériques dans les systèmes d'exploitation.

La seconde solution peut cependant se justifier pour des raisons d'implémentation. En effet, la conception des modules externes (description interne du composant) n'a pas à tenir compte de spécifications liées à la structure du prototype.

Définition IV.28 : **Traitement des événements positifs liés aux Ressources externes**

La gestion des événements positifs liés aux Ressources externes doit être effectuée :

- soit par appel d'une primitive offerte par le module de gestion des Ressources,
- soit via une primitive offerte permettant au module de gestion des Ressources d'interroger le module externe.

3.2. SERVICES OFFERTS

Dans cette section, nous détaillons les services offerts par le module de gestion des Ressources (Figure IV.22).

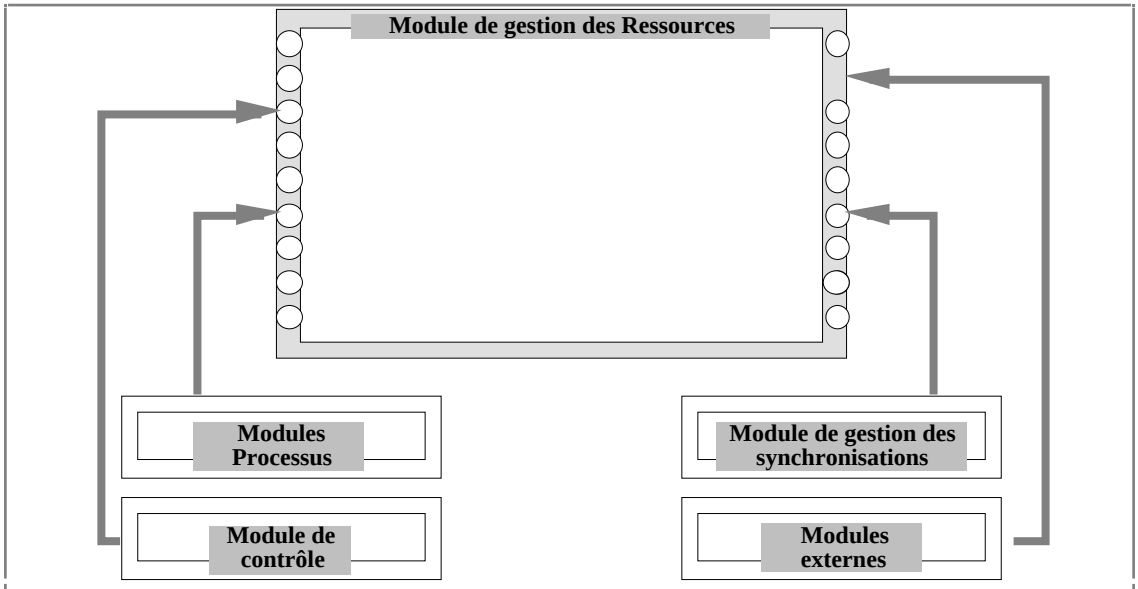


Figure IV.22 : Catégories de services offerts par le module de gestion des Ressources.

L'ensemble des marques contenues dans les Ressources d'un réseau correspondent à une base de données dont l'état conditionne le lancement des Actions du prototype.

Si le modèle est coloré, il faut pouvoir gérer des marques typées. Ces marques peuvent être composées : ce sont des N-uplets.

Dans le cas où le réseau de Petri n'est pas coloré, seul le nombre de marques contenues dans chaque Ressource compte. Un réseau de Petri non coloré est équivalent à un réseau de Petri coloré dont l'unique domaine de couleurs ne possède qu'un seul élément.

Les services permettent de résoudre différentes catégories de problèmes. Certaines sont liées au formalisme utilisé pour exprimer les spécifications, comme l'évaluation d'une précondition ressource ou la production d'une postcondition ressource. Les autres sont issues de l'architecture du prototype.

3.2.1. Evaluation d'une précondition ressource

L'évaluation d'une précondition ressource est équivalente à une requête dans une base de données. Il faut consommer des marques dans un ensemble de Ressources tout en respectant les relations établies au niveau du prédicat de l'Action concernée. C'est à ce prix que nous pourrons lancer les procédures conformément à ce qui est spécifié dans un modèle *coloré*.

Nous pouvons aisément discerner les opérations liées aux bases de données : jointures, sélections et restrictions [Gardarin 86].

***Définition IV.29 :* Caractérisation d'une jointure**

Les marques consommées dans une Ressource dépendent de celles consommées dans d'autres Ressources :

Soit pr Pr, a α_p (pr), r_i et r_j Π_r (a) | $r_i \neq r_j$ avec r_i et r_j contenant des marques composées. Si dans χ_r (a) une relation du type "v_i op v_j" avec v_i référençant un champ d'une marque de r_i et v_j référençant un champ d'une marque de r_j, alors la précondition ressource est équivalente, pour les places considérées, à une jointure sur leur contenu.

Définition IV.30 : **Caractérisation d'une sélection**

Les marques consommées doivent avoir un certain profil :

Soit pr Pr, a α_p (pr), r Π_r (a). Si dans χ_r (a) une relation du type "v op valeur" avec v, référençant un champ d'une marque de r (ou une marque entière de r), alors la précondition ressource est équivalente, pour la place considérée, à une sélection sur son contenu.

Définition IV.31 : **Caractérisation d'une restriction**

Seules certaines composantes d'une marque composée (un N-uplet) sont utilisées dans la postcondition :

Soit pr Pr, a α_p (pr), r Π_r (a). Si dans Π_r (a) v référençant une marque, ou une composante d'une marque de r, tel que v n'est pas référencée dans Θ (a), alors, il y a restriction sur les marques consommées dans la précondition ressource.

Ces opérations peuvent être combinées. Elles ne sont pas réalisées au même niveau :

- les sélections et les jointures sont traitées par le module de gestion des Ressources : elles sont liées à l'évaluation de la précondition;
- la restriction est effectuée au niveau du module-processus concerné : elle est liée à l'utilisation du résultat fourni par le module de gestion des Ressources.

Exemple IV.5 : Considérons les Actions de la Figure IV.23. *EP_1* et *EP_2* sont des Etats_Processus, *R_1* et *R_2* des Ressources contenant des marques colorées. Nous pouvons assimiler les préconditions des Actions *Tjoin*, *Trestr* et *Tsel* à des requêtes de bases de données.

- **Exemple I** : l'évaluation de la précondition associée à *Tjoin* nécessite la réalisation d'une jointure entre l'ensemble des marques de *R_1* et l'ensemble des marques de *R_2*. Une marque sera arbitrairement choisie dans le résultat puis stockée dans le mot d'état du processus.
- E**xemple II** : l'évaluation de la précondition associée à *Tsel* nécessite la réalisation d'une sélection sur l'ensemble des marques de *R_1*. Une marque sera arbitrairement choisie dans le résultat puis stockée dans le mot d'état du processus.
- **Exemple III** : Au moment de la réalisation de *Trestr*, on effectue une restriction puisque seuls les champs a et c du N-uplet extrait de *R_1* sont affectés dans le mot d'état.

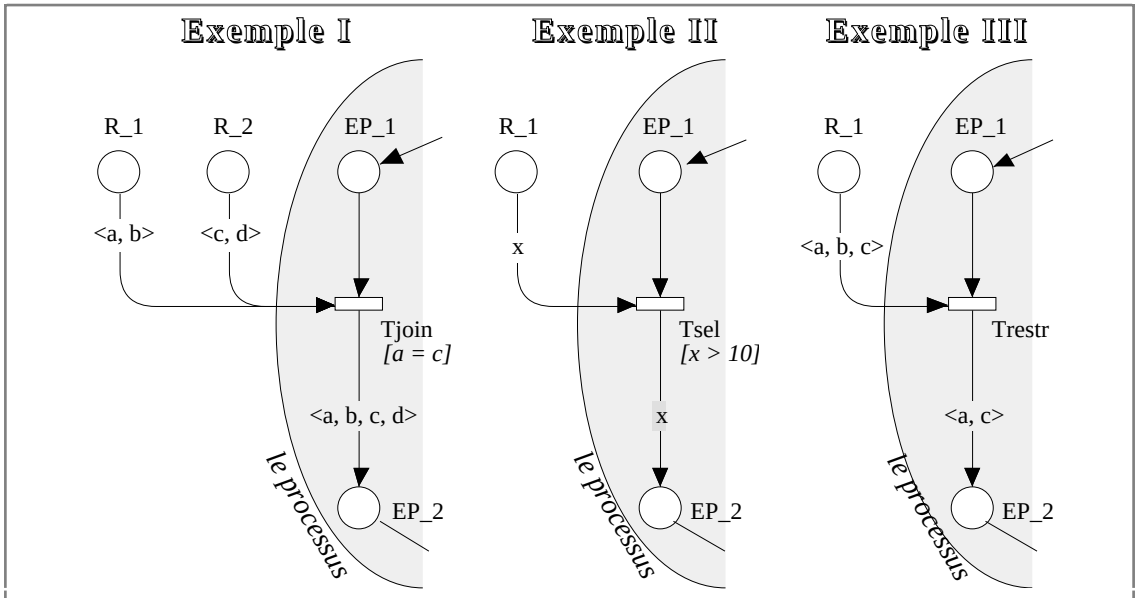


Figure IV.23 : Interprétations de préconditions ressource en terme de requêtes de type bases de données.

La précondition ressource d'une Action est divisée en deux parties : la *précondition contrainte* et la *précondition libre*.

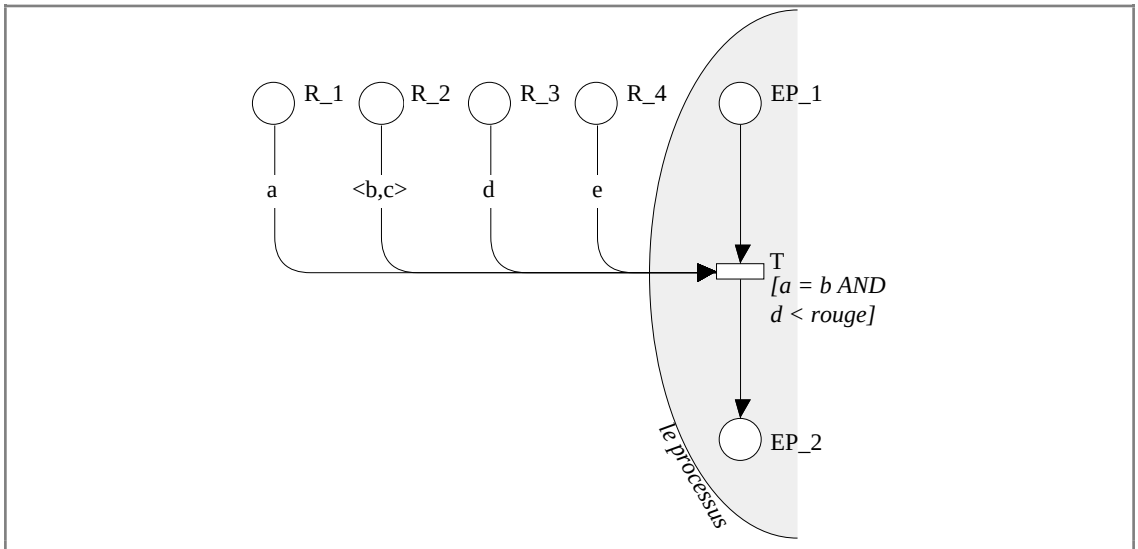


Figure IV.24 : Exemple de précondition ressource comportant une partie contrainte et une partie libre.

***Définition IV.32 :* Précondition contrainte**

La précondition contrainte contient l'ensemble des Ressources liées à une jointure ou une sélection.

***Définition IV.33 :* Précondition libre**

La précondition libre contient l'ensemble des Ressources auxquelles aucune jointure ou sélection n'est liée : il est possible de consommer des marques quelconques.

Exemple IV.6 : Considérons l'Action de la Figure IV.24. *EP_1* et *EP_2* sont des Etats_Processus, *R_1* , *R_2*, *R_3* et *R_4* des Ressources contenant des marques colorées.

Les Ressources *R_1*, *R_2* et *R_3* appartiennent à la précondition contrainte de *T*. *R_4* appartient à la précondition libre de *T*.

L'évaluation d'une précondition ressource suppose que l'on bloque momentanément l'accès de l'ensemble des Ressources qui la composent : il est impossible de consommer *a priori* le marquage, composante par composante. Cela entraînerait des interblocages.

En cas de succès de l'évaluation, le marquage des Ressources concernées est mis à jour. Dans le cas contraire, il faut attendre un événement positif (Définition IV.27) concernant la Ressource ayant provoqué l'échec.

Afin d'éviter les problèmes de famine, il faut estampiller les demandes. A chaque *événement positif*, la plus ancienne demande réveillée sera réévaluée.

3.2.2. Production d'une postcondition ressource

Contrairement à l'évaluation d'une précondition, la production d'un marquage postcondition n'est jamais bloquante. Elle est *positive* pour le système puisqu'elle provoque éventuellement le réveil de préconditions bloquées sur un défaut de marquage.

C'est également un service fractionnable : la production de marques dans chaque composante peut être effectuée séparément.

3.2.3. La notion de condition

Le module de gestion des Ressources offre deux services :

- evaluation d'une précondition;
- production d'un marquage postcondition.

La notion de *condition* permet d'uniformiser la description des préconditions et des postconditions.

Définition IV.34 : **Condition**

Une condition permet de représenter des relations entre des marques. Elle décrit indifféremment, selon son utilisation, une précondition ou une postcondition.
Une condition est décrite sous une forme arborescente. Les nœuds de l'arbre sont de trois types :

- Des nœuds-variables ou des nœuds-constantes : de tels nœuds sont forcément des feuilles. Ils référencent soit une marque (ou une composante d'une marque composée), soit des valeurs immédiates. Il existe des constantes particulières : "n'importe quelle valeur" (consommation d'une marque de profil quelconque) et "toutes les valeurs d'une classe" (diffusion).

- Des nœuds-modificateurs : ces nœuds représentent les fonctions successeur ou prédécesseur. Ils ont deux fils : à gauche, un nœud de type variable (la variable référençant la marque à laquelle s'applique le modificateur), à droite, une constante numérique (le rang du modificateur).
- Des nœuds-comparaison : ils représentent des fonctions logiques de comparaison. Ils ont deux fils qui sont, soit de type variable, soit de type modificateur.
- Des nœuds-logiques : ils représentent des fonctions booléennes ET, OU et NON. Les fonctions ET et OU sont d'arité quelconque; la fonction NON est unaire. Leurs fils sont, soit des nœuds-logiques, soit des nœuds-comparaison.

Selon que le client souhaite produire ou consommer des marques, la condition définie devient une postcondition ressource ou une précondition ressource. Leurs définitions respectives doivent cependant respecter quelques règles :

- **Construction d'une précondition ressource** : la précondition contrainte est, dans un premier temps, déduite des relations entre marques (ou composantes de marques) référencées dans le prédicat associé à l'Action. La précondition libre est ensuite construite, à partir des Ressources non référencées dans le prédicat. La précondition contrainte et la précondition libre, reliées par un nœud ET, décrivent la précondition ressource.
- **Construction d'une postcondition ressource** : l'arbre décrivant une postcondition se réduit à un nœud-opérateur ET dont tous les fils sont des arbres de la forme : <Ressource, = , *constante*>; *constante* définit une marque (composante par composante) qui sera produite dans la Ressource associée.

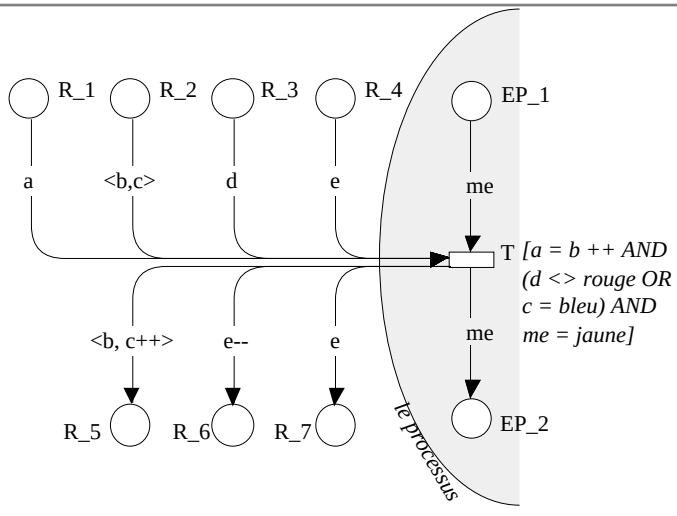


Figure IV.25 : Exemple d'Action ayant une précondition et une postcondition complexes.

Exemple IV.7 : Considérons la Figure IV.25. *EP_1* et *EP_2* sont des Etats_Processus, *R_1* , *R_2*, *R_3*, *R_4*, *R_5*, *R_6* et *R_7* des Ressources contenant des marques colorées.

La précondition ressource associée à *T* comporte :
- une précondition contrainte ,
- une précondition libre.

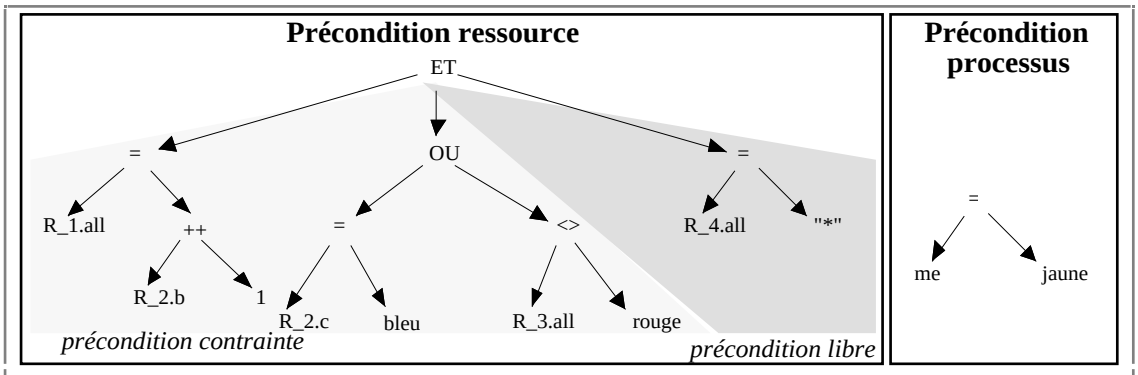


Figure IV.26 : Représentation de la précondition associée à la transition T de la Figure IV.25.

Nous donnons en Figure IV.26 l'arbre de description associé à la précondition de *T*. L'arbre de description de la précondition processus, évaluée par le module-processus concerné, est indiquée à droite. L'expression "Ressource.all" désigne une marque complète.

Nous donnons en Figure IV.27 l'arbre décrivant la postcondition associée à *T*. Toutes les variables présentes dans la postcondition doivent être référencées, soit dans le mot d'état du Processus, soit dans la précondition de l'Action. Les valeurs consommées à l'activation de la précondition seront produites dans les Ressources postcondition.

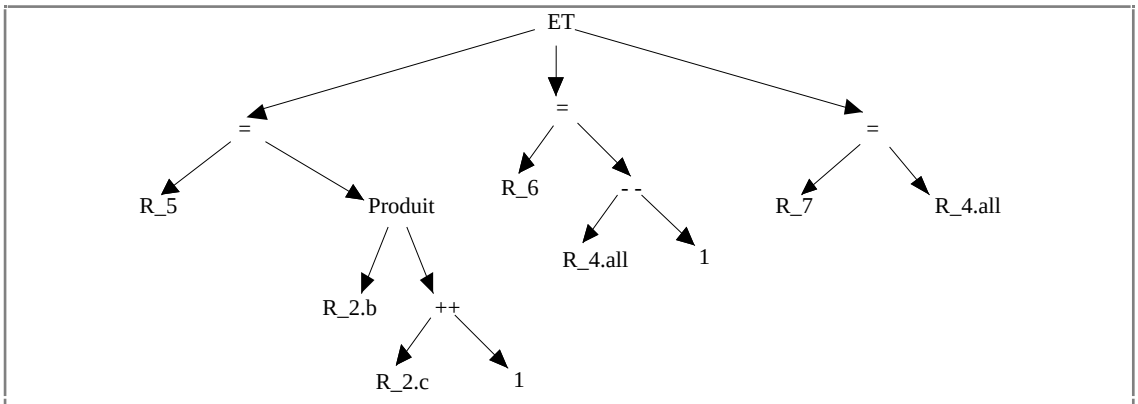


Figure IV.27 : Représentation de la postcondition associée à la transition T de la Figure IV.25.

3.2.4. Initialisation et terminaison

L'initialisation du module de gestion des Ressources est asservie au contrôle de l'application. A l'initialisation, il faut :

- mettre le mécanisme en état d'accepter des demandes de service,
- indiquer le marquage initial des Ressources.

La terminaison du module de gestion des Ressources est également asservie au contrôle de l'application. A ce stade, il faut détruire l'ensemble des structures ayant permis la réalisation des services offerts.

3.3. SERVICES REQUIS

Nous décrivons, dans cette section, l'ensemble des services requis par le module de gestion des Ressources. Ces services sont liés :

- au contrôle du prototype;
- à la gestion des composants externes;

- à la communication avec les clients (modules-processus ou gestionnaire des synchronisations).

Certains de ces services sont offerts par les modules externes, d'autres par les modules clients (Figure IV.28).

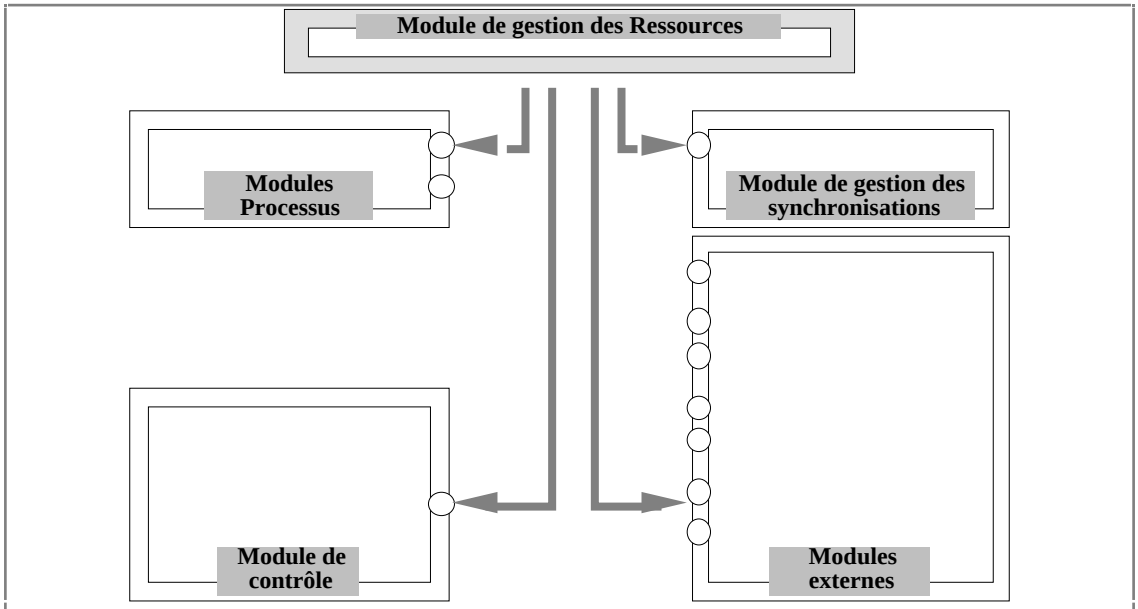


Figure IV.28 : Catégories de services requis par le module de gestion des Ressources.

3.3.1. Services requis au module de contrôle

En cas d'erreur survenant durant l'exécution d'une primitive associée à un composant, il est intéressant de disposer d'un service `PROBLEME_EXTERNE` permettant au gestionnaire de Ressources de propager l'événement au module de contrôle, permettant ainsi de diagnostiquer rapidement les erreurs de conception des composants.

3.3.2. Services requis aux modules externes

Les services que doit offrir chaque module externe sont de deux catégories :

- Les primitives correspondant aux interfaces avec les modules externes (qu'elles soient associées à un composant externe réversible ou non-réversible). Elles doivent respecter les contraintes définies dans la section 3.2;
- `EVENEMENT_POSITIF` : permettant de savoir si un événement positif a eu lieu pour la Ressource externe en sortie spécifiée en paramètre. *Cette primitive n'est requise que si la détection d'un événement positif est à la charge du module de gestion des Ressources.*

3.3.3. Services requis aux modules clients

Les services demandés aux clients permettent au module de gestion des Ressources de transmettre des informations durant les échanges qui ont lieu dans le cadre de l'évaluation d'une précondition ressource.

Certains services sont indépendants de la politique d'implémentation des Etats_Processus alternatifs. Ils concernent tous les types de clients (gestion des Actions synchronisées et modules-processus) :

- PRECONDITION_REALISEE : permettant de signaler au client que la précondition dont il a demandé l'évaluation est réalisée. Par ce service, le module de gestion des Ressources transmet les marques consommées (choix a priori d'une Action postcondition) ou potentiellement consommables (évaluation en parallèle des Actions postcondition).
- REPONSE_ANNULATION : permettant de signifier au client l'acceptation ou le refus de l'annulation d'une évaluation.

Si la politique d'évaluation simultanée des Actions postcondition est choisie pour la réalisation des Etats_Processus alternatifs, les modules-processus devront, en outre, fournir un service CONSOMMATION_EFFECTUEE, permettant au module de gestion des Ressources de signaler la consommation effective d'un marquage précondition pour l'Action élue.

3.4. SYNTHÈSE DES INTERFACES

Nous récapitulons l'ensemble des services requis et offerts par le module de gestion des Ressources (Figure IV.29).

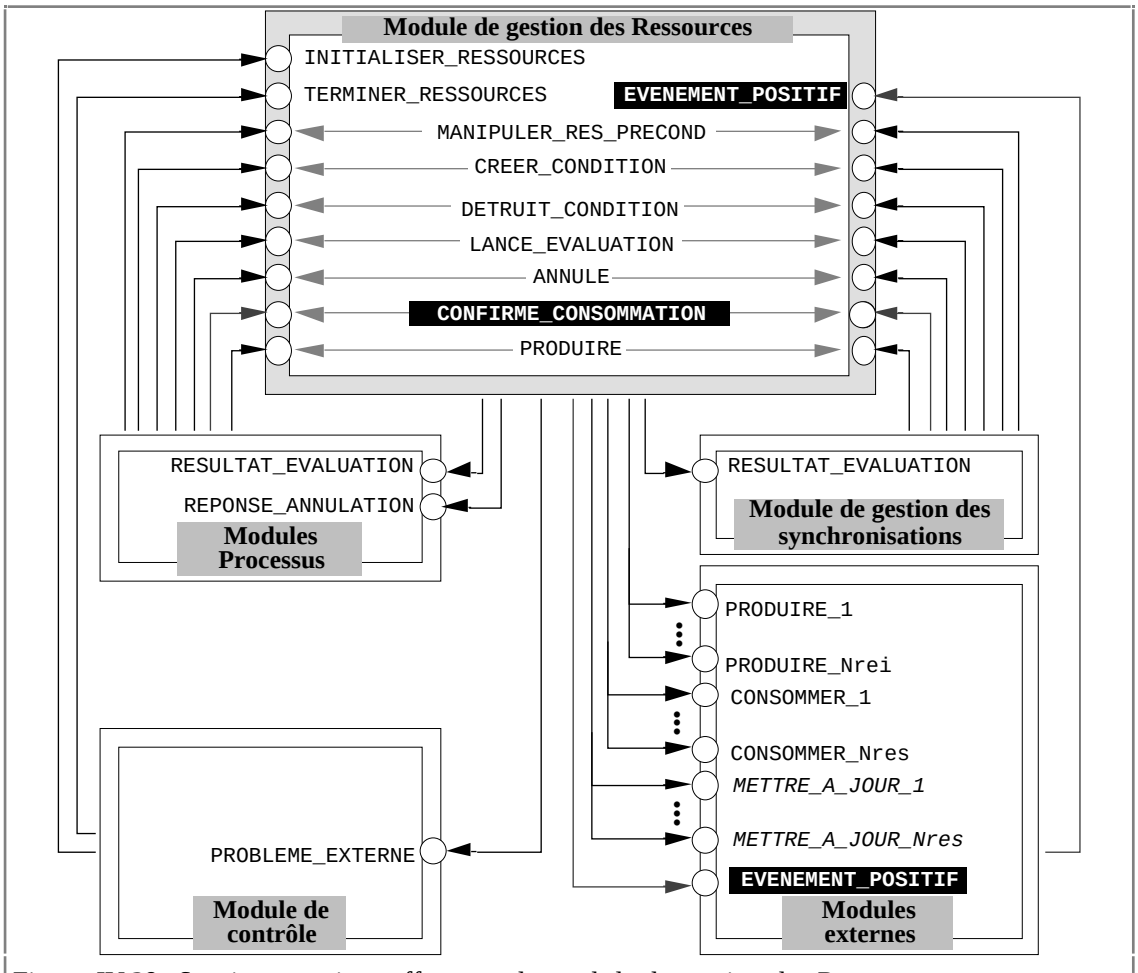


Figure IV.29 : Services requis et offerts par le module de gestion des Ressources.

Le service EFFECTUE_CONSOMMATION n'existe que si l'on implémente la politique d'évaluation en parallèle des Actions postcondition d'un Etat_Processus alternatif. Les modules-processus, comme le gestionnaire des synchronisations, utilisent, s'il existe, ce service afin de préserver l'homogénéité des relations client-serveur.

Le service EVENEMENT_POSITIF est offert, soit par le module de gestion des Ressources, soit par les composants externes, selon la stratégie d'implémentation choisie. L'existence de ce service est liée à celle de Ressources externes. Si la spécification n'en comporte pas, toute relation avec des modules externes est inexistante.

Les modules associés à des composants externes non-réversibles ne diffèrent de ceux qui sont associés à des composants externes réversibles que par la présence des services MISE_A_JOUR, permettant de restituer les marques non consommées dans une Ressource externe en sortie.

Le service REPONSE_ANNULATION n'est requis que pour les modules ayant au moins une Action simple gardée comportant une précondition ressource en postcondition d'un Etat_Processus alternatif.

Enfin, le service MANIPULER_RES_PRECOND permet aux clients de consulter le marquage consommé, après évaluation positive d'une précondition.

3.5. ARCHITECTURES POSSIBLES DU MODULE

Le marquage d'une Ressource constitue une donnée critique dont l'accès doit être protégé. Un tel accès est réalisé par des *serveurs*. Nous pouvons considérer plusieurs politiques :

- un seul serveur gère l'accès à l'ensemble des Ressources;
- il existe autant de serveurs que de Ressources : chaque serveur gère l'accès à une seule Ressource;
- il existe N serveurs pour M Ressources : chaque serveur gère un ensemble de Ressources.

Nous allons, dans cette section, étudier les implémentations possibles dans les trois cas de figure.

3.5.1. Un seul serveur pour l'ensemble des Ressources

Considérons dans un premier temps que les Ressources sont gérées par un seul serveur [Kordon 91a]. Un tel choix est parfaitement envisageable si toutes les tâches de l'application s'exécutent sur un même site.

Nous ramenons la relation entre modules-processus et module de gestion des Ressources à un modèle de type client-serveur (Figure IV.30). Les clients effectuent des demandes de service (évaluation d'une précondition ou production d'un marquage postcondition). Le serveur, lorsque le service est traité, réveille les clients. En cas d'accès à une Ressource externe, il effectue l'appel de la primitive correspondante.

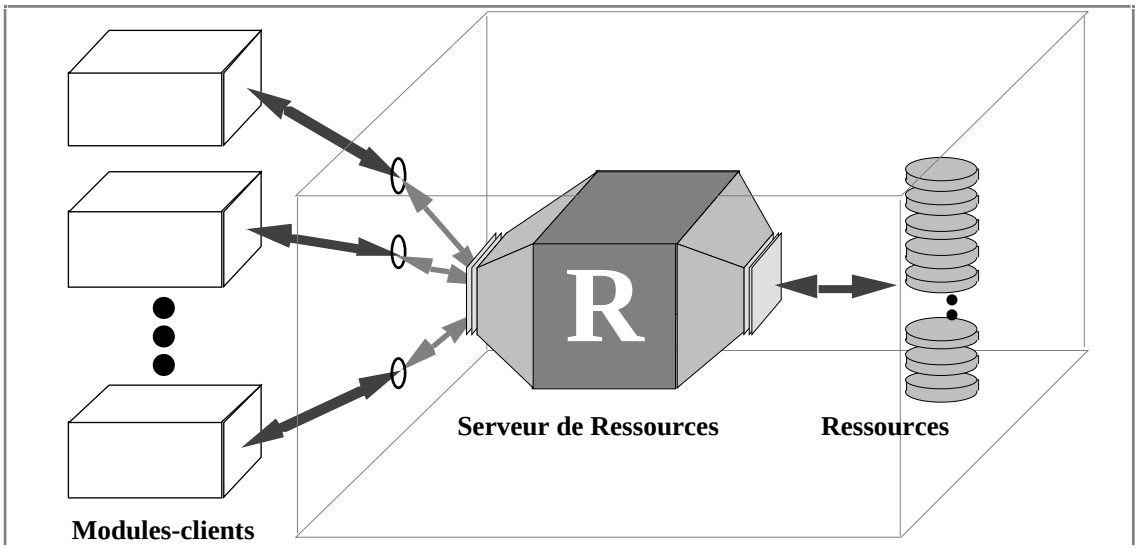


Figure IV.30 : Gestion des Ressources effectuée par un seul serveur.

Le traitement des requêtes des clients s'en trouve grandement simplifié puisque l'accès aux Ressources est contrôlé par une seule tâche. En revanche, le parallélisme n'est pas optimal car des demandes concernant différentes Ressources sont traitées, les unes après les autres, par le serveur : cela constitue un goulot d'étranglement.

3.5.2. Un serveur par Ressource

Accorder à chaque serveur la gestion d'une seule Ressource revient à distribuer complètement la gestion des Ressources de façon similaire à [Hauschildt 87, Taubner 87, Bréant 90]. Les serveurs deviennent des sémaphores chargés de protéger des sections critiques (Figure IV.31). Les sémaphores associés aux Ressources externes effectuent directement l'appel de la primitive associée.

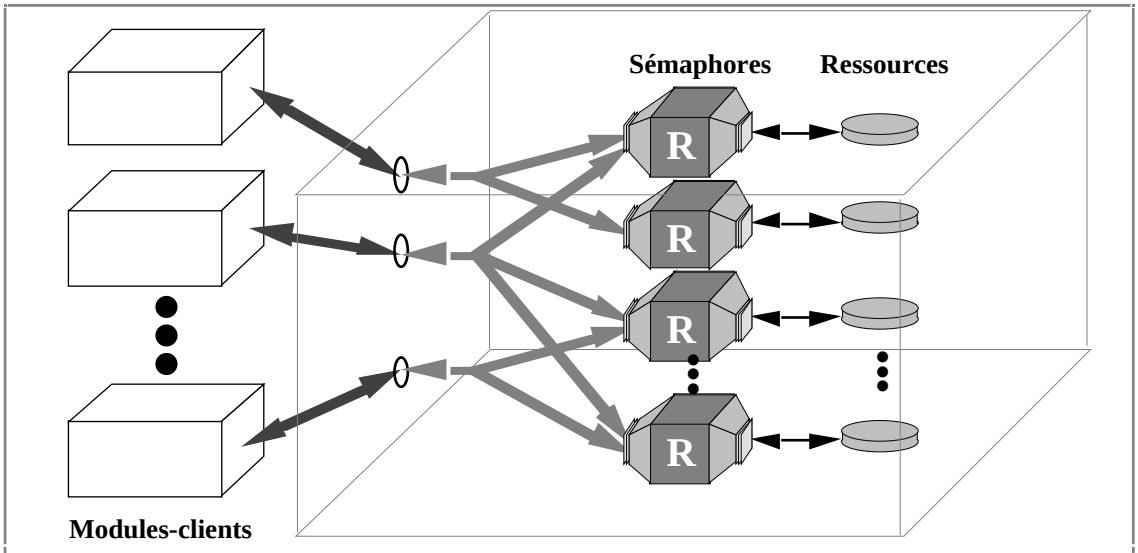


Figure IV.31 : Chaque Ressource est gérée par un serveur. Les serveurs se ramènent à des sémaphores.

Le principal avantage de cette architecture est de maximiser le parallélisme : deux demandes n'ayant aucune Ressource en commun sont traitées en parallèle.

La production d'un marquage postcondition ne pose pas de problème puisqu'elle peut être effectuée composante par composante. En revanche, l'évaluation en parallèle de plusieurs préconditions ayant des Ressources en commun peut induire des interblocages. Des techniques de détection (avec relance des requêtes) ou de prévention (ordonnancement des Ressources) doivent être intégrées.

Une telle architecture est envisageable pour des architectures mono-processeur ou parallèles fortement couplées. Dans le cas d'une architecture de type multi-processeurs, le placement des Ressources, et donc des sémaphores qui leur sont associés, devient un problème délicat : un mauvais placement aura un impact important sur les performances.

3.5.3. Un serveur pour plusieurs Ressources

Une telle architecture peut être proposée comme un "moyen terme" aux deux précédentes. L'idée est de regrouper la gestion de plusieurs Ressources sous la responsabilité d'un seul serveur.

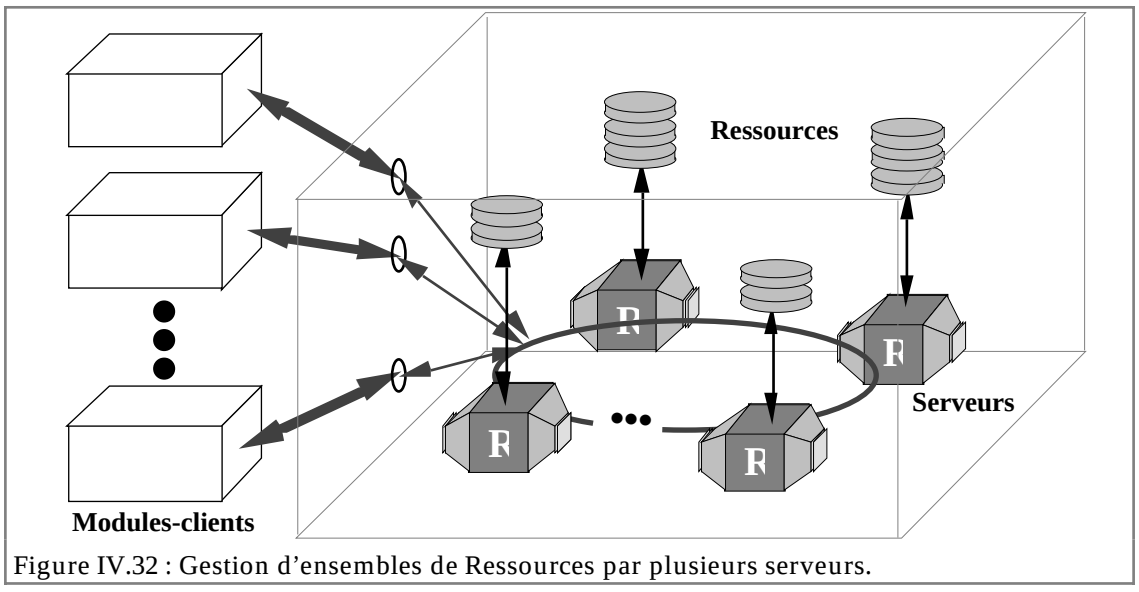


Figure IV.32 : Gestion d'ensembles de Ressources par plusieurs serveurs.

Nous obtenons ainsi l'organisation donnée en Figure IV.32. La configuration des liens de communication entre les serveurs peut être :

- un anneau : chaque serveur ne connaît que ses voisins immédiats,
- une clique : chaque serveur connaît l'ensemble de ses homologues.

Une telle structure permet de préserver un certain degré de parallélisme, tout en minimisant les conflits d'accès en les localisant au niveau des serveurs.

Cependant, des problèmes liés à la gestion de l'état global des Ressources, similaires à ceux envisagés dans les bases de données réparties, doivent être considérés [Bernstein 78, Bernstein 81, Raynal 91]. Différentes solutions sont

envisageables [Le Lann 77, Lamport 78, Bernstein 80, Raynal 84, Birman 85, Garcia-Molina 85, Raynal 87, Birman 89].

La gestion des Ressources externes pose un autre problème. Un protocole de dialogue particulier, sur la base des restrictions effectuées sur leur utilisation, doit être établi.

Une telle configuration peut se concevoir dans le cadre d'une architecture parallèle faiblement couplée. Chaque serveur est alors situé sur un site différent.

Un protocole de dialogue entre les serveurs doit être élaboré pour prévenir :

- l'incohérence dans le marquage des Ressources,
- l'interblocage ou la famine dans le traitement des demandes.

Cette architecture est étudiée de manière plus détaillée dans le Chapitre VII, section 4.

4. Le module de gestion des Actions synchronisées

Dans cette section, nous détaillons le module de gestion des Actions synchronisées. Après avoir décrit les problèmes liés à l'activation d'une Action synchronisée, nous présentons les services offerts et requis par ce module. Nous discutons ensuite de son architecture.

Le module de gestion des Actions synchronisées est chargé des communications synchrones dans le prototype. Il offre des services pour le compte des modules-processus qui doivent réaliser un rendez-vous avec un ou plusieurs autres Processus.

La Figure IV.33 représente les relations entre le module de gestion des Actions synchronisées et les autres modules composant le prototype.

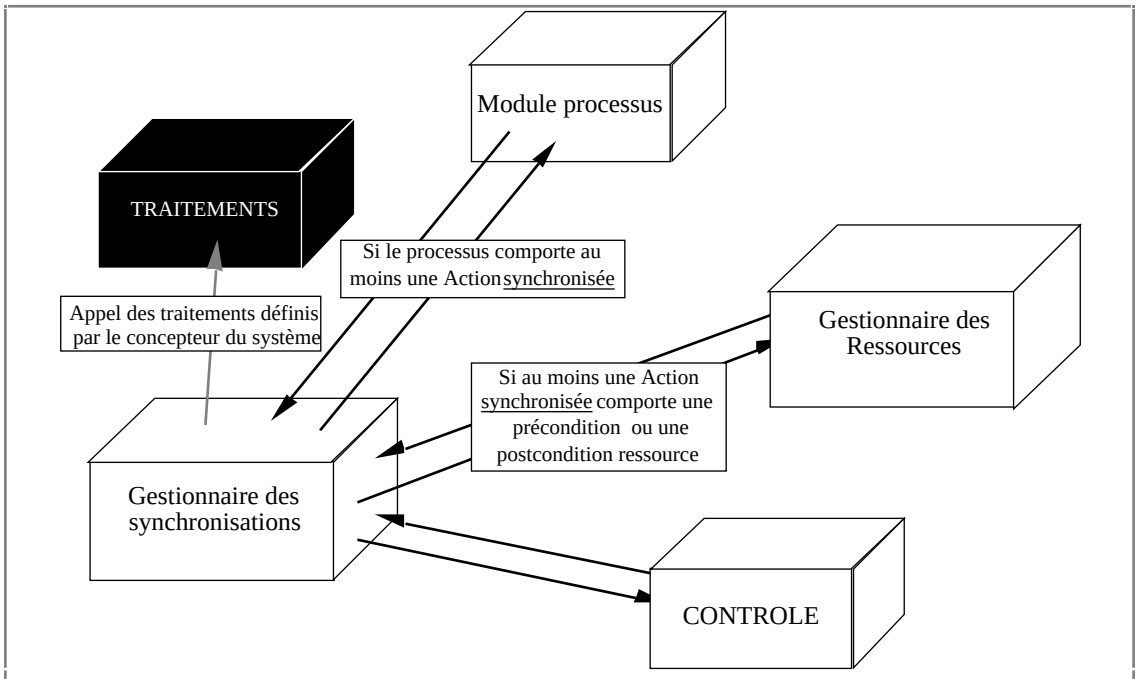


Figure IV.33 : Relation entre le module de gestion des Actions synchronisées et les autres modules du prototype.

L'initialisation et la terminaison des communications synchrones sont asservies au module de contrôle du prototype.

Il existe un lien avec le module de gestion des Ressources s'il existe au moins une Action synchronisée ayant une précondition ou une postcondition ressource.

Aucun service n'est requis par le module définissant les traitements associés aux Actions puisqu'aucun dialogue n'a lieu. Ses relations avec le module de gestion des Actions synchronisées sont unidirectionnelles.

4.1. PROBLÈMES LIÉS À L'ACTIVATION D'UNE ACTION SYNCHRONISÉE

La réalisation d'une synchronisation est soumise :

- à la présence des divers participants,
- éventuellement, au contenu du mot d'état de chacun des participants (Actions synchronisées gardées comportant une précondition processus),
- éventuellement, au contenu des Ressources précondition (Actions synchronisées gardées comportant une précondition ressource).

L'évaluation de la précondition ressource est réalisée par le module de gestion des Ressources, la précondition processus est à la charge du module de gestion des Actions synchronisées.

La précondition processus d'une Action synchronisée regroupe l'ensemble des Etats_Processus précondition de l'Action ainsi que les contraintes qui leur sont associées. Certaines contraintes n'impliquent qu'un seul participant, d'autres plusieurs. Nous définissons de la sorte la précondition statique et la précondition synchrone.

Définition IV.35 : **Précondition statique**

Soient a ASY, pr Pr $\cap$ Π_p (a). Nous définissons Π_t (a, pr) la précondition statique de l'Action synchronisée a pour le participant pr :

- Π_t (a, pr) = (χ_t (a, pr), {p | p E $\cap$ pr et un arc reliant p à a}) où χ_t (a, pr) définit le prédicat associé à la transition concernant les places p | p E $\cap$ pr.

χ_t (a, pr) ne doit pas faire référence à des marques (ou composantes de marques) contenues dans d'autres Etats_Processus que p ou dans des Ressources.
Une précondition statique sans aucune contrainte sur les Etats_Processus en entrée est dite *dégénérée*.

Pour les Actions simples, la précondition processus se réduit toujours à une précondition statique. Corollairement à ce que nous avons énoncé en section 2.1.1, les Actions synchronisées et les préconditions statiques, associées à un processus postcondition d'un Etat_Processus simple, doivent être dégénérées (Définition IV.12).

Définition IV.36 : **Précondition synchrone**

Soient a ASY, pr Pr $\cap$ Π_p (a). Nous définissons Π_s (a) la précondition statique de l'Action synchronisée a pour le participant p :

- Π_s (a) = (χ_s (a), $_{i=1..|\Pi p|}$ {p_i} | p_i E $\cap$ pr et un arc reliant p à a)

où χ_s (a) définit le prédicat associé à la transition concernant les places p | p E $\cap$ pr avec :

- p_i Π_p (a) et pr_j | p_i ε (pr_j), χ_s (a) $\cap$ χ_t (a, pr_j) = .

Exemple IV.8 : Considérons le modèle de la Figure IV.34. Une synchronisation entre deux Processus, définis respectivement par les Etats_Processus *P1a, P1b, P1fin* et *P2a, P2b, P2fin* y est décrite (Action *Sync*). Le mot d'état de chaque processus comporte une variable (*t1* ou *t2*) qui ne peut prendre que deux valeurs 1 et 2 (1 est le successeur de 2 et 2 le successeur de 1). L'Action synchronisée ne peut avoir lieu que pour des instances de processus ayant même identité (*num1* = *num2*) et qui peuvent tenter de se synchroniser (*t1* et *t2* valent 1). Les marques consommées depuis les Ressources *R_1* et

R_2 ne doivent pas être égales à l'identité des instances de processus. Au bout d'un certain nombre de synchronisations, chaque instance de processus peut se terminer.
Sync comporte :

- une précondition ressource : {"marque provenant de *R_1* différente de la valeur contenue dans *num1* **ET** marque provenant de *R_2* différente de la valeur contenue dans *num2*", {R_1, R_2}};
- deux préconditions statiques (associées à chacun des participants) :
  pour le processus de gauche : {"la composante *t1* des marques contenues dans *P1a* vaut 1", {P1a}};
  pour le processus de droite : {"la composante *t2* des marques contenues dans *P2a* vaut 1", {P2a}};
- une précondition synchrone : {"la composante *num1* de la marque provenant de *P1a vaut* la composante *num2* de la marque provenant de *P2a*", {P1a, P2a}}.

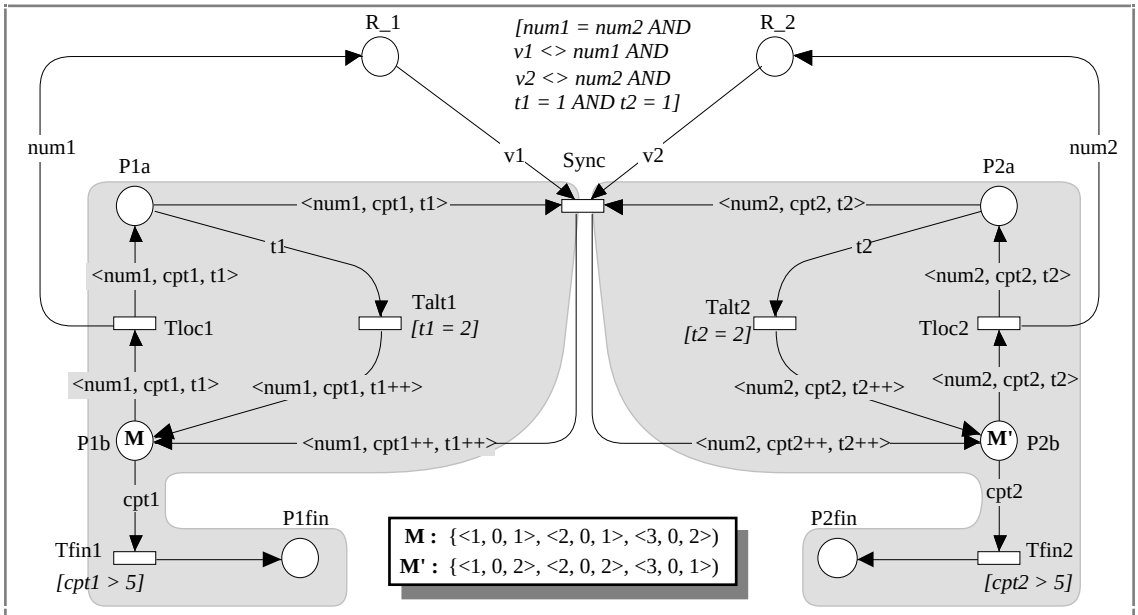


Figure IV.34 : Exemple de modèle avec une Action synchronisée comportant une précondition ressource, une précondition statique et une précondition synchrone.

Le module de gestion des Actions synchronisées effectue des demandes au gestionnaire des Ressources pour évaluer la précondition ressource ou générer la postcondition ressource.

Comme dans le cas des processus, la production d'une postcondition ressource ne pose aucun problème : le service n'est pas bloquant.

L'évaluation d'une précondition ressource est plus délicate : le gestionnaire des synchronisations, même s'il est en attente d'une réponse, doit pouvoir traiter d'autres demandes. Le protocole déjà défini pour la relation entre le gestionnaire des Ressources et les modules-processus est utilisé.

Le traitement associé à une Action est effectué par le module de gestion des Actions synchronisées. Ce traitement ne doit être effectué qu'une seule fois, pour le compte de l'ensemble des clients qui participent à sa réalisation.

A chaque Action synchronisée correspond un *contexte* qui évolue au fur et à mesure des demandes de connexion, des demandes de retrait, et de la réalisation des Actions. Le module de gestion des Actions synchronisées doit

évaluer régulièrement ces contextes afin d'extraire des *instances* de synchronisations.

Définition IV.37 : **Instance d'une Action synchronisée**

Une instance d'une Action synchronisée a ASY est une synchronisation effective entre des participants ayant émis le désir de réaliser l'Action. L'instance est décrite par le couple (λ (a), μ (a)) où :

- λ (a) représente l'ensemble des instances des processus participants vérifiant la précondition synchrone,
- μ (a) représente les marques consommées dans la précondition ressource.

Les instances sont déduites du contexte de l'Action synchronisée. A chaque évolution de ce contexte, il y a tentative de construction d'une instance.

Dans un premier temps, les préconditions statiques sont évaluées. S'il existe au moins une instance sélectionnée pour l'ensemble des participants, il y a évaluation de la précondition synchrone. S'il existe toujours au moins une instance par participant, une requête est transmise au gestionnaire des Ressources pour évaluer la précondition ressource.

Exemple IV.9 : Considérons le modèle de la Figure IV.34. Nous notons *P1* le processus "de gauche" (défini par *P1a*, *P1b* et *P1fin*) et *P2* le processus "de droite" (défini par *P2a*, *P2b* et *P2fin*).
Considérons les événements présentés dans le tableau ci-dessous :

précondition de Sync	**événement**	**R_1**	**R_2**	**Contexte lié à Sync**	
				P1	P2
non vérifiée	+ P1 (<1,0,1>)	1		<1,0,1>	
non vérifiée	+ P1 (<2,0,1>)	1,2		<1,0,1>, <2,0,1>	
non vérifiée	+ P2 (<3,0,1>)	1,2	3	<1,0,1>, <2,0,1>	<3,0,1>
non vérifiée	- P1 (<1,0,1>)	1,2	3	<2,0,1>	<3,0,1>
non vérifiée	+ P2 (<2,0,1>)	1,2	2,3	<2,0,1>	<3,0,1> <1,0,1>
vérifiée	+ P1 (<3,0,1>)	1,2,3	2,3	<2,0,1>, <3,0,1>	<3,0,1>, <1,0,1>
Instance		**1**	**2**	**<3,0,1>**	**<3,0,1>**

Les deux premiers événements enrichissent le contexte de Sync mais aucune instance n'est réalisable car Π_t (Sync, P2) n'est pas vérifiée.
Le troisième événement entraîne la réalisation de Π_t (Sync, P2). Cependant, Πs (Sync) n'est pas vérifiée.
Le retrait de l'instance <1,0,1>) de P1 modifie le contexte mais aucune instance ne devient réalisable.
Le cinquième événement provoque toujours l'échec de Π_s (Sync). Ce n'est pas le cas du dernier événement mentionné dans le tableau. Comme Π_r (Sync) est également vérifiée, une instance de l'Action *Sync* peut être activée pour le compte de P1 (<3,0,1>) et P2 (<3,0,1>) avec les marques 1 et 2 prises respectivement dans *R_1* et *R_2*.

4.2. SERVICES OFFERTS

Dans cette section, nous détaillons les services offerts par le module de gestion des Actions synchronisées (Figure IV.35). Ils sont destinés :

- aux modules-processus, pendant le fonctionnement du prototype,

- au module de contrôle, durant l'initialisation et la terminaison du prototype.

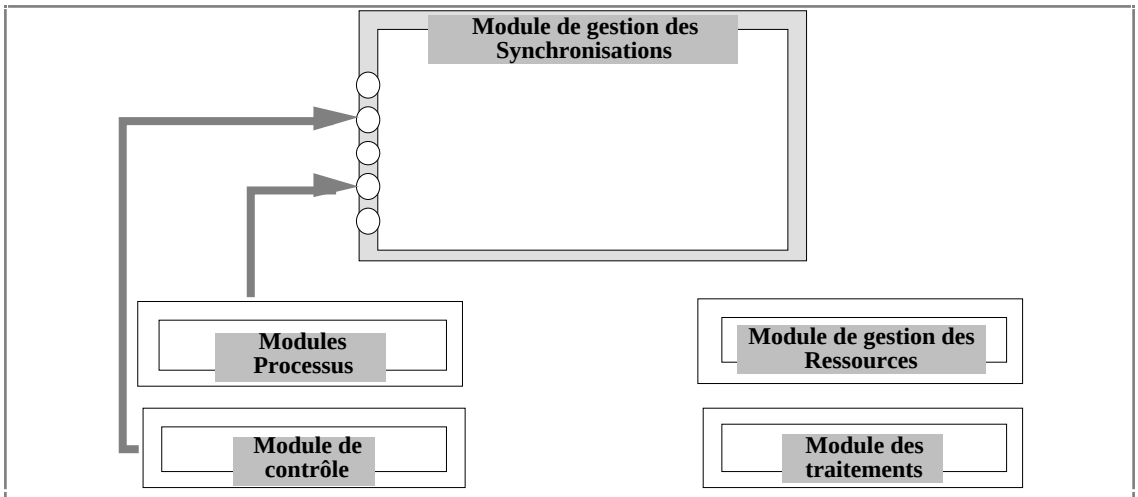


Figure IV.35 : Les catégories de services offerts par le module de gestion des Synchronisations.

Aucun service n'est offert au module de gestion des Ressources ni au module des traitements.

4.2.1. Services offerts aux modules-processus

Le module de gestion des Synchronisations doit offrir les services suivants :

- `JE_ME_CONNECTE` : il permet à un client de signaler qu'il est candidat à une synchronisation. Ce service correspond, côté gestion des Actions synchronisées, au service `LANCE_EVALUATION` requis par les modules-processus.
- `JE_ME_RETIRE` : il permet à un client de se retirer d'une synchronisation. L'utilisation de ce service n'est possible que si, pour le client considéré, l'Action synchronisée est postcondition d'un Etat_Processus alternatif. Ce service correspond au `ANNULE` requis par les modules-processus.

A ces services, dont l'existence est indépendante du choix d'implémentation des Etats_Processus alternatifs, s'ajoute, si l'on choisit la politique d'évaluation en parallèle des Actions postcondition :

- `CONFIRMER_SYNCHRONISATION` : il permet à un client de confirmer le lancement de l'Action synchronisée, après évaluation positive de sa précondition. Ce service correspond, côté gestion des Actions synchronisées, au service `CONFIRME` requis par les modules-processus.

4.2.2. Services offerts au module de contrôle

L'initialisation et la terminaison de la gestion des Actions synchronisées sont asservies au mécanisme de contrôle de l'application. Le gestionnaire des Synchronisations doit donc disposer de deux services :

- `INITIALISE_ASY` : par lequel le mécanisme de contrôle rend le module opérationnel;
- `TERMINE_ASY` : par lequel le mécanisme de contrôle provoque la terminaison des fonctionnalités du module.

4.3. Services requis

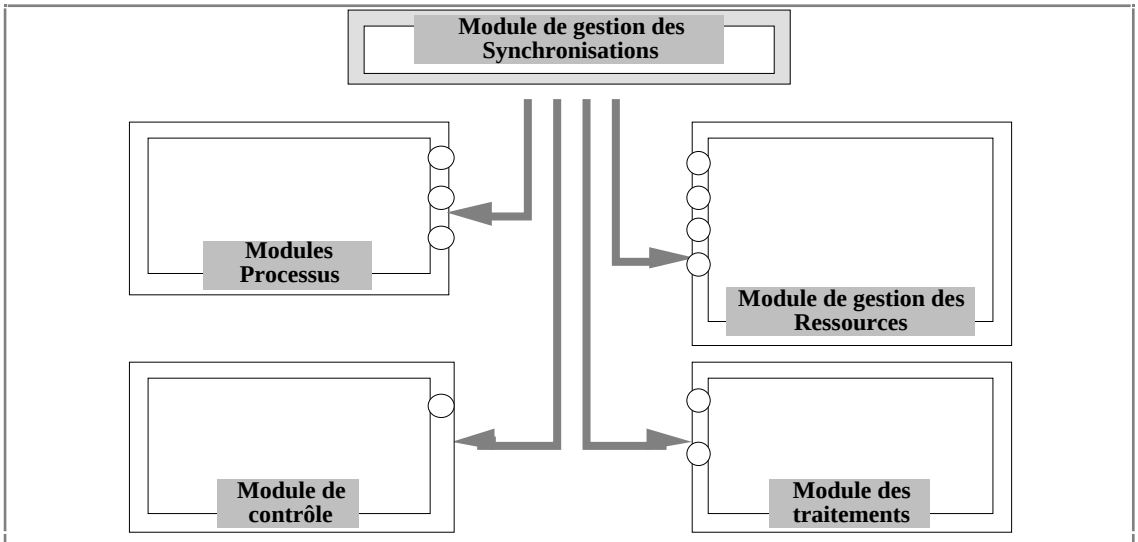


Figure IV.36 : Catégories de services requis par le module de gestion des synchronisations.

Dans cette section, nous détaillons les services requis par le module de gestion des synchronisations (Figure IV.36). Ces services doivent être offerts par :

- les modules-processus;
- le module de gestion des Ressources, s'il existe au moins une Action synchronisée ayant une précondition ressource;
- le module définissant les traitements associés aux transitions du modèle;
- le module de contrôle.

4.3.1. Services requis aux modules-processus

Certains services sont indépendants de la politique d'implémentation des Etats_Processus alternatifs :

- `SYNCHRONISATION_REALISEE` : permettant de signaler au client que l'Action synchronisée qu'il a tenté d'activer est réalisée. Par ce service, le gestionnaire des synchronisations transmet le mot d'état, éventuellement modifié par la postcondition processus.

  Ce résultat est effectif (choix a priori d'une Action postcondition) ou potentiel (évaluation en parallèle des Actions postcondition).
- `REPONSE_ANNULATION` : permettant de signifier au client l'acceptation ou le refus de l'annulation d'une évaluation.

L'évaluation des préconditions statiques et synchrones et la construction de la postcondition impliquent une manipulation des données contenues dans le mot d'état des participants à la synchronisation. Des services, regroupés sous la dénomination `MANIPULER_MOT_D_ETAT`, doivent être fournis par les modules-processus impliqués dans la synchronisation.

4.3.2. Services requis au module de gestion des Ressources

Les services requis pour le dialogue entre le module de gestion des Ressources et le gestionnaire des Actions synchronisées sont les suivants :

- EVALUE_PRECONDITION_RESSOURCE : il permet l'évaluation de la précondition ressource associée à une Action synchronisée.
- ANNULE_EVALUATION : il permet au module de gestion des Actions synchronisées d'annuler une demande d'évaluation d'une précondition ressource. Cela arrive si le client ayant effectué une connexion souhaite se retirer.
- PRODUIRE_POSTCONDITION : il permet de produire le marquage postcondition, lorsqu'une Action synchronisée est activée.

La génération de la postcondition ressource et la modification du mot d'état de chaque participant peuvent entraîner la manipulation du marquage consommé pour réaliser la précondition. Des primitives de manipulation (MANIPULER_RES_CONSO) du marquage issu de la réalisation d'une précondition doivent être fournies par le gestionnaire de Ressources.

4.3.3. Services requis au module des traitements

Le module des traitements doit fournir l'ensemble des primitives définissant les traitements associés aux transitions du réseau de Petri. Le service correspondant est utilisé lorsqu'une Action synchronisée est activée.

4.3.4. Services requis au module de contrôle

En cas d'erreur survenant durant l'exécution d'une primitive du module de traitement, il est intéressant de disposer d'un service PROBLEME_EXTERNE permettant au gestionnaire des synchronisations de propager l'événement au module de contrôle, permettant ainsi de diagnostiquer rapidement les erreurs de conception des composants externes.

4.4. SYNTHÈSE DES INTERFACES

La Figure IV.37 récapitule les services requis et offerts par le module de gestion des synchronisations.

Le service RESULTAT_ANNULATION n'est requis que pour les processus ayant au moins une Action synchronisée postcondition d'un Etat_Processus alternatif.

Les services CONFIRMER_SYNCHRONISATION (offert) et CONFIRME_CONSOMMATION (requis) n'existent que si la politique d'évaluation en parallèle des Actions postcondition d'un Etat_Processus alternatif a été choisie.

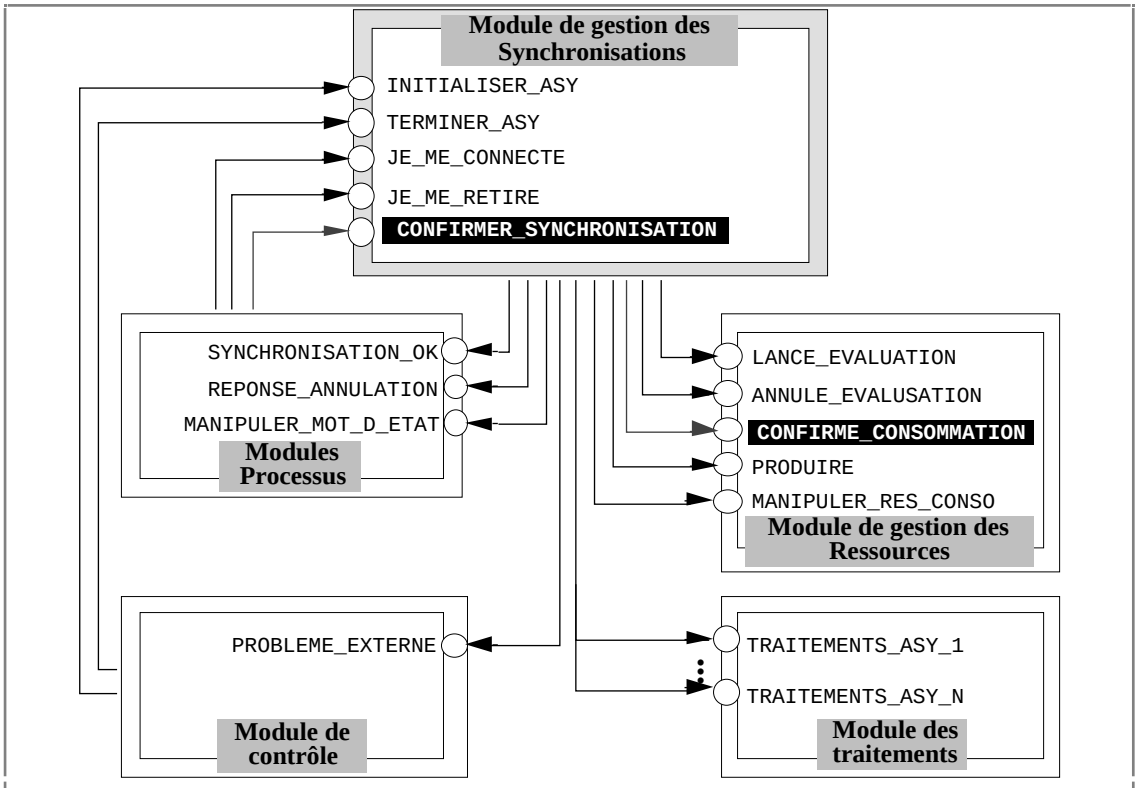


Figure IV.37 : Services requis et offerts par le gestionnaire des Actions synchronisées.

4.5. ARCHITECTURE DU MODULE

Le contexte d'une Action synchronisée constitue une donnée critique dont l'accès doit être protégé. Un tel accès est réalisé par des *serveurs*. Nous pouvons considérer plusieurs politiques :

- un seul serveur gère l'ensemble des Actions synchronisées;
- il existe un serveur par Action synchronisée.

Considérons dans un premier temps que l'ensemble des Actions synchronisées est géré par un seul serveur (Figure IV.38). Comme pour la gestion des Ressources, la relation entre modules-processus et gestionnaire des synchronisations respecte les règles du modèle client-serveur.

L'accès aux contextes des Actions synchronisées est contrôlé par une seule tâche. Le serveur constitue un goulot d'étranglement. Le parallélisme n'est pas optimal car les demandes ne peuvent être traitées simultanément alors qu'il n'y a jamais d'interaction entre les contextes.

L'introduction du goulot d'étranglement, ainsi que la séquentialisation du traitement des services ne sont pas nécessaires puisqu'elles n'apportent pas d'avantages conséquents. Il faut donc écarter une telle architecture.

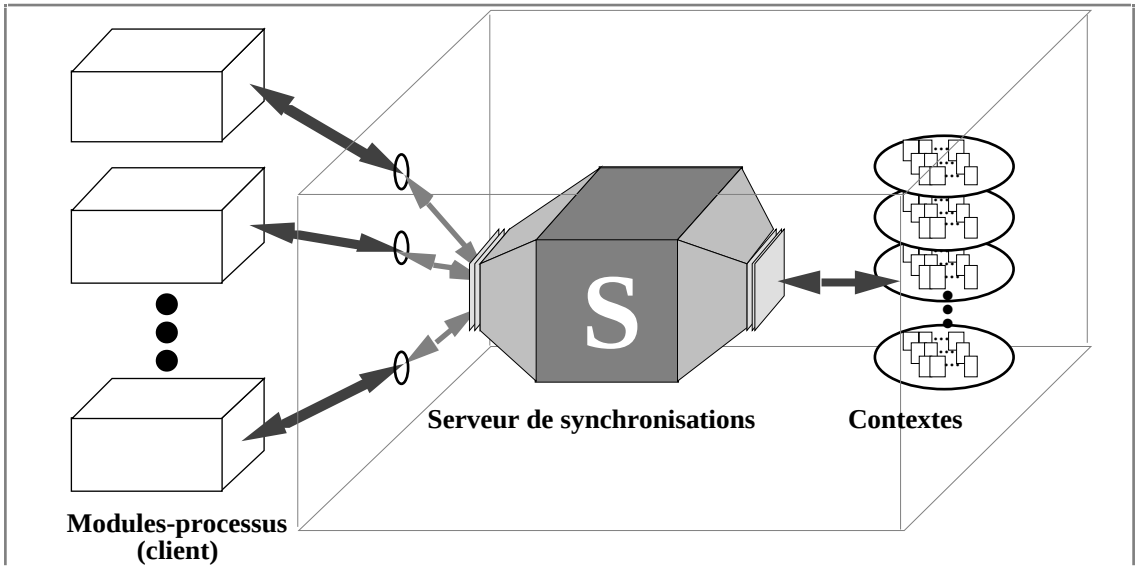


Figure IV.38 : Gestion des contextes des Actions synchronisées effectuée par un seul serveur.

Considérons désormais que chaque Action synchronisée est gérée par un serveur dédié (Figure IV.39). Chaque participant à une Action synchronisée se connecte au serveur concerné. Le parallélisme dans la gestion du contexte d'exécution associé à chaque Action synchronisée est assuré.

Chaque serveur enregistre les demandes des clients qui désirent se connecter (éventuellement, les retraits), met à jour un contexte local puis essaye d'en extraire une instance.

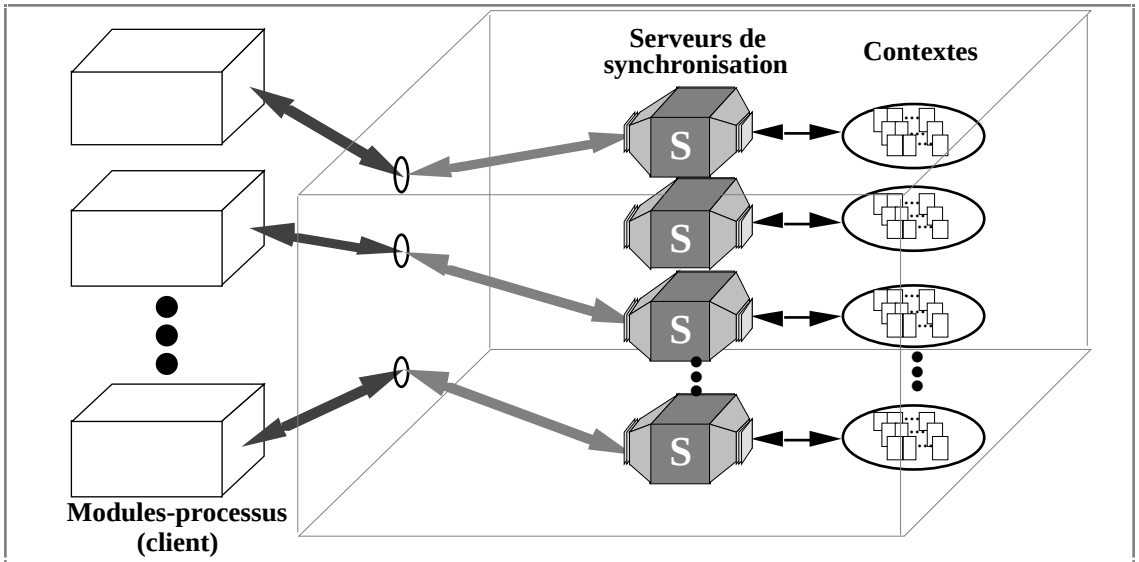


Figure IV.39 : Le contexte de chaque Action synchronisée est géré par un serveur dédié.

Deux étapes, dans le déroulement d'une Action synchronisée, sont potentiellement bloquantes :

- l'évaluation de la précondition ressource doit être effectuée par le module de gestion des Ressources;
- l'appel de la procédure associée à l'Action, ainsi que la production du marquage postcondition, sont réalisés par le serveur, pour le compte de l'ensemble des participants à la synchronisation.

Dans les deux cas, ce service étant bloquant, le serveur d'une Action synchronisée est incapable de traiter une autre demande jusqu'à expiration de la temporisation qu'il s'est donnée. Il faut créer dynamiquement une tâche chargée de l'évaluation de la précondition ressource, puis de la réalisation de l'Action. En cas de retrait d'un participant, et si la précondition ressource n'est pas encore réalisée, cette tâche de service est détruite. Lorsque la précondition ressource est réalisée, et l'Action activée, cette tâche de service prévient les clients concernés puis termine son exécution.

5. Le module de contrôle

Le contrôle d'une application parallèle est fréquemment réparti sur l'ensemble des processus composant l'application [Raynal 87, Mullender 89].

Le fait qu'il y ait ici des tâches actives (les instances des Processus du modèle) et des tâches passives (les tâches des modules de service) rend la répartition de ce contrôle plus délicate.

Nous avons choisi d'attribuer le contrôle du déroulement du prototype à un module dédie : le module de contrôle. Ses relations avec les autres modules composant le prototype sont données en Figure IV.40.

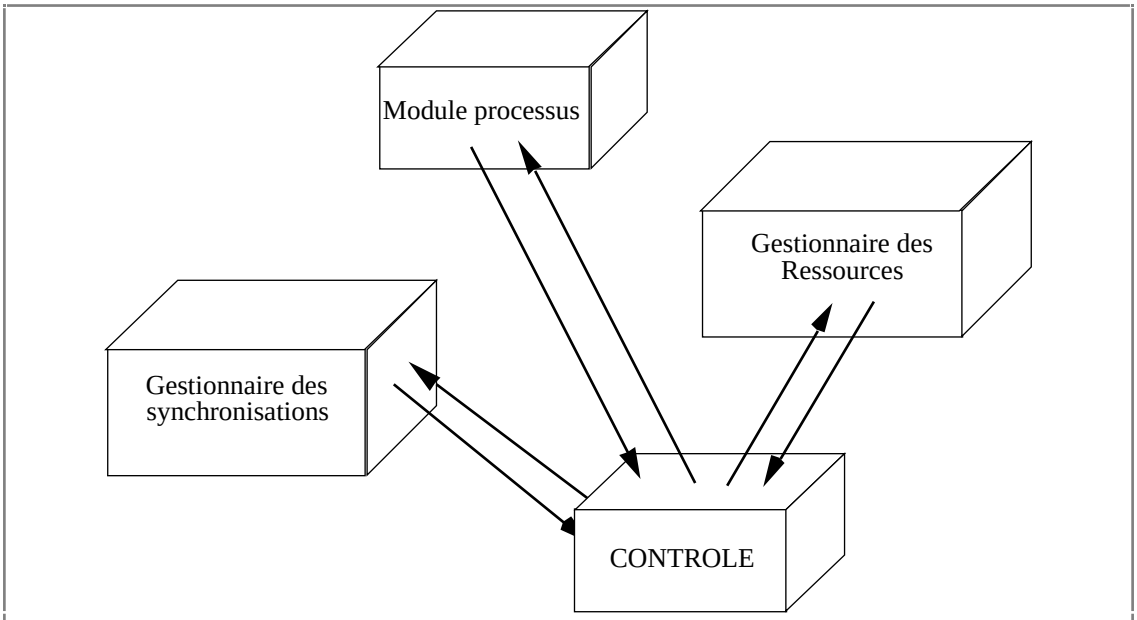


Figure IV.40 : Relations du module de contrôle avec les autres modules.

Le module des traitements et les modules externes n'ont aucune relation avec la gestion du contrôle car ils ne nécessitent aucune opération de contrôle (traitements associés à l'initialisation et la terminaison du prototype) :

- le module des traitements n'est pas concerné par de telles opérations,
- l'initialisation et la terminaison des modules externes doivent être explicitement spécifiées à l'aide de services, modélisés par des places d'interface.

Cette section décrit les services qu'il faut définir, ainsi que les architectures possibles dans la réalisation du module.

5.1. SERVICES OFFERTS

Les services offerts par le module de contrôle permettent la mise à jour du contexte global de l'application. Ces services sont offerts aux modules-processus et aux modules de service (Figure IV.41).

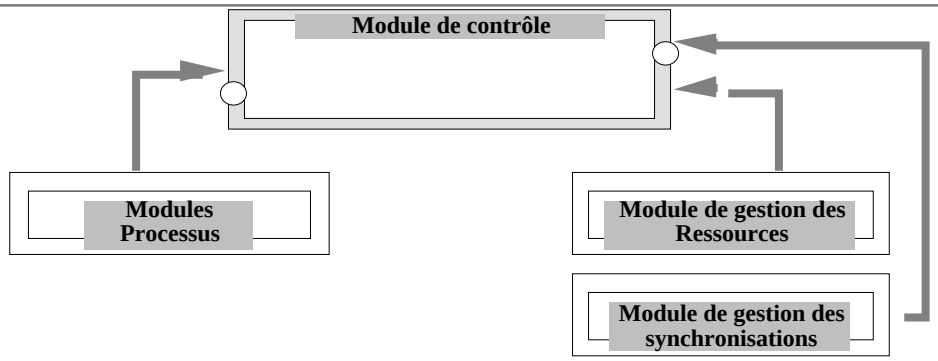


Figure IV.41 : Catégories de services offerts par le module de contrôle.

Les problèmes liés au contrôle du prototype sont les suivants :

- L'initialisation de l'application. Ce processus se déroule en deux étapes :
  - l'initialisation physique : il s'agit de construire l'application sur une architecture donnée (répartition des tâches, établissement des liens de communication...); cette étape n'a de sens que si le code généré est réparti sur un ensemble de processeurs;
  - l'initialisation logique : il s'agit de rendre opérationnel l'ensemble des modules composant le prototype.
- La terminaison de l'application qui intervient dans deux cas :
  - lorsque toutes les instances des Processus du modèle ont achevé leur exécution;
  - en cas d'erreur : si le générateur de code est certifié, elle ne peut provenir que des programmes fournis par le concepteur du système (procédures associées aux transitions et primitives liées aux composants externes).

5.1.1. Initialisation

L'initialisation logique de l'application s'effectue à partir des *constantes de démarrage* déduites de l'analyse du modèle ou acquise au cour de l'*enrichissement* effectué durant les différentes phases du prototypage.

Définition IV.38 : **Constantes de démarrage**

Les constantes de démarrage du prototype sont constituées de l'ensemble des données permettant de décrire :

- Les instances des Processus du modèle (nombre, Etat_processus initial, mot d'état initial) ainsi que le marquage initial des Ressources. Ces informations sont déduites du marquage initial du réseau de Petri.
- Diverses données correspondant aux contraintes spécifiées par le concepteur du modèle. Il s'agit d'informations supplémentaires, liées au langage cible ou à l'architecture cible, gérées par l'outil de prototypage.

Les paramètres liés à l'analyse du modèle sont implémentés sous forme de constantes. Il est intéressant de placer les données spécifiées par le concepteur du modèle dans un fichier de configuration consulté au démarrage de

l'application. Leur modification n'entraînera pas la recompilation du prototype.

Aucun service n'est offert dans le cadre de l'initialisation logique du prototype.

5.1.2. Terminaison

Le module de contrôle doit connaître l'état de l'ensemble des tâches composant le prototype. Lorsque la phase d'initialisation est terminée, elles sont toutes actives. Leur terminaison doit être signalée au module de gestion du contrôle qui mettra à jour sa connaissance de l'état du prototype.

Le module de gestion du contrôle commence son exécution au démarrage du prototype. Il réalise alors la phase d'initialisation.

Les tâches composant le prototype se terminent de deux manières différentes : normalement ou sur une erreur.

Le module de contrôle de l'application doit disposer d'un service `JE_ME_TERMINE` permettant à toute instance d'un processus de l'application de signaler les conditions de sa terminaison. Le module de contrôle prend alors les dispositions qui s'imposent :

- en cas de terminaison normale, seule la connaissance du module de contrôle doit être mise à jour;
- en cas de terminaison sur erreur, deux possibilités sont envisageables :
    - l'erreur est rattrapable (panne d'un site, dans le cas d'une architecture multi-processeurs...) : des mécanismes de reprise sur erreur peuvent être mis en place [Avizienis 87, Nelson 90];
    - l'erreur n'est pas rattrapable (elle provient d'une partie du prototype qui n'a pas été générée automatiquement) : il faut procéder à la terminaison du prototype.

Lorsque toutes les tâches correspondant aux processus instanciés du modèle se sont terminées normalement, le module de contrôle doit procéder à la terminaison des modules de service.

La terminaison des modules de service est asservie. Cependant, ils doivent pouvoir propager un problème intervenant au niveau des primitives fournies par le concepteur d'un système (traitements associés aux Actions, primitives associées aux Ressources externes). Un service `PROBLEME_EXTERNE` doit être défini à cette fin. Les paramètres de ce service permettent de définir d'où provient l'erreur.

5.2. SERVICES REQUIS

Cette section décrit les services requis, dans le cadre du contrôle du prototype, aux différents modules de l'application (Figure IV.42).

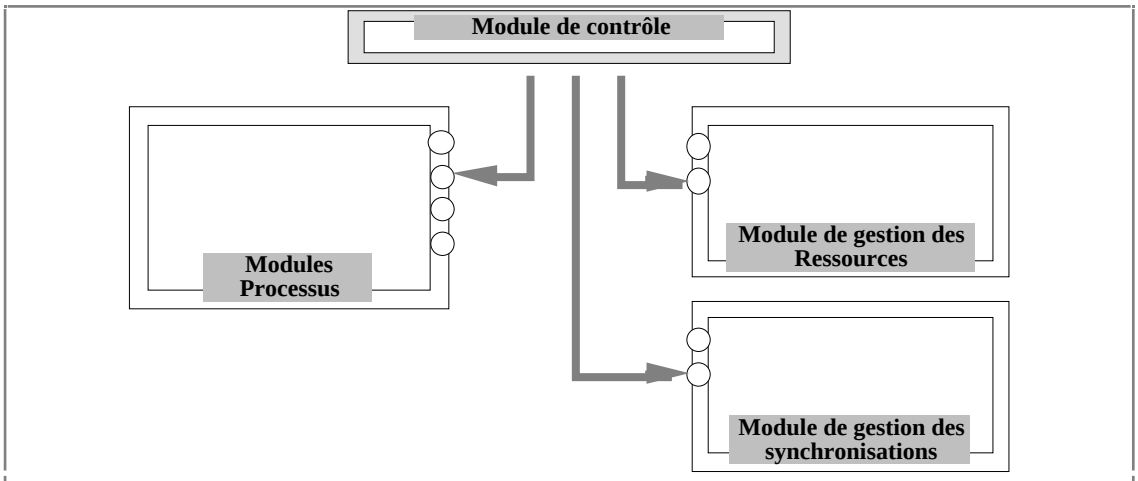


Figure IV.42 : Catégories de services requis aux autres modules du prototype.

Les services requis par le module de contrôle aux autres modules de l'application permettent la transmission d'ordres en vue de réaliser l'initialisation et la terminaison du prototype.

Les modules-processus et les modules de service doivent fournir :

- `INITIALISE_MODULE` : permettant de rendre opérationnelles les fonctionnalités du module concerné.
- `TERMINE_MODULE` : permettant de désactiver le module concerné. Pour les modules-processus, ce service n'est utilisé qu'en cas de terminaison prématurée du prototype. C'est par contre la seule façon de terminer les modules de service.

Comme nous l'avons déjà évoqué, le contrôle des modules externes et du module de traitement n'est pas assuré par le module de contrôle qui, initialement, ne fait pas partie de la spécification du système.

L'initialisation du gestionnaire des Ressources comprend la gestion du marquage initial du modèle. Dans ce cas précis, le module de contrôle manipule une condition qu'il transmet comme un paramètre d'initialisation.

5.3. SYNTHÈSE DES INTERFACES

La Figure IV.43 récapitule les différents services requis et offerts par le module de contrôle.

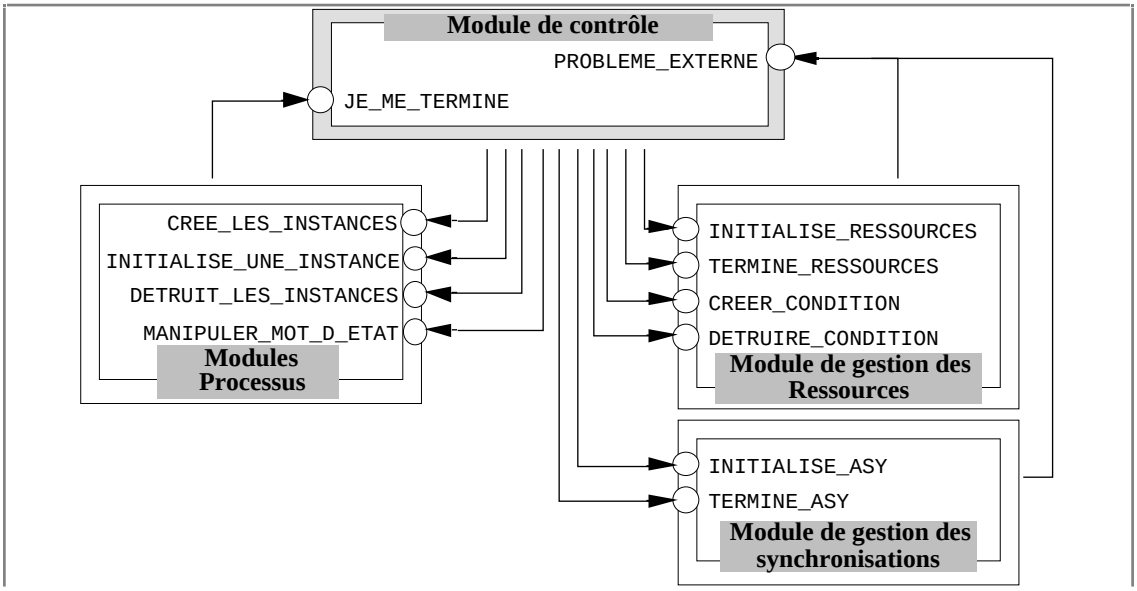


Figure IV.43 : Interfaces du module de contrôle avec les autres modules composant le prototype.

5.4. ARCHITECTURES POSSIBLES POUR LE MODULE DE CONTRÔLE

Des tâches, appelées *gardiens*, s'occupent du contrôle. Deux possibilités sont envisageables :

- Un seul gardien gère l'ensemble du prototype. Une telle solution est intéressante si le prototype s'exécute sur une machine mono-processeur.
- Plusieurs gardiens gèrent le prototype. Cette solution est intéressante si le prototype s'exécute sur une architecture multi-processeurs. Chaque gardien gère l'ensemble des tâches présentes sur un site.

Nous étudions dans cette section les deux types d'architecture.

5.4.1. Contrôle effectué par un seul gardien

Considérons dans un premier temps qu'un seul gardien gère l'ensemble des tâches du prototype. Les relations entre le gardien, les modules de service et les modules-processus sont directes (Figure IV.44). Les tâches de l'application signalent au gardien qu'elles se terminent. Le gardien s'adresse aux tâches de l'application, soit pour les initialiser, soit pour provoquer leur terminaison.

Les tâches composant le prototype, contenues dans les différents modules de l'application, s'adressent au gardien qui, seul, a une connaissance globale de l'état du prototype.

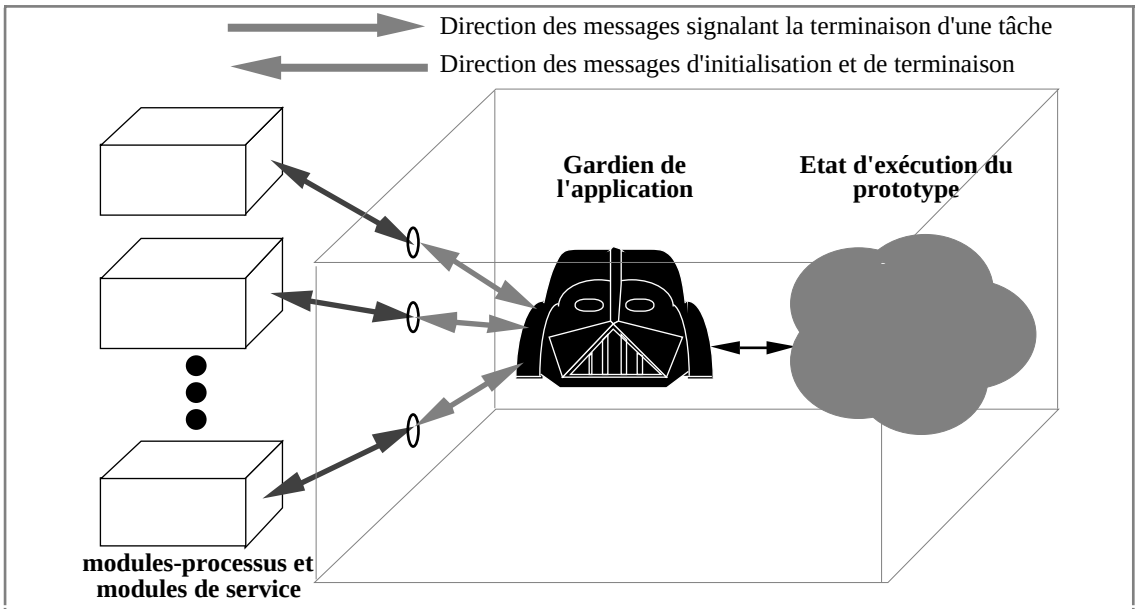


Figure IV.44 : Architecture avec un seul gardien.

5.4.2. Contrôle effectué par plusieurs gardiens

Considérons désormais une architecture à plusieurs gardiens (Figure IV.45). Chaque gardien gère un ensemble de tâches qui lui a été attribué à l'initialisation physique de l'application.

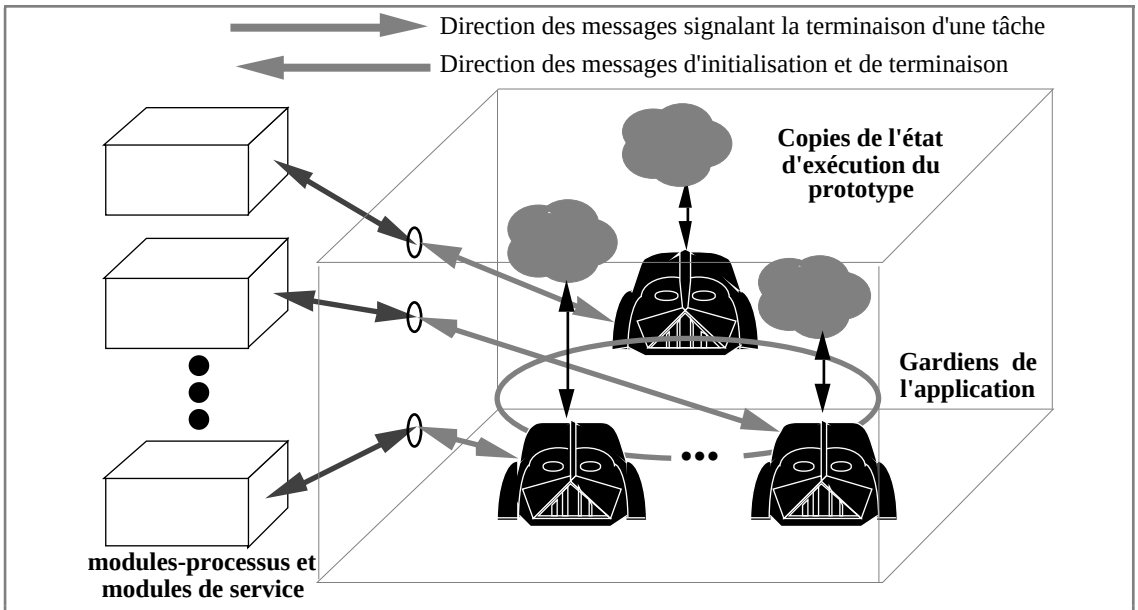


Figure IV.45 : Architecture avec plusieurs gardiens.

Les tâches de l'application s'adressent toujours au gardien qui les a créées. Chaque gardien gère une copie de l'état d'exécution du prototype. Ces copies doivent être identiques : dès qu'un événement modifiant l'état d'exécution du prototype survient, le gardien concerné doit les diffuser à ses homologues. La configuration des liens de communication entre les gardiens peut être :

- un anneau : chaque serveur ne connaît que ses voisins immédiats,
- une clique : chaque serveur connaît l'ensemble de ses homologues.

Le contexte d'exécution du prototype constitue un état global. Des mécanismes doivent être mis au point afin de mettre à jour ce contexte et détecter correctement la terminaison du prototype [Raynal 84, Blanc 90, Raynal 91]. Nous ne les décrirons pas dans ce chapitre.

6. Conclusion

Nous avons décrit dans ce chapitre l'organisation d'un prototype généré automatiquement à partir d'une spécification réseau de Petri.

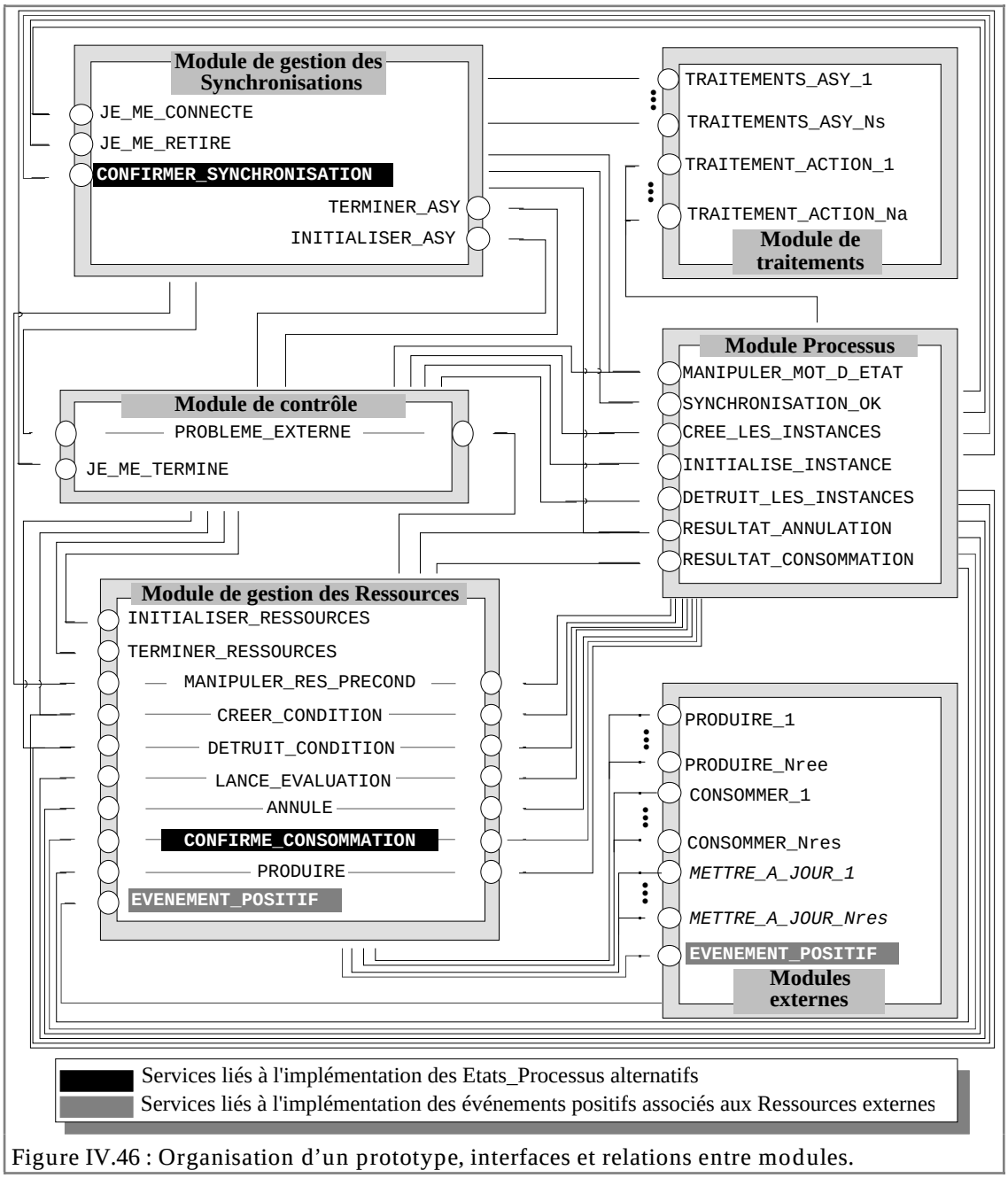


Figure IV.46 : Organisation d'un prototype, interfaces et relations entre modules.

Notre objectif était de donner des directives générales pour l'implémentation des diverses fonctionnalités que nous pouvons déduire de l'analyse du réseau de Petri :

- les Processus du modèle,
- les mécanismes de communication synchrone et asynchrone,
- le contrôle du prototype.

Les règles proposées dans ce chapitre sont indépendantes du langage de programmation du prototype. Nous supposons cependant qu'il est impératif, procédural, et qu'il autorise la gestion du parallélisme.

Les stratégies d'implémentation que nous proposons concernent des programmes multi-tâches. Les choix de certaines politiques, dans la gestion des Ressources, des Actions synchronisées, ou du contrôle de l'application, dépendent de l'architecture cible.

La Figure IV.46 récapitule l'ensemble des services offerts par les modules composant le prototype. Pour chaque service, le requérant est indiqué. Comme nous l'avons évoqué dans ce chapitre, certains d'entre eux dépendent de choix stratégiques :

- La stratégie d'implémentation des Etats_Processus alternatifs a une influence sur le protocole de dialogue entre clients (modules-processus ou gestionnaire des synchronisations) et serveurs (gestionnaire des Ressources ou des synchronisations). Les services `CONFIRMER_SYNCHRONISATION` et `CONFIRME_CONSOMMATION` peuvent ne pas exister (choix a priori d'une Action postcondition).
- La stratégie d'implémentation des événements positifs liés aux Ressources externes peut être gérée de deux façons différentes :
  - le gestionnaire de Ressources interroge régulièrement les composants externes; dans ce cas, le service `EVENEMENT_POSITIF` est offert par les modules externes;
  - les modules externes réveillent le gestionnaire de Ressources dès qu'un événement positif survient; le service `EVENEMENT_POSITIF` est alors offert par le module de gestion des Ressources.

PARTIE II

APPLICATION AU LANGAGE ADA

INTRODUCTION À LA PARTIE II

La deuxième partie de ce document applique au langage Ada la méthodologie de prototypage introduite précédemment. A partir de la structure fonctionnelle du prototype à générer automatiquement, il est nécessaire de détailler les mécanismes à mettre en œuvre pour ce langage.

L'objectif de cette partie est donc de définir précisément le cahier des charges (en terme de résultat à produire) du générateur, puis de détailler son architecture fonctionnelle.

Nous visons deux types d'exécution du prototype : mono-processeur et multi-processeurs. Dans le premier cas, il s'agit d'un prototype centralisé composé de tâches Ada dont l'exécution est implicitement gérée par le scheduler Ada. Le principe de mise en œuvre du prototype réparti repose sur une gestion explicite, par le générateur, de l'exécution de plusieurs programmes Ada sur une architecture multi-processeurs. Il faut donc munir le prototype d'un ensemble de mécanismes de répartition de l'exécution et de gestion d'accès à des ressources réparties. Ces différents mécanismes reposent naturellement sur les services offerts par le système d'exploitation (UNIX™ dans notre cas) et sur les outils de communication et de partage des données associés (NFS, NIS...) [Stevens 90].

La décomposition fonctionnelle que nous avons définie supporte les deux types d'exécution envisagés. Il faut donc identifier, dans les différents modules, les éléments impliqués dans une exécution répartie.

Ainsi, nous définissons le prototype réparti, à partir du prototype centralisé muni d'un service de répartition.

L'architecture du prototype respecte les principes énoncés dans le Chapitre IV. Nous y avons présenté différentes stratégies d'implémentation indépendantes de la nature du prototype (centralisé ou réparti). Ces stratégies portent sur :

- Les Etats_Processus alternatifs : l'implémentation des Etats_Processus alternatifs est effectuée selon la politique du choix a priori d'une Action postcondition (Chapitre IV, section 2.2). Ce choix nous paraît plus équitable.

  Ainsi, les services `CONFIRME_CONSOMMATION` pour le gestionnaire des Ressources, et `CONFIRMER_SYNCHRONISATION` pour le gestionnaire des synchronisations, n'existent pas.
- Le traitement des événements positifs des Ressources externes : le service permettant de gérer les événements positifs est offert par les modules externes. Le gestionnaire de Ressources interroge ces derniers afin de savoir s'il peut relancer l'évaluation d'une précondition bloquée sur une Ressource externe (Chapitre IV, section 3.1.4).

Ce choix a été effectué pour faciliter la réalisation des primitives associées aux Ressources externes associées aux composants externes.

Ainsi, le service `EVENEMENT_POSITIF` n'est pas offert par le gestionnaire des Ressources mais par les modules externes.

Les services offerts par les différents modules, dans le prototype centralisé, sont récapitulés dans la Figure d. Les services requis et offerts pour chacun des modules ne sont pas dépendants du mode d'exécution du prototype. Cependant, la partie privée des modules dont la bordure est ombrée est fonction du mode d'exécution du prototype.

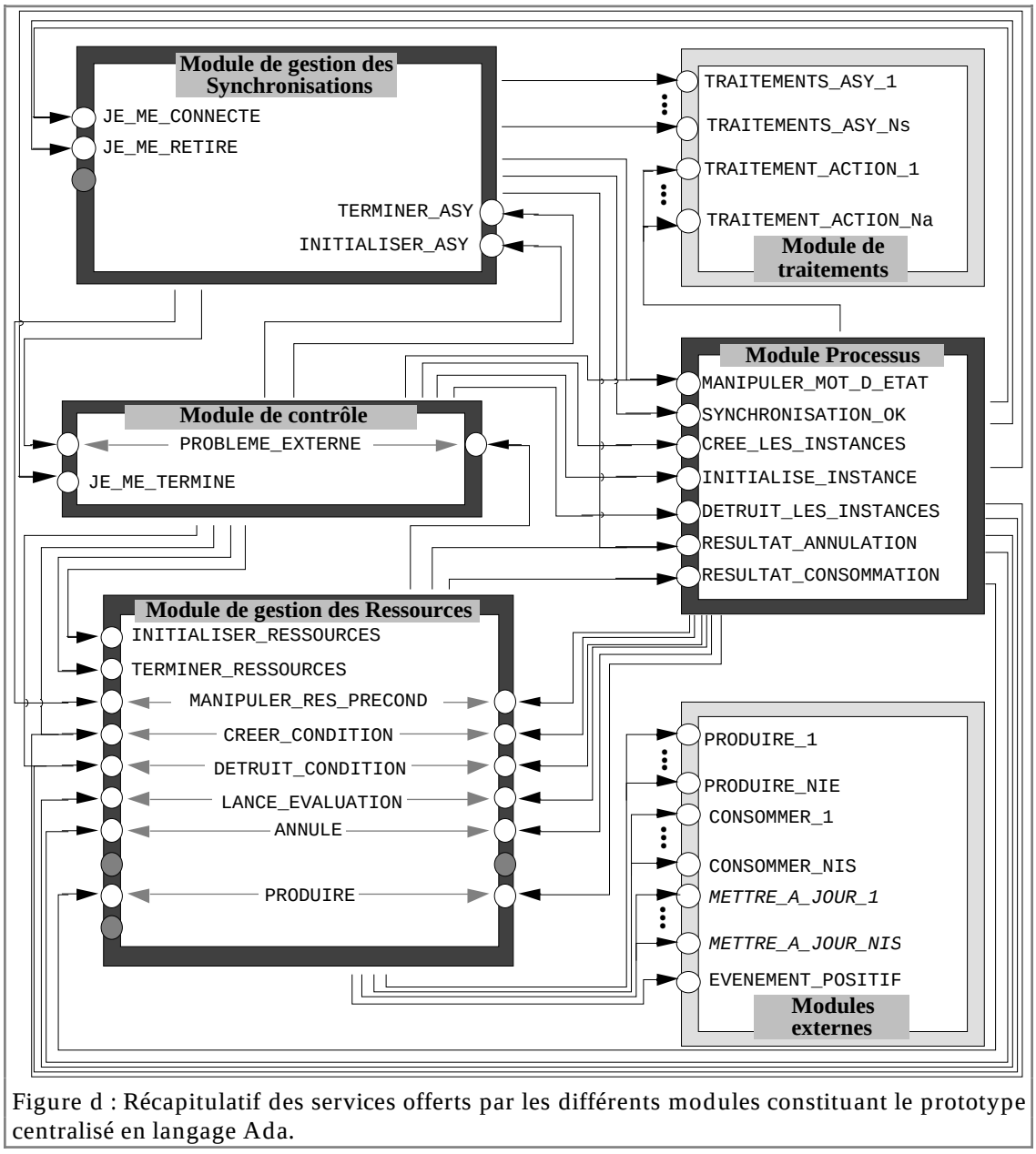


Figure d : Récapitulatif des services offerts par les différents modules constituant le prototype centralisé en langage Ada.

Le Chapitre V présente les caractéristiques du prototype Ada, en fonction de nos contraintes méthodologiques et des mécanismes impliqués.

Le Chapitre VI est consacré à l'étude détaillée du prototype Ada centralisé. Il s'agit de la mise en œuvre des techniques exposées dans le Chapitre IV.

Le Chapitre VII décrit la structure du prototype réparti. Nous exposons dans un premier temps les techniques que nous avons développées en vue de répartir un programme Ada quelconque sur plusieurs sites reliés par un réseau, avant d'aborder l'application de ces principes au prototype.

Le Chapitre VIII décrit l'architecture du générateur que nous avons conçu. Nous présentons sa mise en œuvre dans l'Atelier de Modélisation Interactive (AMI) [Bernard 90].

			Introduction générale
	Chapitre I	:	*Prototypage de systèmes parallèles*
	Chapitre II	:	*Interprétation sémantique d'un réseau de Petri*
	Chapitre III	:	*Méthodologie de prototypage*
	Chapitre IV	:	*Structure du prototype généré*
+	Chapitre V	:	*Caractéristiques du prototype Ada*
	Chapitre VI	:	*Le prototype centralisé*
	Chapitre VII	:	*Le prototype réparti*
	Chapitre VIII	:	*CPN/TAGADA : L'outil de prototypage*
			Conclusion générale
			Bibliographie
			Annexes & Appendices

Chapitre V :

Caractéristiques du prototype Ada

1. Introduction 199
2. Modules fonctionnels et paquetages Ada 201
   - 2.1. Modules internes 201
   - 2.2. Les modules externes et le module de traitement 202
3. Les unités propres au prototype Ada 204
   - 3.1. Evaluation du prototype 204
   - 3.2. Le mécanisme de communication client-serveur 207
4. Les unités de "bas niveau" 212
   - 4.1. Informations générales sur le prototype 212
   - 4.2. La gestion des marques 213
5. Conclusion 216

1. Introduction

Dans ce chapitre, nous présentons les caractéristiques essentielles du prototype lorsqu'il est généré en langage Ada. Ces caractéristiques sont communes aux prototypes centralisé et réparti.

Nous avons choisi le langage Ada, d'une part parce qu'il respecte les hypothèses de base nécessaires à l'application de nos techniques de prototypage (il est procédural et permet la manipulation du parallélisme) mais aussi pour les aspects suivants :

- la notion d'*unité de programme*, permettant de diviser le prototype généré en paquetages avec des parties privées et visibles;
- la notion d'*exception*, permettant d'élaborer un mécanisme de détection des erreurs fort utile en phase de débogage, que ce soit celui du générateur de code (et donc du prototype) ou celui des unités fournies par l'utilisateur;
- la *généricité*, permettant une réutilisation adaptée des unités;
- les types *polymorphes* et *mutables*, permettant de gérer des structures de données complexes tout en conservant une sécurité dans la vérification des types dans les expressions;
- la notion de *sous-types* et *types dérivés*, permettant de regrouper des littéraux de façon logique, tout en conservant à la fois un classement et une vérification des types dans les expressions;
- enfin, il constitue un standard : sa portabilité n'est plus à démontrer.

Certains aspects du langage ont fortement été utilisés (paquetages, tâches, sous-types, types polymorphes, pointeurs...). Par contre, d'autres le sont moins, c'est le cas de la généricité.

La faible utilisation de la généricité est motivée par les raisons suivantes :

- La complexité des modules pour lesquels ce concept est applicable (gestion des Ressources et des synchronisations) rend nécessaire des imbrications de génériques comportant beaucoup de paramètres. Les temps de compilation s'allongent et le débogage en devient éprouvant.
- Les unités génériques doivent traiter toutes les configurations possibles, empêchant ainsi la réalisation d'optimisations pouvant accroître les performances d'exécution.

C'est pourquoi nous avons restreint l'utilisation de la généricité à une unité de service : un gestionnaire de collections ordonnées d'objets. Cette unité possède quatre paramètres génériques formels et le nombre important d'utilisations (au niveau des modules de service, par exemple) motive son existence.

Certains mécanismes de bas niveau, trop dépendants du langage cible, n'ont pas été définis dans le Chapitre IV. Ainsi, le prototype Ada contient-il des unités supplémentaires, divisées en deux groupes :

- les unités propres au prototype Ada : leur existence est motivée par des caractéristiques du langage Ada;

- les unités “de bas niveau” : elles décrivent des concepts et des structures de données manipulées par tous les modules du prototype.

Le type de prototypage visé (Chapitre I, section 1.4) correspond à une *approche par raffinements* [Floyd 84, Hallmann 91]. Pour être compilé dans n’importe quel environnement supportant Ada, le prototype que nous générons ne repose que sur des concepts définis dans la norme [Ada 83].

Une contrainte essentielle était de permettre la mise en œuvre du processus itératif défini dans l’approche par raffinements. Nous avons donc conçu le prototype afin que son concepteur puisse revoir sa spécification ou corriger les modules externes, en vue de produire la version suivante de son application.

Ainsi, notre prototype fournit des informations sur son fonctionnement, permettant :

- d’évaluer le comportement du modèle afin d’étudier le comportement du système;
- d’éprouver les modules externes ou le module des traitements au sein du système grandeur nature, après la phase de tests unitaires.

En fonction des spécificités (contraintes temporelles, ressources particulières...) de certaines applications faisant l’objet d’une génération automatique, il est nécessaire de pouvoir ponctuellement adapter le prototype, pour en tirer des informations particulières, ou pour changer certaines stratégies. Ainsi, nous avons systématiquement privilégié les structures privées encapsulées dans des unités.

Après avoir présenté notre stratégie pour exprimer en unités Ada les modules fonctionnels identifiés dans le Chapitre IV, nous décrivons la partie privée de ceux qui sont communs aux prototypes centralisé et réparti.

L’annexe B présente le modèle d’un système devant être généré *automatiquement*. Plusieurs extraits caractéristiques du code généré illustrent les stratégies mises en œuvre.

2. Modules fonctionnels et paquetages Ada

Le concept d'*unité de programme,* défini dans [Ada 83], est réalisé sous la forme de paquetages (modules) ou de sous-programmes (procédures et fonctions). La décomposition du prototype Ada repose essentiellement sur cette notion.

Le Chapitre IV a présenté deux types de modules dans le prototype :

- les modules internes, réalisant une fonction du prototype liée à une catégorie de G-objets;
- les modules externes, liés aux composants externes, et le module de traitement, définissant les Actions exécutées par les processus instanciés du prototype.

Nous allons détailler, dans ce paragraphe, la manière dont nous réalisons ces deux catégories de modules.

2.1. MODULES INTERNES

Les modules fonctionnels décrits dans le Chapitre IV sont réalisés à l'aide de paquetages. Nous définissons les équivalences entre les composantes d'un module et les paquetages Ada (Figure V.1) :

- les services offerts sont réalisés sous la forme de primitives dont le profil est donné dans la spécification du paquetage correspondant;
- les services requis apparaissent sous la forme de clauses de visibilité sur les paquetages qui les offrent;
- le comportement propre du module (relation entre services offerts et services requis, réalisation des services...) est défini dans le corps du paquetage.

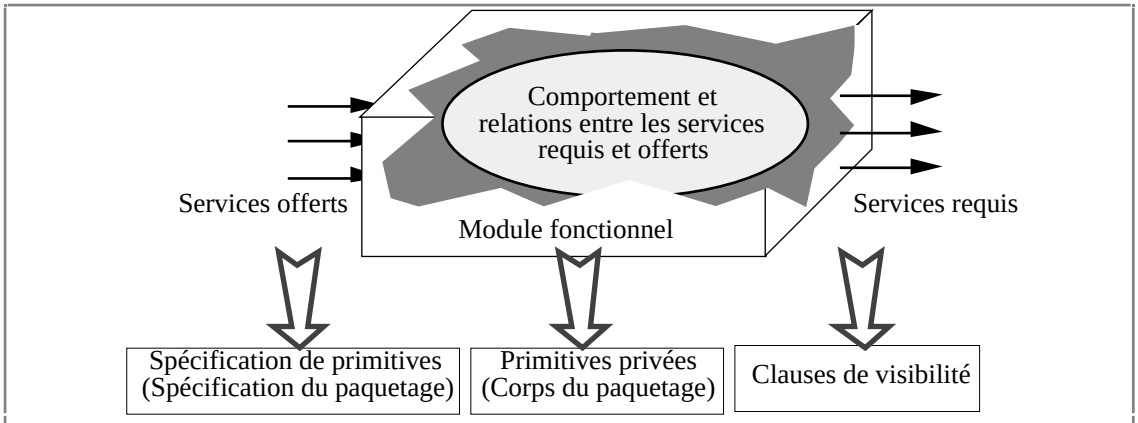


Figure V.1 : Réalisation des services (requis, offerts) et de la partie privée d'un module fonctionnel en Ada.

Pour une meilleure précision et lisibilité du prototype, les modules offrant un grand nombre de primitives ont été divisés en fonction de "sous-ensembles de services".

Afin de partitionner un module interne en plusieurs paquetages tout en respectant la décomposition fonctionnelle décrite dans le Chapitre IV, nous introduisons la notion de *paquetage écran*.

***Définition V.1 :* Paquetage écran**

Un paquetage écran regroupe l'ensemble des services offerts aux autres modules du prototype. Ce paquetage redéfinit les primitives visibles d'autres paquetages par renommage et les structures de données à l'aide de sous-types (Figure V.2).

Les clauses de visibilité associées aux services requis sont définies autant de fois qu'il y a de paquetages réalisant le module fonctionnel.

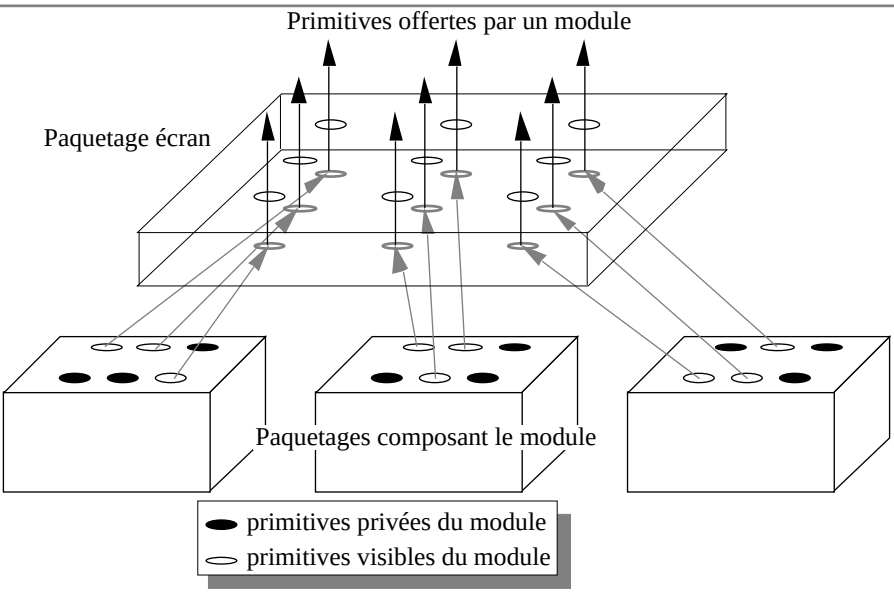


Figure V.2 : Réalisation d'un module du prototype à l'aide d'un paquetage écran.

2.2. LES MODULES EXTERNES ET LE MODULE DE TRAITEMENT

Les modules externes, comme le module de traitement, sont décrits par le concepteur de l'application. Les primitives implémentées doivent respecter un format précis connu du prototype.

Les modules externes

La spécification des primitives associées aux Ressources externes respecte les règles énoncées dans le Chapitre III (Définitions III.4 et III.7). Si les marques sont colorées, les paramètres de ces primitives sont du type associé à la classe de couleurs, ou au domaine de couleurs de la marque concernée.

La transformation des valeurs transmises en marques manipulables par les différents composants externes est à la charge du gestionnaire des Ressources.

Le module de traitement

Pour limiter la taille de l'unité contenant les primitives associées aux Actions du modèle, le module de traitement est divisé en plusieurs paquetages :

- un paquetage par classe de processus, contenant les procédures associées aux Actions simples regroupées dans la classe considérée;
- un paquetage regroupant l'ensemble des Actions synchronisées du modèle.

Les procédures associées aux Actions simples comportent les paramètres suivants :

- le mot d'état du processus instancié, s'il existe;
- l'identité du processus instancié.

Les procédures associées aux Actions synchronisées comportent les paramètres suivants :

- le mot d'état de chacun des participants qui en possède un;
- l'identité de chacun des processus instanciés participants.

Par défaut, un corps vide, ou comportant une simple trace, est associé à ces procédures.

3. Les unités propres au prototype Ada

Dans cette section, nous décrivons des unités, propres au prototype Ada, qui n'ont pas été présentées dans le Chapitre IV car elles correspondent à des choix de réalisation. Elles sont liées aux contraintes du langage cible. Il s'agit des unités suivantes :

- L'unité de communication supportant les interactions asynchrones entre modules-serveurs et modules-clients. Le rendez-vous Ada se prête en effet mal, tel qu'il est défini dans [Ada 83], aux protocoles de communication tels qu'ils ont été définis [Ichbiah 86, Richard Foy 91].
- Des unités de mise au point gèrent l'ensemble des informations liées à l'exécution du prototype.

3.1. EVALUATION DU PROTOTYPE

Le concepteur du modèle intègre le prototype généré dans un environnement réel. Il peut alors tester les programmes contenus dans ses librairies (procédures associées aux Actions, primitives d'interface associées aux composants externes...). A cette fin, des mécanismes de trace ont été développés; ils permettent d'obtenir des informations sur l'état du prototype (notre générateur ayant lui-même été écrit en Ada, ces mécanismes ont été utilisés pour le déboguer).

Durant l'exécution du prototype, les erreurs potentielles proviennent :

- Du code généré automatiquement : cela ne doit plus arriver dans la phase d'exploitation du générateur de code. Ces erreurs signifient que les modules de service sont dans un état incohérent; aucune reprise n'est alors possible. Le module de contrôle provoque l'arrêt de l'application.
- Des unités définies par le concepteur du modèle : la récupération de l'erreur n'est pas du ressort du module de contrôle.

Notre mécanisme de trace permet d'observer :

- la séquence qui a abouti à l'erreur car différentes exécutions d'une application parallèle peuvent engendrer des problèmes différents;
- la pile des appels au moment de l'erreur permettant ainsi de localiser l'unité fautive.

Le prototype Ada gère les erreurs à l'aide d'exceptions. Si le concepteur du modèle souhaite obtenir des informations relatives à son exécution, il commute la génération en mode DEBUG. Dans le cas contraire, les mécanismes décrits dans cette section ne sont pas générés.

Un des mécanismes clefs du langage Ada repose sur le traitement des exceptions. Nous le présentons brièvement avant de développer les mécanismes de traces conçus afin d'obtenir un historique de l'exécution du prototype. Enfin, nous expliquons comment nous obtenons la pile des appels.

3.1.1. Rappel sur les exceptions en Ada

Le langage Ada permet, pour le traitement des erreurs et des situations exceptionnelles, de provoquer des échappements et de définir des points de reprise. Tout ceci est réalisé à l'aide d'*exceptions* [Ada 83].

En levant une exception, on abandonne l'exécution normale du programme. L'exception est alors propagée jusqu'à un point de reprise. Si aucun point de reprise n'est défini dans la pile d'appels de sous-programmes pour l'exception qui est levée, l'exécution de l'application s'achève.

Dans le manuel de référence du langage, ces points de reprise sont dénommés *traite_exception*. Ils sont définis :

- à la fin d'un sous-programme (procédure ou fonction);
- à la fin d'une instruction bloc;
- à la fin du corps d'un paquetage;
- à la fin du corps d'une tâche.

Il existe deux types d'exceptions :

- les exceptions prédéfinies, qui caractérisent des cas d'erreurs propres au langage ou à son implémentation,
- les exceptions définies par le programmeur, qui correspondent aux cas d'erreurs qu'il souhaite gérer.

Une exception peut être levée explicitement par un programmeur, qu'elle soit prédéfinie ou propre à son application. De la même manière, il est possible, au niveau d'un point de reprise, de propager une exception que l'on vient de rattraper.

Exemple V.1 : Le concepteur d'une application peut décider de lever lui-même une exception afin de propager à l'appelant une situation incohérente. Cela se fait au moyen de l'instruction `raise`. Dans l'exemple que nous donnons ci-dessous, l'exception `OUT_OF_RANGE` est définie par l'utilisateur.

```
if INDICE > MAX then
   raise OUT_OF_RANGE;
end if;
```

Les exceptions prédéfinies du langage peuvent également être levées à l'exécution incorrecte d'une instruction.

Exemple V.2 : L'exécution de l'instruction `x := Y / Z;` provoque la levée de `NUMERIC_ERROR`[9] (calcul d'une expression numérique dont le résultat ne peut être représenté) si Z vaut 0. La levée de cette exception n'a pas été explicitement définie par le concepteur de l'application.

---

9 Ce sera vraissemblablement `CONSTRAINT_ERROR` dans la norme Ada9X.

3.1.2. Trace du fonctionnement du prototype

Le prototype peut générer une trace au début et à la fin de tous les sous-programmes qui le composent. La trace d'entrée dans le sous-programme indique les paramètres en entrée. La trace de fin de sous-programme indique sa valeur de retour lorsqu'il s'agit d'une fonction.

Toutes les traces d'une unité sont gardées par une constante booléenne privée (définie dans le corps de l'unité) positionnée à `FALSE`. Ainsi, par défaut, même en mode trace, le prototype ne génère pas d'affichages parasites. Il suffit de positionner à `TRUE` le booléen de garde afin de les obtenir pour l'unité concernée.

Un problème concerne cependant l'affichage en Ada. [Ada 83] ne spécifie pas que l'unité d'entrée-sortie `TEXT_IO` doive gérer des affichages en parallèle. Dans certains environnements, ou pour certains compilateurs, cela peut engendrer une exécution anormale du programme. En mode DEBUG, nous générons par conséquent une unité de trace protégeant les entrées-sorties. La spécification de ce paquetage est donnée dans Code 1 (Annexe B).

Deux primitives, `TRACE_AVEC_RC` et `TRACE_SANS_RC`, permettent d'afficher des messages sur la sortie standard avec ou sans saut de ligne. L'affichage est gardé par un sémaphore réalisé au moyen d'une tâche Ada.

Deux primitives, `INITIALISE_TRACE` et `TERMINE_TRACE`, sont utilisées par le module de contrôle pour initialiser et détruire le sémaphore.

3.1.3. Génération de la pile des appels

Lorsqu'une exception n'est pas rattrapée, le programme Ada compilé génère parfois (cela dépend des compilateurs) la pile des appels au moment où l'exception a été provoquée.

Les exceptions levées par des tâches ne sont pas propagées au maître [Ada 83]. L'exécution du programme continue donc jusqu'à ce que la disparition de la tâche provoque d'autres exceptions (communication avec un partenaire inexistant...) ou génère un interblocage. A ce stade il est en général trop tard pour facilement faire un diagnostic.

Le mécanisme de signalisation des erreurs dont nous avons doté le prototype (services `PROBLEME_EXTERNE` et `JE_ME_TERMINE`) permet de provoquer sa terminaison dès qu'une tâche se termine anormalement. C'est le module de contrôle qui s'en charge.

Des exceptions sont définies pour signaler les situations non rattrapables. Ainsi, il n'y a *jamais* levée explicite d'une exception prédéfinie du langage dans le prototype. Le nom de l'exception constitue une information intéressante dans le diagnostic d'un problème.

Les sous-programmes du prototype disposent de points de reprise (traite_exception) dans lesquels le nom de l'unité concernée est empilé.

Ensuite, l'exception est propagée. Juste avant de se terminer, le gardien de l'application écrit, dans un fichier, tous les messages empilés de la sorte.

Ce dispositif permet de localiser rapidement l'origine du problème, qu'il provienne du squelette du prototype, en phase de débogage du générateur de code, ou des modules externes.

La gestion de la pile des messages engendrés en cas d'erreur est réalisée par une unité dédiée : `MECANISME_DE_DIAGNOSTIC`. La spécification de cette unité est donnée dans Code 2 (Annexe B).

Une primitive, `POSITIONNER_DIAGNOSTIC`, empile un message. Une autre primitive - `RESTITUER_DIAGNOSTIC` - écrit la pile des messages dans un fichier.

3.1.4. Le mode DEBUG

Les mécanismes d'auto-évaluation du prototype entraînent une augmentation de la taille du code du prototype. C'est pourquoi ils ne sont générés que si le mode DEBUG est positionné.

Nous donnons, dans Code 3, un exemple de primitive générée en mode DEBUG. Elle est chargée d'appliquer la fonction successeur à une marque non composée. Si la marque est composée, elle doit lever une exception. Ce cas de figure ne peut arriver qu'en phase de débogage du générateur de code ou si le concepteur du modèle a mal validé son système : le problème lui est alors signalé.

Toutes les traces d'exécution sont gardées par le booléen `TRACE` , qui est une constante locale à l'unité, positionnée par défaut à `FALSE`.

La primitive donnée dans Code 3 peut, dans certains cas, lever l'exception `L_EXEMPLE_PANIQUE_MARQUAGE`. Lorsque cela arrive, la raison de l'erreur est indiquée dans la pile des appels. Si une exception survient pendant l'appel d'un sous-programme (`L_EXEMPLE_SUCC_OU_ALLER` par exemple), l'identificateur de la primitive est empilé au niveau du traite_exception avant propagation de l'exception. La pile complète des appels est ainsi constituée.

Nous avons indiqué en italique l'ensemble du code lié au mode DEBUG. S'il n'est pas positionné, le code généré devient plus simple.

La spécification et le corps des deux unités liées au mode DEBUG sont indépendants du réseau de Petri décrivant le système.

3.2. LE MÉCANISME DE COMMUNICATION CLIENT-SERVEUR

Cette section décrit les mécanismes que nous avons conçus pour réaliser le protocole de communication entre module-client et module-serveur.

3.2.1. Motivation

Le modèle du multi-tâches Ada fournit un seul mécanisme pour la synchronisation et la communication entre tâches [Richard Foy 91].

Les mécanismes de communication client-serveur que nous avons décrits dans le Chapitre IV (section 2.5.1) impliquent l'existence de communications asynchrones dans le sens serveur ß client.

Les communications entre tâches Ada s'effectuent au moyen de points d'entrée. Le manuel de référence du langage spécifie que, lorsqu'une tâche en appelle une autre, l'appelante reste suspendue pendant que l'appelée réalise le point d'entrée.

Le comportement du rendez-vous Ada correspond à ce qui est décrit dans (a) (Figure V.3) : le client appelle un serveur et suspend son exécution jusqu'à réalisation du service demandé. La tâche serveur est bien active mais elle ne peut prendre en considération d'autres demandes. Si le service est bloquant (l'évaluation d'une précondition), il y a risque d'interblocage.

Nous souhaitons avoir le comportement (b) (Figure V.3). Le client effectue une demande de service puis se met en attente d'une réponse. Le serveur reste disponible pour d'autres demandes et réveille le client dès qu'il a pu réaliser le service.

Une première implémentation, à l'aide de plusieurs points d'entrée, peut être envisagée : le client se connecte au serveur qui enregistre sa demande. Il se bloque ensuite en attente d'un point d'entrée appelé par le module-serveur, après réalisation du service demandé.

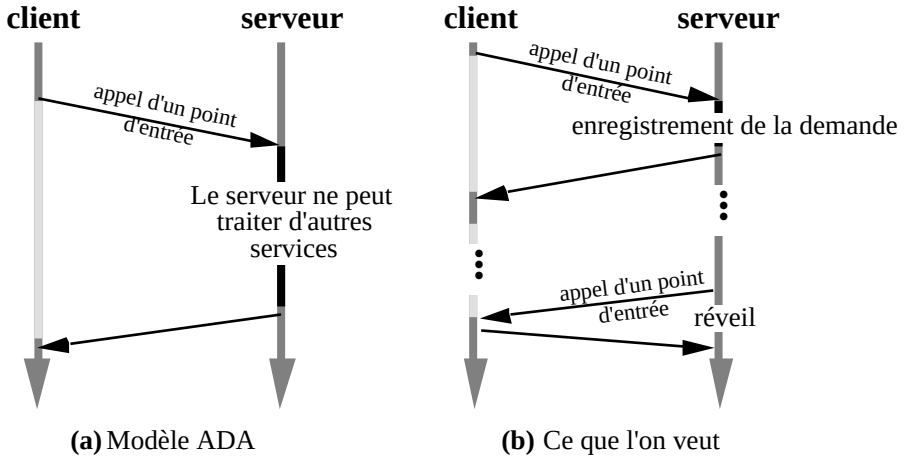


Figure V.3 : Comportement défini par le modèle Ada de communication entre tâches comparé à celui dont nous avons besoin.

Deux problèmes se posent cependant :

- Les modules-serveurs doivent connaître les services de réveil de tous leurs clients.
- En cas d'expiration d'une temporisation au moment où le client concerné demande une annulation, le serveur est en attente de l'accept d'une

tâche, elle-même en attente d'un accept du serveur concerné. Il y a interblocage. Une solution consiste à confier au serveur la gestion des temporisations des clients pouvant annuler des demandes bloquantes. Cela nous semble néanmoins une surcharge inutile.

Des techniques, basées sur l'utilisation de tâches intermédiaires (*agents*), sont proposées dans [Ichbiah 86]. Le mécanisme que nous avons développé repose sur ce principe.

3.2.2. Description de l'unité

L'unité offrant des services de communication client-serveur doit permettre :

- à un client de bloquer son exécution pour une durée déterminée ou indéfiniment,
- à un serveur de réveiller un client.

Un type énuméré permet de spécifier le comportement souhaité par un client. Pour le moment, le calcul des délais de temporisation est effectué par l'unité de gestion des communications.

Il existe une primitive permettant à un client de suspendre son exécution en attendant la réalisation du service qu'il demande. Si l'attente n'est pas temporisée, le client n'est réveillé qu'après la réalisation du service qu'il a demandé. Dans le cas contraire, il doit savoir s'il est réveillé à la fin de la temporisation ou sur réception d'un résultat. Lorsqu'il reprend son exécution, une convention, dans le résultat transmis, lui permet de savoir s'il est réveillé sur expiration d'une temporisation ou non.

Les modules-serveurs utilisent des primitives dédiées pour réveiller un client lorsque le service bloquant demandé a été réalisé.

En cas de réveil sur réalisation d'une précondition, le module-serveur doit transmettre le marquage consommé. En cas de réveil sur réalisation d'une synchronisation, le module-serveur doit transmettre un mot d'état. Comme les résultats à transmettre au client ne sont pas de même nature, les primitives doivent être dupliquées :

- Il existe une primitive permettant aux modules-processus de se bloquer en attente de la réalisation d'une précondition. Une autre permet au gestionnaire de Ressources de réveiller un client.
- Pour chaque module-processus possédant une Action synchronisée, il existe une primitive permettant de se bloquer en attente d'une synchronisation. De même, il existe une primitive dédiée à la transmission des résultats pour chacun des Processus susceptibles de se synchroniser.

3.2.3. Mise en œuvre

Le mécanisme fonctionne à l'aide de *tâches d'interface* chargées de gérer les communications et les temporisations. Les tâches d'interface sont des tâches agents capables d'intégrer les différents comportements définis dans le

protocole de dialogue entre module-client et module-serveur. Le principe de fonctionnement est décrit par la Figure V.4.

Dans un premier temps, un client effectue une demande de service bloquante qui est enregistrée (étape 1) par le module serveur. Le demandeur appelle ensuite la primitive d'attente du paquetage de communication, se bloquant ainsi sur une tâche d'interface (étape 2). Si le service est rendu avant expiration d'une temporisation (étape 3a), la tâche d'interface, réveillée par le module serveur, transmet le résultat qu'elle a reçu. A l'expiration de la temporisation, (étape 3b) elle réveille le client et lui rend un résultat qui par convention, signifie que le service n'a pu être réalisé.

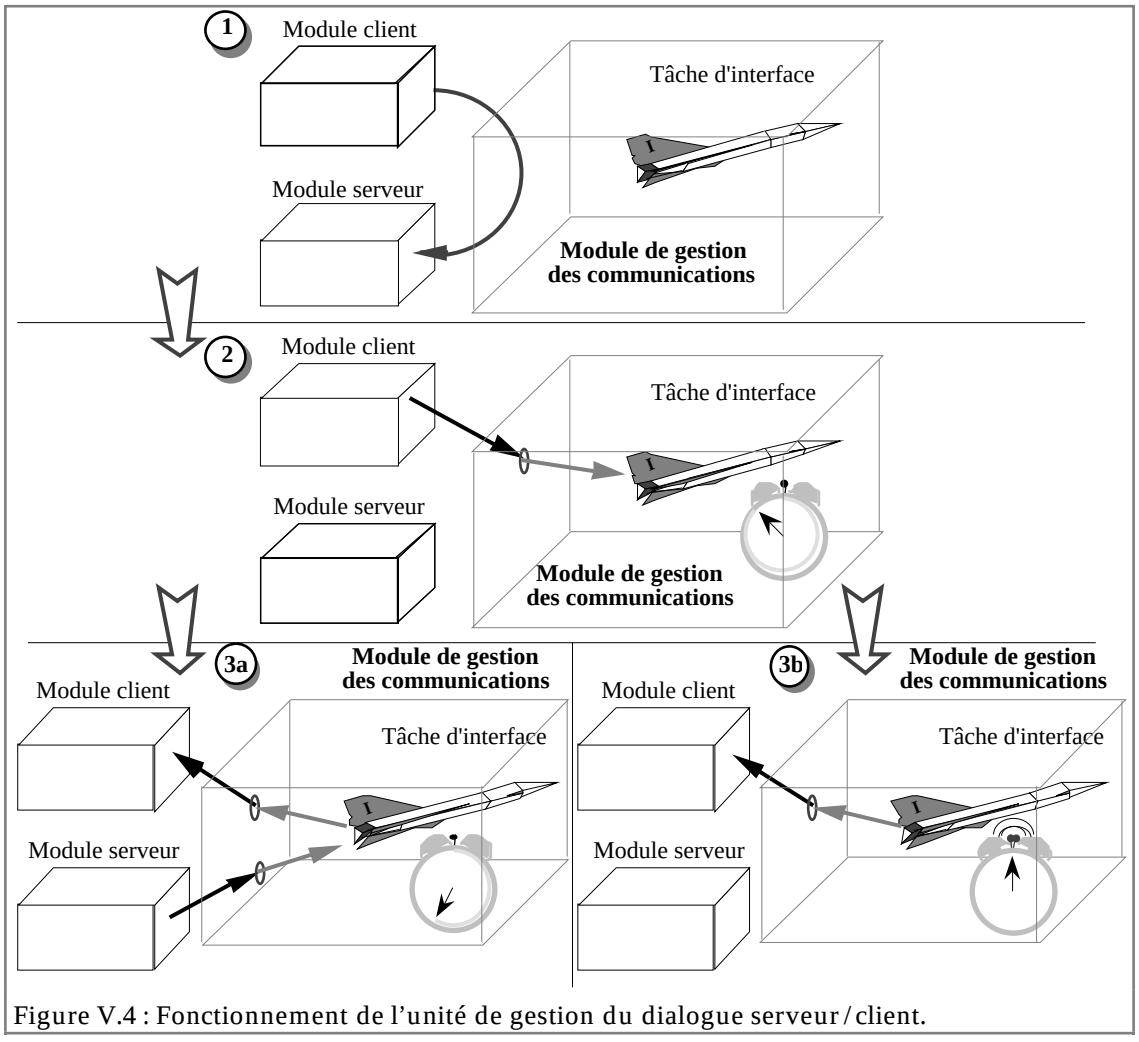


Figure V.4 : Fonctionnement de l'unité de gestion du dialogue serveur / client.

Si les clients doivent attendre indéfiniment la réalisation d'un service bloquant, le scénario est alors composé des seules étapes 1, 2 et 3a.

Il existe une tâche d'interface pour chacun des processus instanciés susceptibles d'effectuer une demande au module de gestion des Ressources ou au module de gestion des Actions synchronisées[10]. Elles sont créées à

[10] Des tâches d'interface géreront de la même manière le dialogue entre le module de gestion des Actions synchronisées et le module de gestion des Ressources lorsque l'hypothèse 5 (références à

l'initialisation du prototype; leur durée d'existence est celle du client auquel elles sont associées.

Les tâches d'interface sont passives [Richard Foy 91] puisqu'elles sont toujours bloquées sur un `accept`. Nous donnons dans Code 4 un exemple de spécification de l'unité de gestion des communications.

L'initialisation et la terminaison de ce mécanisme sont à la charge du module de contrôle.

---

des Ressources dans une Action synchronisée) définie dans le Chapitre VIII en section 3.2 sera levée.

4. Les unités de “bas niveau”

Nous détaillons dans ce paragraphe les paquetages qui décrivent les structures de données utilisées par les différents modules composant le prototype, c’est-à-dire :

- une unité définissant des structures manipulées par l’ensemble du prototype (Etats_Processus, estampilles).
- des unités de gestion du marquage : les classes, les fonctions de base sur les classes, les domaines de couleurs, les structures décrivant un marquage et les fonctions de manipulation de marques sont décrites dans ces unités;

4.1. INFORMATIONS GÉNÉRALES SUR LE PROTOTYPE

Les structures de données regroupées dans cette unité sont les suivantes :

- un type permettant d’identifier les Processus;
- les états actifs de tous les Processus du modèle;
- la description d’une estampille, pour classer les demandes;
- la description d’une structure permettant d’identifier un processus instancié de façon unique.

Nous regroupons les états actifs de tous les processus du modèle dans un type énuméré Ada. Cela autorise la manipulation des états actifs par le module de contrôle dans la phase d’initialisation du prototype. Chaque module processus redéfinit ses états actifs comme un sous-ensemble (sous-type) du type ainsi défini.

Pour identifier les différents processus instanciés du modèle, nous utilisons le couple de valeurs suivant :

<processus, numéro d’instance>

Le numéro d’instance est attribué par le module de contrôle à l’initialisation du prototype. Cette identité est utilisée :

- à chaque demande de service (synchronisation, évaluation d’une précondition ressource...),
- pour signaler la terminaison d’une instance.

Outre la définition de la structure, l’unité définit des primitives de traitement permettant de manipuler des identités.

Enfin, nous y définissons également un type estampille permettant d’ordonner des demandes. Ces estampilles sont manipulées par les modules-processus et par le gestionnaire de Ressources.

Nous donnons, dans Code 5, un exemple de spécification de ce paquetage.

4.2. La gestion des marques

Les marques d'un réseau de Petri coloré sont typées (Définition I.6). Il faut pouvoir respecter le type de chaque marque tout en autorisant une manipulation aisée.

Pour cela, nous avons défini un type *marquage*, pouvant contenir une marque de l'un des formats (classe, domaine) indiqué dans le modèle décrivant le système.

Dans un premier temps, nous expliquons comment nous réalisons les classes (décrivant des marques simples) et les domaines (définissant les marques composées). Ensuite, nous décrivons le type marquage et les fonctions qui lui sont applicables.

4.2.1. Définition des classes de couleurs

Les AMI-Nets (formalisme décrivant les réseaux de Petri Colorés dans l'atelier AMI) autorisent trois types de classes [Bonnaire 92a] :

- les classes de type numérique : les marques ont une valeur numérique, éventuellement comprise entre deux bornes;
- les classes de type caractère : les marques prennent leurs valeurs dans l'ensemble ou des sous-ensembles de caractères;
- les classes énumérées quelconques : les marques ont une valeur parmi celles définies dans un intervalle.

Dans les trois cas, une classe de couleurs s'assimile à un ensemble fini de valeurs. Elle peut être réalisée au moyen d'un type Ada discret. Nous utilisons des types entier, caractère et énuméré.

Pour faciliter la définition du type marquage, nous avons décidé d'intégrer la notion de type universel. Chaque type de classe (numérique, caractère, énumérée) est défini par rapport à un "super-type" prenant l'ensemble des valeurs possibles pour cette catégorie. Pour les classes numérique et caractère, le type englobant est prédéfini dans le langage Ada. Il faut le créer pour les classes énumérées.

Nous définissons ainsi chaque classe de couleurs comme un sous-type Ada du "super-type" associé. La définition du type marquage s'en trouve simplifiée et nous ne perdons pas les avantages liés au fort typage : lorsqu'un paramètre ne respecte pas les bornes définies, l'exception `CONSTRAINT_ERROR` est levée à la première manipulation.

Exemple V.3 : Considérons les classes de couleurs suivantes :

- C1, composée des valeurs v1, v2 et v3;
- C2, composée des valeurs v4, v5 et v6;
- C3, définie par l'intervalle des valeurs comprises entre 10 et 30;
- C4, définie par l'intervalle des valeurs comprises entre 100 et 200.

Un type énumératif contient l'ensemble des valeurs de C1 et C2. Des sous-types sont ensuite associés à chacune de ces deux classes. Pour les classes C3 et C4, le type universel associé est prédéfini. Elles sont définies comme des sous-types d'entiers. Nous obtenons l'organisation donnée en Figure V.5.

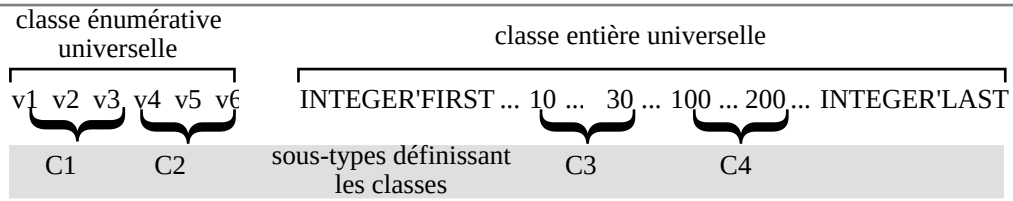


Figure V.5 : Hiérarchie dans la définition des classes de couleurs.

Nous devons également définir des valeurs particulières :

- "n'importe quelle marque", utilisée lors de la définition de masques, pour l'évaluation des préconditions libres (Définition IV.33). Nous adoptons les conventions suivantes :
  - Pour les types numériques, nous utilisons INTEGER'FIRST (première valeur possible définie par l'implémentation). Cette valeur ne pourra jamais faire partie d'une classe numérique.
  - Pour les types caractères, nous utilisons CHARACTER'FIRST (première valeur possible définie par l'implémentation). Cette valeur ne pourra jamais faire partie d'une classe caractère.
  - Pour les types énumérés, nous utilisons une valeur dédiée, c'est la première valeur du super-type définissant tous les littéraux des classes énumérées. Cette valeur n'est incluse dans aucune classe énumérée.
- "aucune valeur", utilisée pour des initialisations par défaut. Cette valeur doit être présente pour chaque classe de couleurs. Pour cela, une valeur particulière est définie.

Nous donnons, dans Code 6, un exemple de paquetage définissant les classes de couleurs. Le paquetage se réduit à une spécification.

4.2.2. Définition des domaines de couleurs

Les domaines de couleurs permettent de définir des marques composées (N-uplets). Chaque composante peut être elle-même un autre domaine, ou une classe.

Chaque domaine est défini par un type article dont les champs correspondent aux diverses composantes du domaine.

Comme pour les classes, il est nécessaire de définir une constante "sans valeur" pouvant être utilisée :

- à des fins d'initialisation, quand aucune valeur n'est précisée,
- pour construire le masque, dans l'évaluation de la précondition libre.

Nous donnons dans Code 7 un exemple de paquetage de définition des domaines de couleurs. Comme pour la définition des classes, le paquetage se réduit à une spécification.

4.2.3. Le type marquage

La description d'une marque est réalisée au moyen d'un type polymorphe : le discriminant de l'article définit la classe ou le domaine de couleurs de la marque, une valeur particulière permet de désigner des marques non colorées.

Un compteur indique le nombre d'exemplaires existant pour le profil (dont le type est donné par la valeur du discriminant) défini dans l'article. Les marques non colorées se réduisent à un compteur.

Pour optimiser la gestion de la mémoire, et faciliter l'accès aux marques, ces dernières sont allouées et désallouées dynamiquement au cours de l'exécution du prototype. Le marquage d'une Ressource est défini comme une liste de marques.

Un certain nombre d'opérations de base sont définies pour le type marquage : successeur et prédécesseur (pour les classes de couleurs), création, destruction, comparaison de deux marques...

Nous donnons dans Code 8 un exemple de spécification du paquetage décrivant le marquage d'un modèle.

5. Conclusion

Nous avons décrit dans ce chapitre, en terme de cahier des charges, les caractéristiques du prototype Ada, communes pour une exécution centralisée ou répartie.

Nous envisageons deux types d'optimisation de l'architecture du prototype. Le premier concerne la généralisation de certaines unités. Cela est déjà le cas pour :

- la gestion du mode DEBUG (traces et génération de la pile d'appels des sous-programmes);
- la gestion de listes générique (beaucoup utilisée dans la gestion des Ressources et des Actions synchronisées).

Ainsi, il n'est pas forcément nécessaire de les générer et, par conséquent, de les compiler (elles sont déjà présentes dans une bibliothèque). La génération de tous les paquetages est cependant nécessaire lorsque l'on souhaite compiler le prototype dans un autre environnement supportant Ada.

Le second type d'optimisation concerne les programmes à générer. Il est motivé pour les raisons suivantes :

- il est inutile de générer des sections de programmes dont nous savons pertinemment qu'elles ne seront jamais exécutées;
- il est inutile de mettre en œuvre des mécanismes complexes lorsqu'ils peuvent être simplifiés.

Actuellement, de telles optimisations portent sur les points suivants :

- Le mécanisme de communication entre client et serveur n'est mis en place que si le modèle comporte des Ressources ou des Actions synchronisées. Si les communications entre Processus sont strictement synchrones (pas de Ressource), ou strictement asynchrones (pas d'Action synchronisée), seuls les traitements correspondants doivent être générés.
- Pour les unités décrivant le marquage :
  - Si le modèle ne comporte pas de domaines de couleurs (il n'existe pas de marques composées), l'unité correspondante ne sera pas produite.
  - Si le modèle n'est pas coloré, le marquage est tout simplement décrit par un entier naturel. Les fonctions sur les marques n'existent pas. Les modules décrivant un certain nombre d'utilitaires, les conditions et le résultat de l'évaluation d'une précondition s'en trouvent considérablement simplifiés. Nous verrons que la structure du serveur de Ressources, quant à elle, ne change pas.

Nous envisageons d'augmenter le nombre des unités indépendantes des données du modèle décrivant le système : si nous considérons les fonctionnalités actuellement définies dans le cadre du projet Ada9x [Ada 91a, Ada 91b], l'utilisation de concepts objets ("types taggés") permettrait la généralisation de certaines unités. De même, les solutions proposées pour la

réalisation des tâches passives (“protected record”) seraient également applicables à la réalisation du mécanisme de communication.

Nous avons décrit le mode DEBUG, générant, avec le prototype, un mécanisme de trace permettant d’obtenir des informations en cas d’exécution anormale. D’autres types d’informations, comme la durée d’exécution de l’application, ou le nombre d’activations de chacune des transitions associées, seraient intéressants à obtenir.

		Introduction générale
	Chapitre I :	*Prototypage de systèmes parallèles*
	Chapitre II :	*Interprétation sémantique d'un réseau de Petri*
	Chapitre III :	*Méthodologie de prototypage*
	Chapitre IV :	*Structure du prototype généré*
	Chapitre V :	*Caractéristiques du prototype Ada*
+	Chapitre VI :	*Le prototype centralisé*
	Chapitre VII :	*Le prototype réparti*
	Chapitre VIII :	*CPN/TAGADA : L'outil de prototypage*
		Conclusion générale
		Bibliographie
		Annexes & Appendices

Chapitre VI :

Le prototype centralisé

1. Introduction 221
2. Réalisation du module de gestion des Ressources 222
   - 2.1. Description d'une condition 222
   - 2.2. Description du résultat d'une précondition 224
   - 2.3. Réalisation des requêtes 224
3. Réalisation des modules-processus 229
4. Réalisation du module de gestion des Actions synchronisées 232
5. Réalisation du module de contrôle 234
6. Conclusion 235

1. Introduction

L'objet de ce chapitre est de détailler les modules internes composant le prototype Ada centralisé.

Conformément aux principes introduits dans le Chapitre IV, et en accord avec les caractéristiques énoncées dans le chapitre précédent, nous présentons l'ensemble des mécanismes générés. L'exécution des tâches composant le prototype est centralisée sur une machine mono-processeur. Les conditions d'exécution d'une tâche sont alors dépendantes de la logique interne au prototype tout en étant soumises à l'indéterminisme du scheduler Ada.

En conséquence, la gestion du marquage des Ressources du modèle, ou du contexte d'exécution du prototype, s'apparente à la gestion de sections critiques. Des mécanismes d'exclusion mutuelle et d'ordonnancement de requêtes sont mis en œuvre.

Plusieurs stratégies d'implémentation ont été proposées pour certains modules internes. Pour le prototype centralisé, nos choix sont les suivants :

- La gestion des Ressources est effectuée à l'aide d'un seul serveur (Chapitre IV, section 3.5.1). Cela permet de résoudre trivialement les problèmes d'accès concurrents au marquage des Ressources du modèle. Une telle architecture n'est pas pénalisante puisque le prototype s'exécute sur un seul processeur.
- Le contrôle du prototype est réalisé par un seul gardien. Les problèmes de mise à jour du contexte global du prototype sont donc trivialement résolus. Comme pour le gestionnaire des Ressources, ce choix n'est pas pénalisant puisque le prototype s'exécute sur un seul processeur.

Ce chapitre présente nos techniques de réalisation des différents modules internes.

La gestion des Ressources, présentée dans la section 2, implique la définition précise des structures décrivant une condition (Définition IV.34). Le format des résultats retournés aux clients, ainsi que les stratégies de réalisation des services offerts, sont également présentés.

La description du comportement d'un Processus, ainsi que les différents services permettant la manipulation de processus instanciés, ou la communication avec les modules clients, sont détaillés en section 3.

Les sections 4 et 5 décrivent respectivement le module de gestion des Actions synchronisées et le module de contrôle.

Comme dans le chapitre précédent, la description du prototype Ada est illustrée d'extraits, regroupés dans l'annexe B.

2. Réalisation du module de gestion des Ressources

Pour le gestionnaire des Ressources, les services définis dans le Chapitre IV se répartissent dans les trois catégories suivantes :

- les services liés à la manipulation d'une condition (création, destruction);
- les services liés à la manipulation du marquage consommé, après évaluation réussie d'une précondition ressource;
- les requêtes.

Dans le Chapitre IV, la manipulation du résultat du marquage consommé dans une précondition est "résumée" en une seule primitive : `MANIPULER_RES_PRECOND`. Plusieurs services sont cependant liés à cette fonctionnalité. Certains sont utilisés par les modules clients, afin de consulter le marquage consommé; ils figurent dans le paquetage écran. D'autres sont dédiés au gestionnaire de Ressources lorsqu'il construit le résultat. Cette structure est également utilisée pour gérer les résultats intermédiaires, dans les requêtes complexes.

C'est pourquoi le module de gestion des Ressources est divisé en trois parties (Figure VI.1), chacune d'elles correspondant à une catégorie de services.

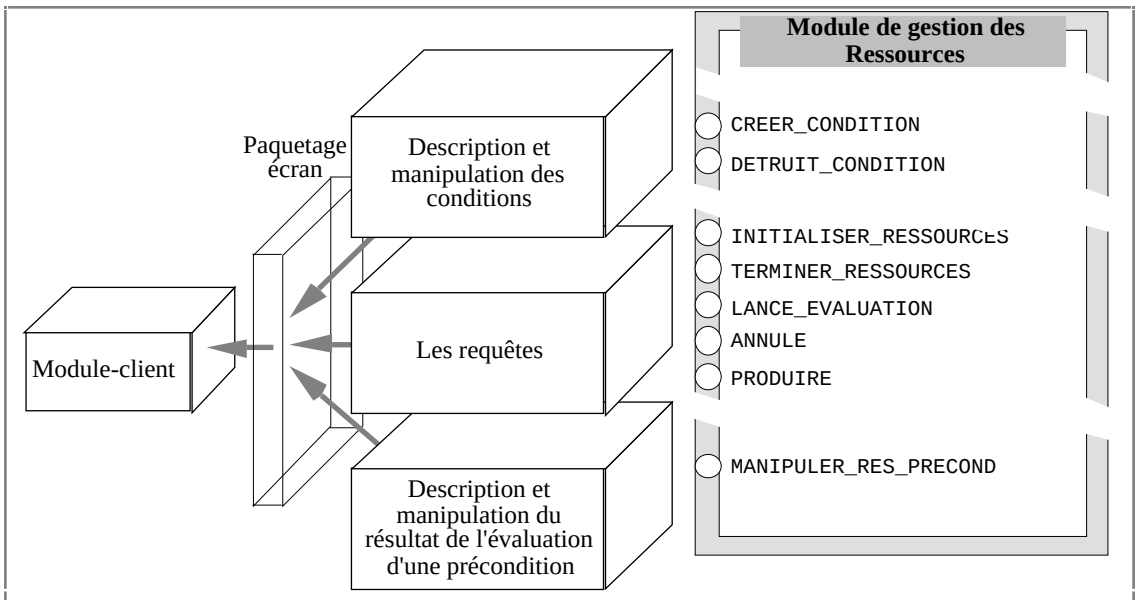


Figure VI.1 : Organisation du module de gestion des Ressources et répartition des services offerts définis dans le Chapitre IV.

2.1. DESCRIPTION D'UNE CONDITION

Les conditions (Définition IV.34), conformément à nos hypothèses, sont définies par un arbre. Ici, les nœuds sont de trois types : description d'un opérateur, description d'une marque, désignation d'une Ressource. Les nœuds modificateurs sont intégrés, s'il y a lieu, dans la désignation d'une Ressource. En Ada, les arbres sont implémentés selon une représentation fils-frère. L'arité variable des nœuds rend cette représentation pertinente.

Les Ressources externes en sortie associées aux composants externes non-réversibles ne peuvent être référencées plus d'une fois dans une précondition (Définition III.5). Pour des raisons d'implémentation et de symétrie, la restriction à été étendue aux Ressources externes en entrée. Ainsi, les Ressources externes liées aux composants externes non-réversibles sont référencées dans un champ dédié et non dans l'arbre décrivant la condition. Cela permet de les considérer au dernier moment, lorsque les autres composantes de la condition ont été traitées.

Un type polymorphe décrit un nœud de l'arbre de description des préconditions ressource. Il peut prendre trois formes :

- Description d'un nœud opération (OU, ET) ou comparaison. Dans le dernier cas, un champ permet alors de connaître l'opérateur correspondant (égal, inférieur, supérieur...).
- Description d'une marque : il peut s'agir de la constante "n'importe quelle marque".
- Désignation d'une Ressource : une opération de comparaison peut porter sur un champ des marques, si elles sont composées. Il est donc possible de référencer des composantes de marques : cela est effectué au niveau de la désignation d'une Ressource. La position[11] de la composante est désignée dans un champ du nœud; par convention, 0 signifie : "la marque entière". Si un modificateur est appliqué aux marques contenues dans la Ressource, deux champs indiquent le type du modificateur (successeur, prédécesseur) et son rang.

Un type article décrit la condition : il est composé de quatre champs :

- un booléen indiquant si une Ressource externe associée à un composant externe non-réversible est référencée;
- la référence vers la Ressource externe : ce champ n'est considéré que si le booléen est positionné;
- le marquage associé à la Ressource externe, si elle est référencée : ce champ n'a de sens que si le booléen est positionné et si la condition correspond à une postcondition;
- un pointeur vers la racine de l'arbre décrivant la condition.

Si aucun composant externe non-réversible n'est référencé dans le modèle, l'article se réduit à une référence sur la racine de l'arbre décrivant la condition.

Nous donnons dans Code 9 (Annexe B) un exemple de spécification du paquetage de gestion des conditions.

---

11 Actuellement, conformément à l'hypothèse 4 (règles de manipulation des marques) donnée dans le Chapitre VIII, section 3.2, l'accès à une composante de marque n'est possible qu'à un niveau de profondeur.

2.2. Description du résultat d'une précondition

Le résultat de l'évaluation d'une précondition est constitué d'un *ensemble cohérent de marques*, c'est-à-dire respectant les contraintes définies dans le prédicat associé à une Action. Un tel résultat est utilisé :

- Par le client, qui peut ainsi récupérer l'ensemble de marques correspondant à la précondition dont il a demandé l'évaluation. Dans ce cas, il n'y a toujours qu'un seul ensemble de marques.
- Par le module de gestion des Ressources, pour stocker les résultats intermédiaires des requêtes complexes décomposées en opérations élémentaires [Gardarin 86]. Il peut y avoir plusieurs ensembles de marques car, pour pouvoir résoudre une précondition, le gestionnaire des Ressources doit évaluer toutes les possibilités.

Le résultat de l'évaluation d'une précondition est représenté sous la forme d'ensembles cohérents de marques chaînés entre eux (Figure VI.2). Il existe une représentation particulière signifiant : "ensemble vide". Elle permet à un client de savoir si la précondition dont il a demandé l'évaluation s'est réalisée ou non.

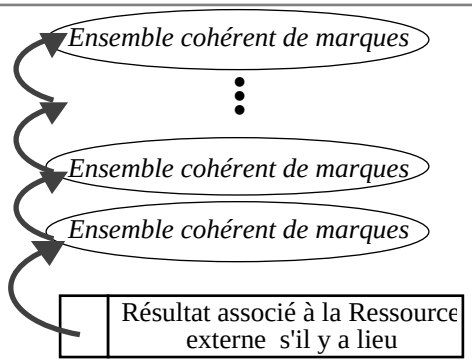


Figure VI.2 : Structure décrivant le résultat de l'évaluation d'une précondition.

Comme dans la description d'une condition, les Ressources externes liées aux composants externes non-réversibles sont considérées à part : leur traitement implique un mécanisme complètement différent.

Nous donnons dans Code 10 un exemple de spécification de l'unité de gestion du résultat d'une consommation.

2.3. RÉALISATION DES REQUÊTES

Dans un premier temps, nous détaillons la structure de données permettant de décrire l'ensemble des Ressources du modèle. Ensuite, nous détaillons le fonctionnement du serveur de Ressources.

2.3.1. Description des Ressources

Il existe un type énumératif décrivant l'ensemble des Ressources. De ce type, sont déduits deux sous-types : le premier regroupe les Ressources internes,

l'autre les Ressources externes. Si l'une des catégories de Ressources n'est pas présente dans le modèle, le sous-type correspondant n'existe pas.

Les Ressources du modèle sont décrites dans un tableau. Chaque entrée du tableau permet de connaître :

- le domaine de couleurs,
- le marquage courant, pour les Ressources internes.

Nous donnons dans Code 11 un exemple de paquetage de description des Ressources.

2.3.2. Traitement des requêtes

Le traitement des requêtes, en particulier l'évaluation d'une précondition, est une opération complexe. La gestion des Ressources est assimilable à celle d'une base de données.

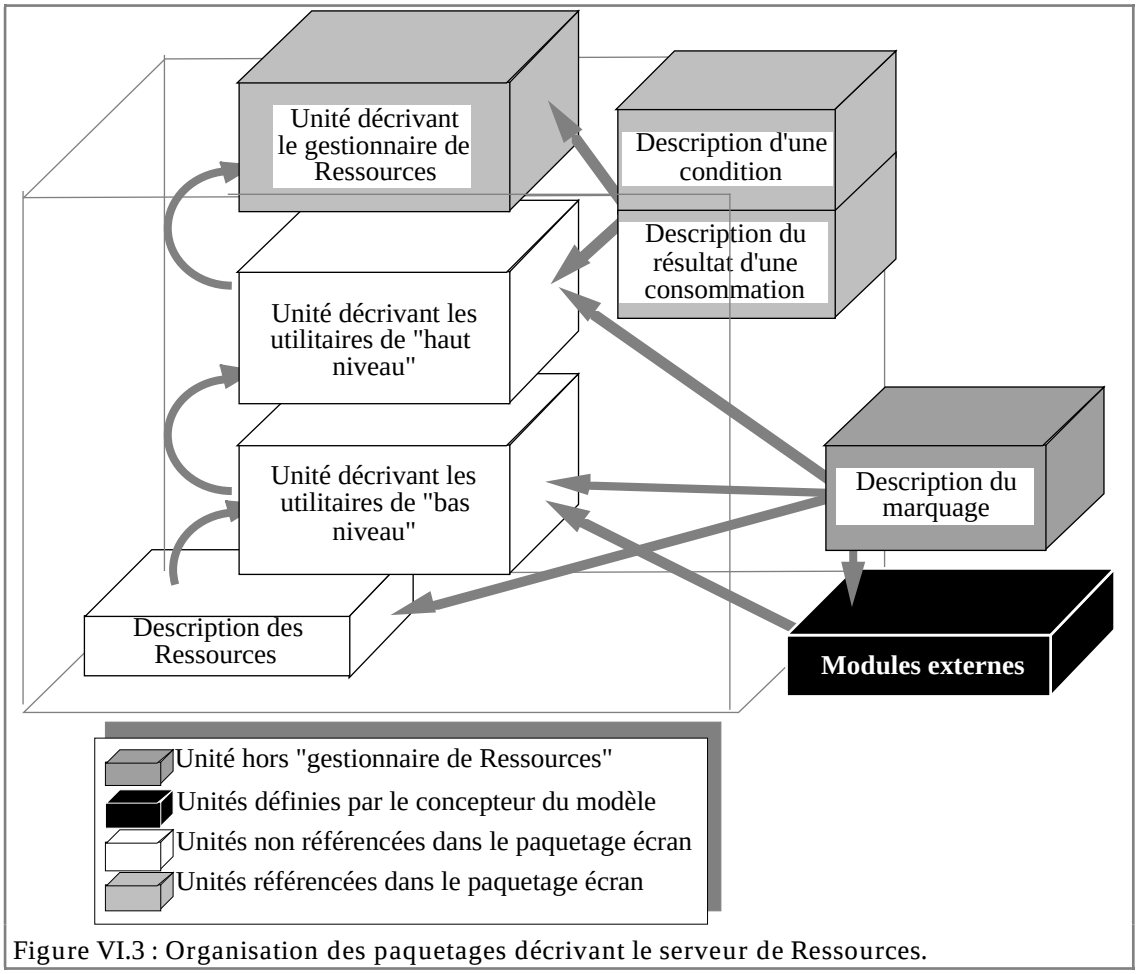


Figure VI.3 : Organisation des paquetages décrivant le serveur de Ressources.

Nous avons décomposé le système en trois unités hiérarchiques réalisant chacune des fonctionnalités différentes (Figure VI.3) :

- Le gestionnaire des Ressources gère la réalisation des services :
  - suspension des préconditions ressource dont l'évaluation a échoué;
  - annulation de demandes d'évaluation;
  - réalisation de postconditions ressource;

- reprise des évaluations en attente d'un événement positif.

La gestion des requêtes est assurée par le niveau inférieur.

- L'unité des "utilitaires de haut niveau" réalise des requêtes à partir de conditions. Ces requêtes sont décomposées en opérations [Gardarin 86], traitées par le niveau inférieur.
- L'unité des "utilitaires de bas niveau" regroupe l'ensemble des opérations élémentaires réalisables sur une Ressource interne ou externe :
  - production d'un marquage;
  - consommation d'un marquage quelconque ou ayant un profil donné;
  - appel des primitives d'interface liées aux ressources externes.

Le gestionnaire des Ressources

Le gestionnaire des Ressources est implémenté au moyen d'une tâche Ada. Les primitives associées à chaque service correspondent à des points d'entrée de cette tâche.

Les primitives réalisant les requêtes des modules-clients sont les suivantes :

- `PRODUIRE` : elle correspond à la production d'un marquage postcondition et réalise l'appel de la primitive de haut niveau correspondante. Après avoir rendu la main au client, le serveur tente de reprendre l'évaluation des préconditions en attente d'événements positifs sur des ressources appartenant à la postcondition qui vient d'être traitée.
- `LANCE_EVALUATION` : elle correspond à une demande d'évaluation d'une précondition. Le serveur tente une première évaluation. Si elle est positive, il rend un résultat immédiat (ensemble de marques non vide). Dans le cas contraire, le client se bloque en attente d'une réponse ou de la fin d'une temporisation. Les demandes mises en attente sont estampillées afin :
  - de les ordonner (éviter la famine);
  - de les identifier en cas d'annulation.
- `ANNULE` : le client transmet l'estampille de sa demande, et reçoit une réponse dans un paramètre en sortie. Le gestionnaire des Ressources ne refuse l'annulation d'une évaluation que si la précondition a déjà été évaluée positivement : ce cas de figure correspond à un croisement entre le réveil du client et la demande d'annulation.
- `INITIALISER_RESS` : elle est utilisée pour initialiser le module de contrôle. Si les ressources du modèle possèdent un marquage initial, une condition le décrivant est requise en paramètre pour ce service.
- `TERMINER_RESS` : elle est utilisée par le module de contrôle pour mettre fin à l'exécution du serveur.

Un exemple de spécification du paquetage décrivant le gestionnaire des Ressources est donné dans Code 12.

L'évaluateur de requêtes

Il existe deux types de requêtes : la production d'un marquage postcondition et l'évaluation d'une précondition.

La production d'un marquage postcondition est réalisée de la façon suivante : l'arbre décrivant la postcondition est parcouru et chaque composante est réalisée l'une après l'autre, à l'aide de la primitive de production unitaire.

L'évaluation d'une précondition est confiée à un évaluateur qui parcourt l'arbre de condition en construisant progressivement tous les ensembles cohérents de marques possibles. Les marques permettant la construction des ensembles cohérents ne sont pas consommées. Des optimisations sont réalisées au niveau des opérateurs ET et OU :

- si aucune marque ne correspond à un argument d'un nœud ET, l'évaluation est immédiatement arrêtée et un résultat vide rendu;
- dès que l'un des arguments d'un nœud OU restitue un résultat non vide, l'évaluation des arguments suivants est stoppée.

La Ressource externe référencée dans la précondition est traitée, s'il y a lieu, lorsque l'arbre décrivant la précondition ressource a été évalué positivement. Si le module externe "fournit une marque", il y a consommation effective de l'un des ensembles cohérents de marques calculés. Cet ensemble est alors transmis comme résultat au client.

En cas d'échec de l'évaluation d'une précondition, l'évaluateur transmet un diagnostic. L'échec peut être lié :

- A un défaut de marque dans une Ressource interne ou externe (échec d'une restriction). Dans ce cas, seul un événement positif sur cette Ressource provoque le réveil de la précondition.
- A l'absence d'un ensemble de marques respectant les contraintes définies dans le prédicat (échec d'une jointure). Dans ce cas, tout événement positif pour une Ressource référencée dans la précondition provoque une nouvelle évaluation.

Un exemple de spécification du paquetage d'évaluation des requêtes est donné dans Code 13.

Les opérations élémentaires sur les Ressources

Elles permettent d'évaluer ou de modifier le marquage d'une Ressource. Il est ainsi possible :

- De produire une marque ou plusieurs marques d'un profil donné : si la Ressource est interne, la structure de données décrivant son marquage est mise à jour. Sinon, la primitive correspondante est appelée.
- De consommer une ou plusieurs marques d'un profil donné : selon que la Ressource est interne ou externe, il y a modification de la description de son marquage ou appel de la primitive associée.

- De connaître l'ensemble des marques contenues dans la Ressource qui respectent un profil donné : cette primitive n'est pas disponible pour les Ressources externes liées à des composants externes non-réversibles.

Un exemple de spécification du paquetage réalisant les requêtes unitaires est donné dans Code 14.

3. Réalisation des modules-processus

Pour les modules-processus, les services définis dans le Chapitre IV se répartissent dans les deux catégories suivantes (Figure VI.4) :

- la définition du mot d'état : si le Processus comporte un mot d'état, une unité dédiée décrit la structure associée ainsi que les primitives de manipulation;
- la description du processus : elle définit le modèle de tâche décrivant le processus ainsi que les primitives permettant de manipuler des processus instanciés.

Dans le Chapitre IV, la manipulation du mot d'état est "résumée" en une seule primitive : `MANIPULER_MOT_D_ETAT`. Plusieurs services sont cependant liés à cette fonctionnalité.

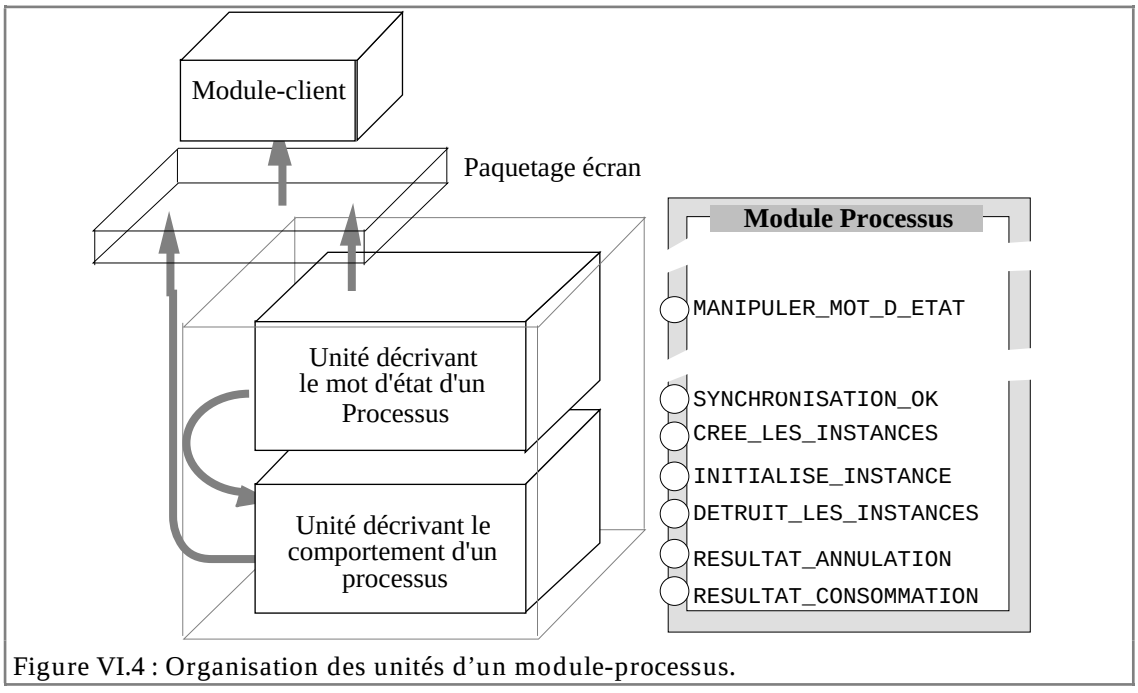


Figure VI.4 : Organisation des unités d'un module-processus.

Les primitives offertes par l'unité décrivant un mot d'état sont utilisées :

- par le paquetage décrivant le comportement d'un Processus;
- par le gestionnaire des Actions synchronisées, qui peut avoir à manipuler les mots d'état des participants;
- par le module de contrôle, durant la phase d'initialisation.

Description du mot d'état

Le mot d'état d'un processus est réalisé à l'aide d'un type article regroupant l'ensemble des composantes du mot d'état. L'accès à ces composantes est effectué au moyen de primitives manipulant des marques.

Nous donnons dans Code 15 un exemple de spécification de l'unité définissant la manipulation du mot d'état d'un Processus.

Décomposition de la description d'un Processus

Pour une meilleure lisibilité, nous avons décomposé le comportement d'un Processus en trois niveaux (Figure VI.5) :

- Le niveau comportemental : il décrit le modèle de tâche ainsi que les primitives permettant de créer, initialiser et détruire les processus instanciés déduits de ce modèle.
- Le niveau état actif : il décrit les traitements associés à chaque état actif du Processus. Deux paquetages composent ce niveau, le premier réalise les traitements associés aux Etats_Processus alternatifs, l'autre ceux associés aux Actions.
- Le niveau utilitaire : il s'agit de traitements de "bas niveau", comme la construction d'une condition (pré ou post) ou la fonction de tirage de choix du successeur d'un Etat_Processus alternatif.

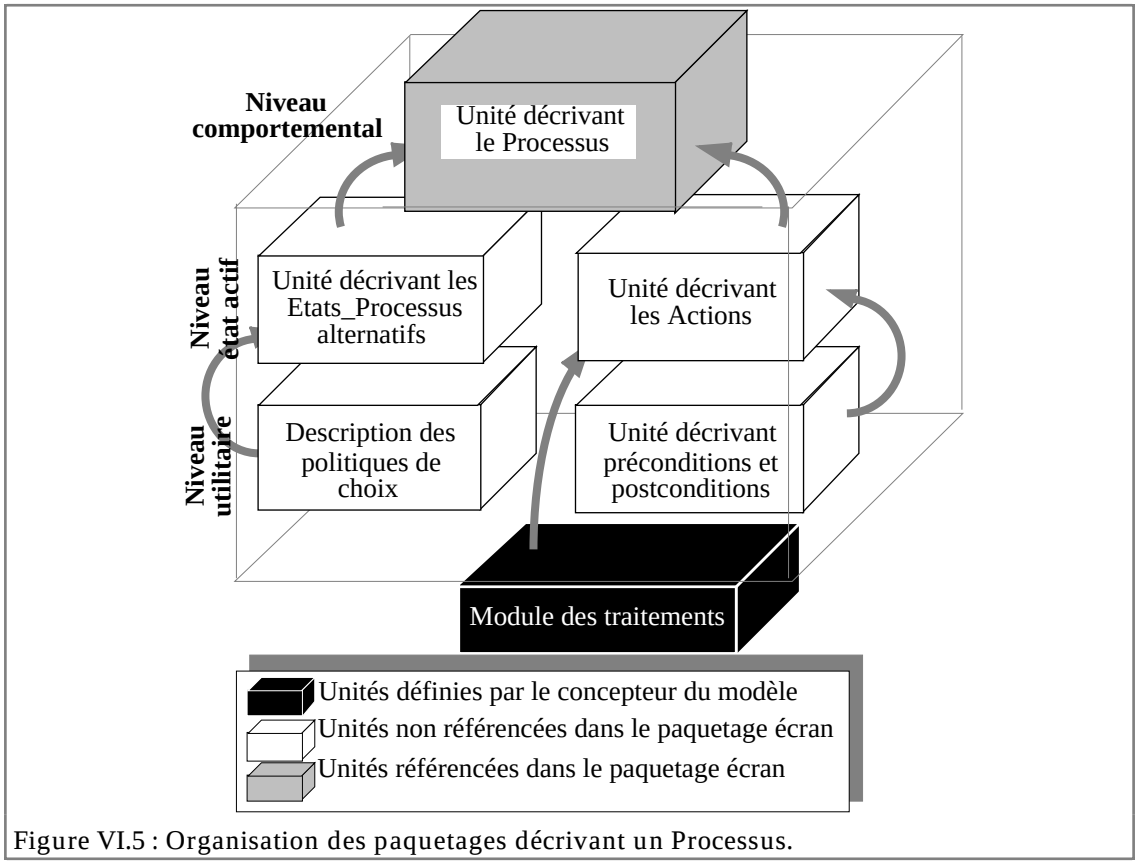


Figure VI.5 : Organisation des paquetages décrivant un Processus.

Le niveau comportemental

Les services qu'il offre sont ceux définis dans le Chapitre IV :

- création, initialisation et destruction des processus instanciés, dédiés au module de contrôle;
- résultat d'une synchronisation, pour le gestionnaire des synchronisations;
- résultat de l'évaluation d'une précondition, pour le gestionnaire des Ressources.

Le service `RESULTAT_ANNULATION` n'est pas explicitement fourni, l'acceptation d'une annulation est notifiée dans le paramètre en sortie des primitives offertes par les gestionnaires des synchronisations et des Ressources.

Nous donnons dans Code 16 un exemple de spécification d'un paquetage décrivant le niveau comportemental. Code 17 contient un exemple de modèle de tâche associé à un Processus.

Le niveau Etat Actif

Le paquetage regroupant les Actions du modèle contient une procédure par Action. Chacune d'elles possède les paramètres suivants :

- le numéro d'instance : il permet de transmettre l'identité d'un processus instancié aux modules de service;
- le mot d'état du processus instancié, s'il existe : ce paramètre est modifié si l'Action a une influence sur les valeurs contenues dans le mot d'état;
- l'état actif en sortie de la primitive;
- un booléen indiquant si le processus instancié doit se terminer ou non (cas d'une Action précondition d'un Etat_Processus de terminaison).

Le paquetage regroupant les Etats_Processus alternatifs contient une fonction par Action. La valeur rendue correspond à l'état actif choisi. Les paramètres sont :

- la stratégie (tourniquet ou aléatoire);
- une graine, gérée par chaque processus instancié.

Si le tirage est effectué selon une politique aléatoire, la graine permet de créer une nouvelle valeur. Dans le cas du tourniquet, elle correspond au "numéro" de successeur dernièrement choisi.

Nous donnons, dans Code 18 et Code 19, la spécification des unités décrivant respectivement les Etats_Processus et les Actions d'un Processus. Code 20 contient le corps d'une primitive associée à un Etat_Processus alternatif et Code 21 celui d'une procédure réalisant une Action.

Le niveau utilitaire

L'unité de service, offerte pour le choix d'une Action dans la postcondition d'un Etat_Processus alternatif, n'est pas dépendante d'un modèle. La même unité est utilisée pour l'ensemble des Processus composant un prototype.

Il existe, dans le paquetage construisant les conditions, une primitive par précondition ressource et par postcondition ressource contenue dans le Processus. Le mot d'état du Processus est transmis, dans le cas où une sélection (Définition IV.30) est liée au contenu d'une composante du mot d'état.

4. Réalisation du module de gestion des Actions SYNCHRONISÉES

Pour une meilleure lisibilité, nous avons associé un paquetage à chaque serveur d'Action synchronisée (Figure VI.6). Les primitives sont renommées dans le paquetage écran.

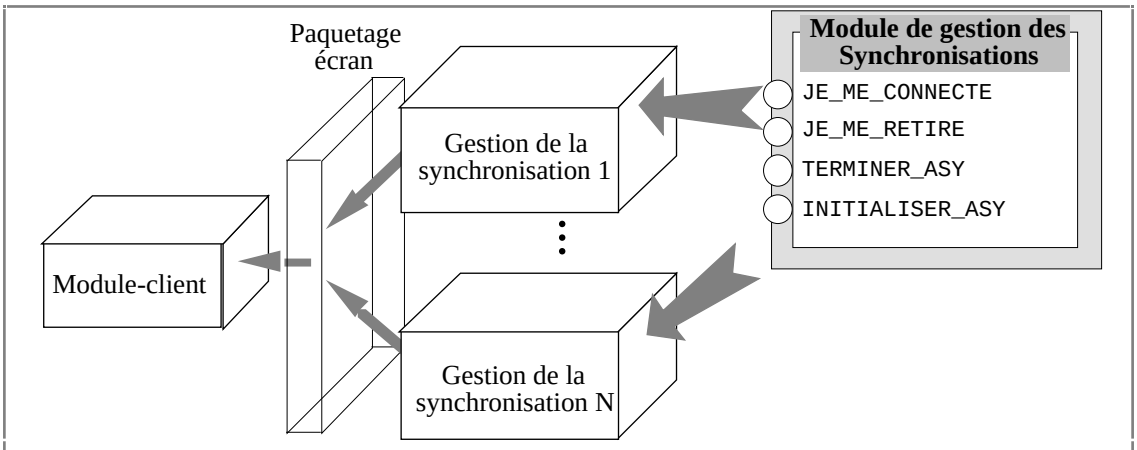


Figure VI.6 : Décomposition en paquetages du gestionnaire des Actions synchronisées.

Deux problèmes sont à envisager : la gestion du contexte lié à une Action synchronisée et la recherche d'une instance.

Le contexte d'une Action synchronisée est réalisé de la manière suivante : une liste par Processus participant à l'Action synchronisée permet de caractériser les processus instanciés en attente (identité, mot d'état).

Une fonction d'évaluation tente de construire une instance composée de participants qui respectent les contraintes définies dans les préconditions statiques et synchrones.

Lorsqu'une instance a été trouvée, le serveur d'une Action synchronisée crée une tâche de service chargée de réaliser l'appel de la procédure associée à l'Action, de modifier, le cas échéant, les mots d'état des participants (marquage postcondition) puis de les réveiller. Une fois que la tâche de service a été lancée, le serveur refuse toute demande d'annulation correspondant à l'instance de synchronisation qu'elle réalise.

Chaque serveur d'une Action synchronisée est réalisé au moyen d'une tâche Ada. Les primitives d'un paquetage de gestion d'une synchronisation font directement appel aux points d'entrée de cette tâche.

Les primitives réalisant les requêtes des modules-processus sont les suivantes :

- `INITIALISER_ASY`, utilisée par le module de contrôle pour initialiser le serveur, au démarrage du prototype.
- `TERMINER_ASY` permet au module de contrôle de l'application de provoquer la terminaison du serveur de synchronisations. Si le prototype se termine normalement, il ne doit pas y avoir de tâches de service en fonctionnement. Dans le cas contraire, la terminaison des tâches de

service est gérée par l'exécutable : l'exception `TASKING_ERROR` sera levée lorsqu'elles tenteront de réveiller un client inexistant.

- `JE_ME_CONNECTE` : ce service est dupliqué en fonction du nombre de clients. Chaque client transmet, via cette primitive :
  - son numéro d'instance,
  - son mot d'état s'il existe.
- `JE_ME_RETIRE` : ce point d'entrée n'existe que si, pour une Action synchronisée, il existe au moins un participant pouvant effectuer un retrait.

Nous donnons dans Code 22 un exemple de spécification d'un paquetage de gestion d'une Action synchronisée.

5. Réalisation du module de contrôle

Le contrôle de l'exécution du prototype est défini dans un paquetage dédié. Le corps de cette unité voit[12] toutes les unités du prototype, c'est-à-dire :

- les paquetages écran des modules-processus et des modules de service;
- le paquetage de gestion des communications;
- les paquetages de gestion des mécanismes de trace.

L'initialisation et la surveillance du fonctionnement de l'application sont effectuées par un gardien unique (section 5.4.1 du Chapitre IV). Ce gardien est initialisé au démarrage de l'application par le programme principal dont c'est la seule fonction.

Pour des raisons d'implémentation, les services `PROBLEME_EXTERNE` et `JE_ME_TERMINE`, définis dans le Chapitre IV, ont été regroupés en une seule primitive ayant deux paramètres :

- le type de terminaison (normale ou sur un problème externe);
- l'identité de la tâche qui effectue l'appel : cette identité n'est significative que si la terminaison est normale car, dans ce cas, il s'agit forcément d'un processus instancié et le contexte doit être mis à jour.

Le contexte d'exécution du prototype est défini par le nombre de processus instanciés en cours d'exécution. Il existe un compteur par Processus du modèle : il est incrémenté à chaque création de processus instancié et décrémenté à chaque fois que le gardien reçoit une notification de fin d'exécution.

L'utilisation de plusieurs compteurs permet d'indiquer, en cas de terminaison sur erreur, de quels types sont les processus instanciés encore actifs dans le prototype.

Lorsque tous les compteurs sont nuls, le serveur termine les modules de service avant d'achever son exécution.

Nous donnons dans Code 23 un exemple de spécification du paquetage réalisant un module de contrôle. Code 24 contient un exemple de programme principal.

[12] Au sens Ada du terme, tel qu'il est défini par la clause "with"

6. Conclusion

Nous avons réalisé un outil, CPN/TAGADA, qui génère un prototype Ada centralisé selon les règles définies dans ce chapitre. Il a été éprouvé sur de nombreux modèles.

Comme il est inutile de générer des sections de programme qui ne seront jamais exécutées, ou de mettre en œuvre des mécanismes pouvant être simplifiés, des optimisations ont été définies :

- Si le modèle ne comporte pas de Ressources, le module de gestion des Ressources n'est pas généré. Par ailleurs, s'il ne comporte que des Ressources externes, ou que des Ressources internes, seuls les traitements correspondants sont effectifs.
- Si le modèle ne comporte pas d'Action synchronisée, le gestionnaire des synchronisations n'est pas généré.
- Le code définissant le modèle de traitement associé aux processus du réseau de Petri est optimisé, suivant les principes énoncés au Chapitre IV, sur la base des attributs définissant les objets, et de leurs positions les uns par rapport aux autres.

D'autres optimisations, d'un niveau plus fin, sont envisageables. Il serait par exemple intéressant de simplifier au maximum l'évaluateur de précondition, en n'introduisant que les traitements associés aux fonctions réellement présentes dans le modèle. Par exemple, si aucun prédicat d'un réseau de Petri ne fait référence à l'opérateur logique OU, les traitements et les descriptions associés n'ont pas à figurer dans le prototype.

			Introduction générale
	Chapitre I	:	*Prototypage de systèmes parallèles*
	Chapitre II	:	*Interprétation sémantique d'un réseau de Petri*
	Chapitre III	:	*Méthodologie de prototypage*
	Chapitre IV	:	*Structure du prototype généré*
	Chapitre V	:	*Caractéristiques du prototype Ada*
	Chapitre VI	:	*Le prototype centralisé*
+	Chapitre VII	:	*Le prototype réparti*
	Chapitre VIII	:	*CPN/TAGADA : L'outil de prototypage*
			Conclusion générale
			Bibliographie
			Annexes & Appendices

Chapitre VII :

Le prototype réparti

1. Introduction 239
2. Les services de répartition 242
   - 2.1. Choix du niveau de répartition 242
   - 2.2. Traitement des unités réparties 243
   - 2.3. Mise en œuvre de la répartition....245
   - 2.4. Principes de conception....247
   - 2.5. Définition d'une boîte à outils de répartition....252
3. Réalisation d'une application répartie 254
   - 3.1. Principes de réalisation....254
   - 3.2. Architecture centralisée 256
   - 3.3. Architecture décentralisée....261
4. Application au prototypage 266
   - 4.1. Choix stratégiques 266
   - 4.2. Première approche du prototype réparti 267
   - 4.3. Seconde approche du prototype réparti....270
5. Observations sur le placement 282
6. Expérimentation 287
   - 6.1. Le modèle à prototyper....287
   - 6.2. Les mesures....290
   - 6.3. Quelques remarques 292
7. Conclusion 294

1. Introduction

Les systèmes informatiques répartis permettent de développer des applications plus performantes qui profitent simultanément de la puissance de calcul de plusieurs sites. C'est la raison pour laquelle, dans un tel contexte, les entités fonctionnelles (les tâches) sont réparties dans des programmes distants.

Malheureusement, pour une telle réalisation, les compilateurs Ada actuels ne sont généralement pas adaptés : bien que le langage autorise le parallélisme, ils ne permettent pas de créer un programme qui puisse s'exécuter sur plusieurs machines. On considère principalement deux types de compilateurs Ada :

- Les compilateurs produisant un exécutable destiné à fonctionner sur une carte nue. Dans ce cas, si l'architecture est parallèle, le concepteur de l'application peut effectuer l'édition de liens de son programme avec un "runtime" capable d'offrir des services de répartition.
- Les compilateurs produisant un exécutable qui fonctionne sur un système d'exploitation. Dans la grande majorité des cas[13], ils produisent un seul programme, quel que soit le nombre de tâches Ada contenues dans l'application. Seul un pseudo-parallélisme est donc possible.

C'est en particulier le cas pour UNIX™, le système d'exploitation utilisé dans le cadre de nos travaux. Récemment, dans l'attente d'une solution normalisée (Ada9x), certaines études ont porté sur la possibilité de répartir des programmes Ada sous UNIX™ [Atkinson 88, Heitz 91, Bazalgette 91, Kordon 91c, Millard 91].

Nous avons tout d'abord développé des techniques, valables pour les applications Ada, dans le but de les appliquer au prototype que nous générons. Il s'agit d'une méthode de conception d'applications Ada réparties sur plusieurs machines faiblement couplées. Ces applications sont découpées en groupes de tâches Ada que nous répartissons sur différents sites. Une extension de cette méthode permet de transformer un programme Ada parallèle déjà existant afin de répartir son exécution sur plusieurs machines.

Les travaux que nous avons effectués reposent fortement sur UNIX™. Cependant, les principes sont parfaitement utilisables pour tout système d'exploitation offrant des mécanismes de communication similaires.

Notre méthode repose sur la définition de services de répartition transparents du point de vue de l'application (Figure VII.1). Ces services constituent le "niveau bas" de l'application. Ainsi, l'architecture logique n'est pas à reconsidérer : elle s'appuie sur des mécanismes différents. L'application originelle est découpée en sous-ensembles s'exécutant sur des processeurs différents via les services de répartition.

---

[13] Quelques constructeurs de compilateurs offrent, pour certains systèmes, la possibilité d'associer une tâche Ada à une tâche du système.

La granularité du découpage de l'application est un premier problème, en particulier dans un langage qui, comme Ada, permet la manipulation d'unités de programme (paquetages) et de tâches [Heitz 91, Kordon 91c].

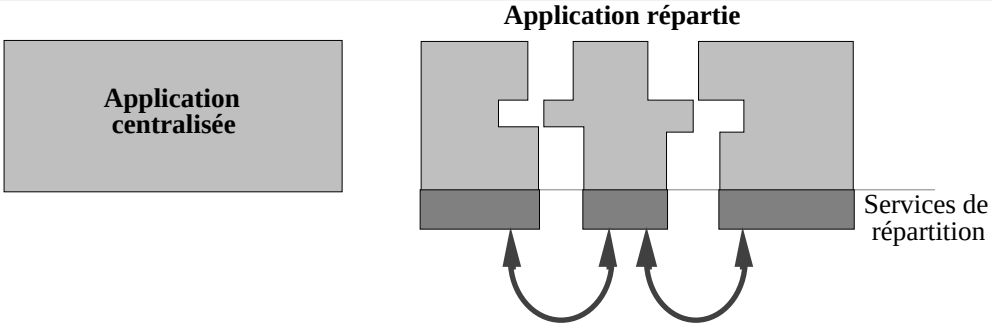


Figure VII.1 : Principes généraux de répartition.

Par ailleurs, les différentes parties de l'application originelle doivent coopérer. L'initialisation et la terminaison de l'application constituent des problèmes classiques de l'informatique répartie qu'il nous faut considérer [Raynal 88, Blanc 90].

Il est intéressant de pouvoir changer le placement des différents composants du prototype sans avoir à le modifier explicitement, ou même à le recompiler. Cela motive l'objectif suivant.

***Objectif 1 :* Placement dynamique**

> La description du placement des différentes parties du prototype doit être externe. Elle est effectuée à l'aide d'un fichier de configuration.

Par ailleurs, nous souhaitons récupérer le maximum d'éléments du prototype centralisé.

***Objectif 2 :* Transparence d'utilisation**

> L'utilisation des mécanismes liés à la répartition doit être la plus transparente possible afin de limiter les différences entre le prototype réparti et non réparti.

Dans un premier temps, nous définissons nos principes de répartition d'une application réalisée en langage Ada. Ces principes sont valables sur tout système d'exploitation permettant la communication entre machines distantes.

Ensuite, nous détaillons plusieurs techniques de réalisation des principes énoncés. Ces techniques mettent en œuvre des architectures différentes, basées sur des stratégies de contrôle de l'application opposées (centralisé et décentralisé).

Dans le Chapitre VI, nous avons décrit la structure d'un prototype centralisé multi-tâches. Nous traitons désormais de l'application des principes de répartition au prototype. Un problème majeur est posé par le parallélisme dans la manipulation des Ressources, ce qui nous amène à envisager plusieurs approches pour la réalisation du gestionnaire de Ressources.

Enfin, nous présentons des observations sur les techniques de placement des différentes composantes du prototype.

2. Les services de répartition

Dans le système UNIX™ (mais ce n'est pas le seul exemple), un programme Ada, même si l'utilisateur en a une perception multi-tâches, est supporté par un seul processus. Notre méthode repose sur un service de répartition des fonctionnalités du système sur plusieurs programmes exécutables. Ces derniers s'exécuteront en parallèlisme effectif.

2.1. CHOIX DU NIVEAU DE RÉPARTITION

En Ada, les techniques évoquées reposent sur les principes suivants [Heitz 91, Bazalgette 91, Kordon 91c, Millard 91] : l'application est divisée en granules intégrés dans différents exécutables (Figure VII.2). Ces exécutables sont lancés à distance sur différents processeurs interconnectés. Un service de répartition assure, de façon plus ou moins transparente, la communication entre les différentes entités composant l'application.

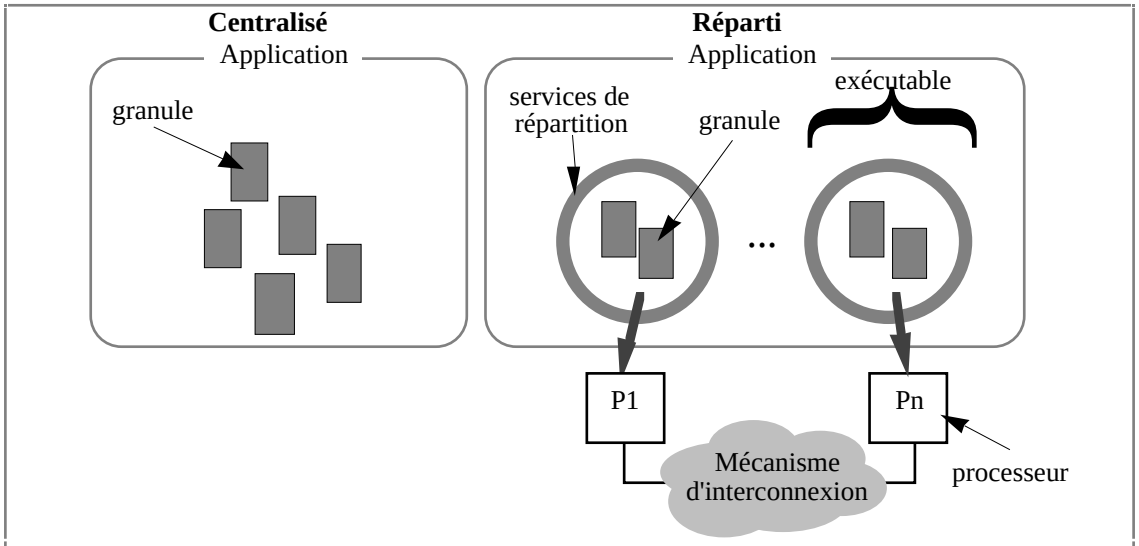


Figure VII.2 : Principe de décomposition d'une application Ada.

Deux niveaux de granularité peuvent être envisagés en Ada : le paquetage [Heitz 91] ou la tâche [Bazalgette 91, Kordon 91c, Millard 91].

Le niveau "paquetage"

Les paquetages composant l'application sont répartis dans différents exécutables dotés d'un mécanisme de communication permettant un dialogue.

Cette technique est très utilisée avec les langages classiques n'intégrant aucune gestion du parallélisme comme C ou Pascal. A l'heure actuelle, les services de répartition se trouvent d'ailleurs intégrés au niveau du système d'exploitation [Stevens 90].

L'option choisie dans [Heitz 91] utilise ce niveau de granularité. Des mécanismes (notion de "faux body") permettent d'assurer la transparence dans la conception de l'application.

Pour être appliquée au prototypage, cette méthode présente l'inconvénient de rendre difficile toute modification de la répartition des travaux à effectuer par chaque programme composant l'application. De plus, il n'y a, a priori, aucune symétrie [Burns 81], ce qui peut nuire, si l'on souhaite intégrer des mécanismes de reprise sur erreur [Avizienis 87, Nelson 90].

Le niveau "tâche"

Les tâches composant l'application sont réparties dans différents exécutables dotés d'un mécanisme de communication permettant un dialogue. Cette technique ne peut être utilisée qu'avec un langage qui, comme Ada, permet la manipulation de tels objets.

Les tâches, en Ada, sont associées à une unité maître [Ada 83]. Cela oblige à positionner les services de répartition au niveau d'une tâche, afin de préserver la transparence du système.

L'avantage de ce niveau de répartition est qu'il n'affecte pas la conception de l'application au niveau du langage Ada. Cependant, il faut envisager une extension de la notion de rendez-vous [Bazalgette 91, Kordon 91c] afin de conserver la compatibilité avec les mécanismes décrits dans [Ada 83].

Choix du niveau de répartition

La répartition de tâches Ada est plus intéressante : il s'agit d'une entité parallèle logique. De plus, si les services de répartition sont correctement définis, la conception d'une application parallèle ne s'en trouve pas altérée.

Cela impose certaines règles de comportement, notamment en ce qui concerne l'utilisation des données partagées. Le partage direct de données entre deux processus constitue un obstacle à la parallélisation [Coffman 71] et peut être évité en utilisant des tâches de service pour assurer l'accès à ces données (sémaphores).

2.2. TRAITEMENT DES UNITÉS RÉPARTIES

Nous considérons deux façons de répartir des tâches Ada sur un ensemble de machines. Une première solution consiste à diviser l'application en autant de processus du système d'exploitation que de tâches Ada (Figure VII.3). Une autre technique consiste à créer plusieurs exécutables qui seront exécutés sur des sites différents (Figure VII.4). Nous détaillons ces deux solutions.

Solution 1 : un processus UNIX par tâche Ada

Les communications sont assurées par le service de répartition, interfacé en Ada avec des primitives du système. Chacun des exécutables se termine lorsque la tâche qu'il contient est achevée.

Pour cela, il faut réécrire les tâches Ada afin de transformer les appels de points d'entrée en appels de services.

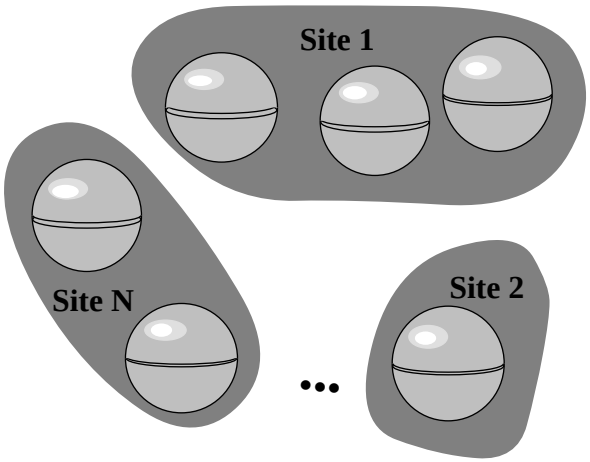


Figure VII.3 : Transformation d'un programme Ada parallèle en autant d'exécutables qu'il y a de tâches dans l'application.

Cette solution est développée dans le projet STRADA [Bazalgette 91]. Un traducteur automatique transforme le programme source de l'application de façon à l'adapter aux paquetages de service permettant la répartition.

Cette approche possède un avantage majeur : il n'est pas nécessaire de modifier l'application pour changer la répartition des tâches. Cependant, la communication entre deux tâches Ada met en œuvre des mécanismes coûteux, même si elles s'exécutent sur le même site.

Solution 2 : un processus UNIX par site

Chaque exécutable contient un ensemble de tâches Ada.

Il existe deux niveaux de communication : les communications locales, gérées uniquement en Ada, et les communication distantes, gérées par un *service de répartition*.

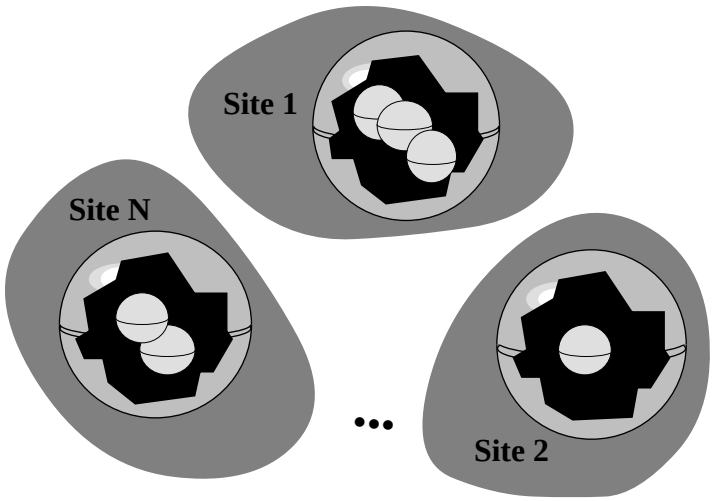


Figure VII.4 : Transformation d'un programme Ada parallèle en plusieurs exécutables. Chacun d'eux contient des tâches Ada et s'exécute sur un site différent.

En revanche, toute modification du regroupement des tâches de l'application implique des modifications dans les programmes composant l'application.

Une solution consiste à réaliser la gestion de la configuration au niveau des services de répartition.

Nous avons choisi de regrouper l'ensemble des tâches Ada s'exécutant sur une machine dans un seul exécutable :

- si les services de répartition sont bien conçus, le placement peut être effectué au lancement de l'application;
- les communications locales ne sont pas désavantagées par un mécanisme complexe.

2.3. MISE EN ŒUVRE DE LA RÉPARTITION

Le service de répartition doit, dans un premier temps, permettre la manipulation de tâches dans les conditions définies par [Ada 83], c'est-à-dire :

- réalisation d'un rendez-vous : ce service est primordial car c'est le seul mécanisme de communication entre tâches[14] disponible en Ada;
- création dynamique de tâches distantes;
- destruction d'une tâche distante.

Les services de répartition se ramènent tous à un échange de messages entre exécutables situés sur des sites différents. Chaque message véhicule une demande de service particulière.

Réalisation d'un rendez-vous distant

Soient T1 et T2 deux tâches. T1 appelle un point d'entrée de T2. La réalisation du rendez-vous Ada se déroule en trois temps :

- établissement : lorsque T1 a appelé T2 et si T2 est prête à "accepter" le point d'entrée, le rendez-vous a lieu;
- traitements : le corps du point d'entrée est réalisé pour le compte de T2, la tâche appelée; T1 reste suspendue;
- terminaison : lorsque les instructions associées au point d'entrée sont exécutées, T1 et T2 reprennent leur exécution indépendamment.

La sémantique est différente en cas d'attentes multiples ou temporisées (instruction `select`), mais le mécanisme de base que nous avons exposé reste le même.

Sa réalisation suit le principe des RPC (Remote Procedure Call) [Nelson 81, Birell 84, Weihl 89]. Nous utilisons deux messages [Bazalgette 91, Kordon 91c, Millard 91]. Un message de *connexion*, émis par T1 vers T2 et, lorsque le point d'entrée a été réalisé, un message de *déconnexion*, émis par T2 vers T1. Les paramètres en entrée du point d'entrée sont codés dans le message de connexion et les paramètres en sortie dans le message de déconnexion (Figure VII.5).

---

[14] Nous ne considérons pas la communication par variable partagée qui nécessite l'utilisation de tâches, si l'on veut assurer la protection des données.

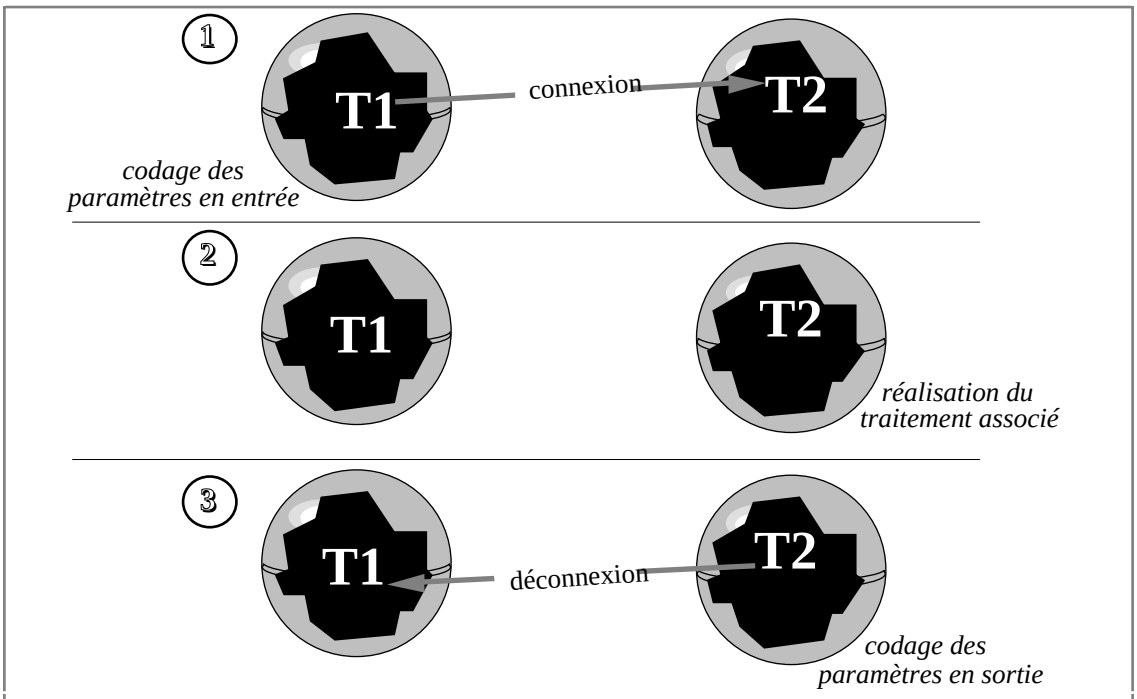


Figure VII.5 : Messages associés à la réalisation d'un rendez-vous distant.

Un problème est cependant posé par la propagation des exceptions survenant pendant l'exécution du traitement associé à un point d'entrée. [Ada 83] spécifie qu'elles sont propagées à la tâche appelée et à la tâche appelante. Lorsque cette dernière est distante, la propagation doit se faire via le message de déconnexion. Le nom de l'exception ne peut être spécifié que s'il est visible au niveau du mécanisme assurant le service de répartition, ce qui n'est en général pas le cas. Il est impossible de reproduire tel quel le comportement défini par [Ada 83] sans travailler au niveau de l'exécutable produit par un compilateur[15]. Il est cependant possible :

- de propager toutes les exceptions prédéfinies du langage nominativement, puisqu'il n'y a pas de problème de visibilité;
- de signaler l'existante d'une exception non prédéfinie. La tâche appelante saura qu'un tel événement s'est produit mais ne pourra pas identifier l'exception.

Création dynamique de tâches distantes

La création dynamique d'une tâche distante repose également sur un mécanisme de type RPC impliquant deux messages. Le premier est un ordre de création; le second permet de transmettre au demandeur de la création, via le service de répartition, l'identité de la tâche qu'il a créée (Figure VII.6).

15 C'est-à-dire en abandonnant tout souci de portabilité.

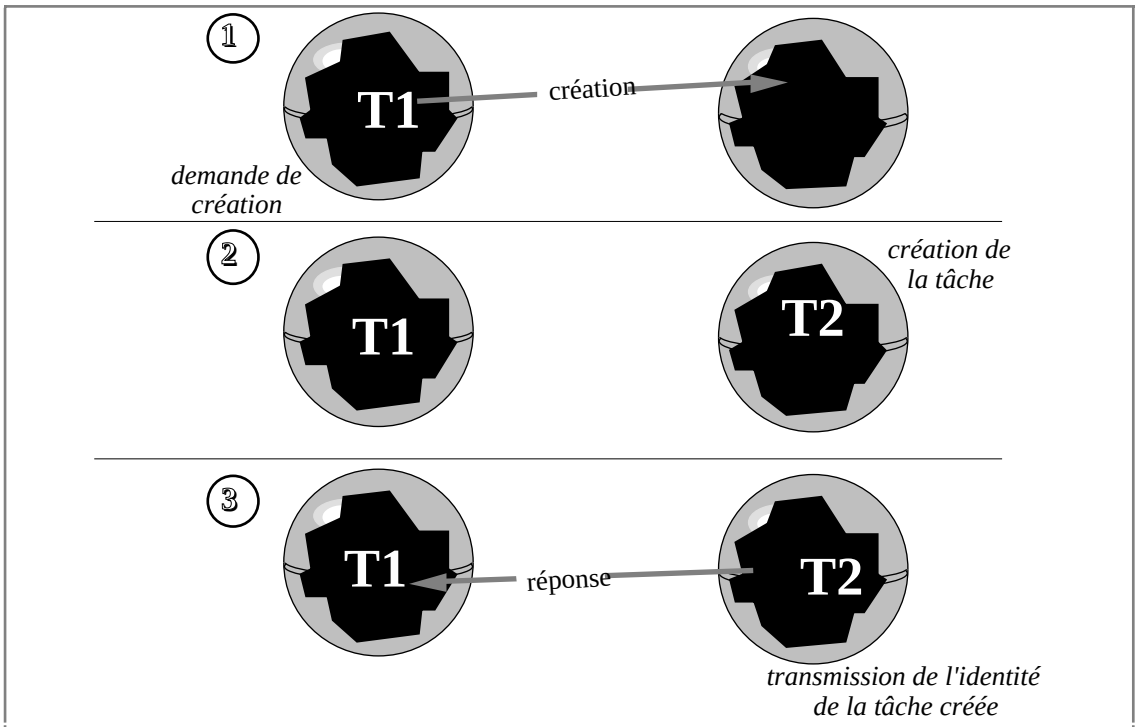


Figure VII.6 : Messages associés à la création dynamique d'une tâche distante.

Destruction d'une tâche distante

Dans ce cas, seul un message de type *abort* est nécessaire. Il doit contenir l'identité de la tâche à détruire. Les services de répartition, dans l'exécutable concerné, effectuent la destruction.

Autres services

D'autres services peuvent offrir des facilités liées à l'exécution d'une application sur une architecture multi-processeurs :

- le calcul de la charge d'un site : cela permet d'effectuer un placement dynamique, en fonction de la charge des machines hôtes au lancement de l'application [Theimer 88, Folliot 89];
- la migration d'une tâche de l'application : cela permet d'ajuster, en cours d'exécution, la charge de l'application sur les différents sites [Bernard 91, Goscinski 91].

2.4. PRINCIPES DE CONCEPTION

Une première approche consiste à remplacer les appels de points d'entrée par des appels de primitives réalisant un service équivalent [Bazalgette 91]. Cette solution ne peut être employée dans notre cas puisque les communications peuvent être locales ou distantes. Pour assurer la transparence, l'utilisation du mécanisme doit être la même dans les deux cas.

Nous introduisons trois principes de base, visant à respecter les objectifs que nous nous sommes donnés.

***Principe de conception P1 :* Un seul exécutable**

Il n'existe qu'une seule version de l'exécutable comportant le code de toutes les tâches. Cet exécutable est lancé à distance sur chacun des sites où l'application s'exécute.

Cet exécutable contient l'ensemble des tâches de l'application, plus des mécanismes permettant d'offrir un service de répartition.

Ainsi, la répartition n'est pas fixée dans le programme, elle est déterminée de façon externe. Il n'existe qu'un seul exécutable, que l'on peut lancer sur un nombre quelconque de machines. Une telle structure offre également l'avantage d'une totale symétrie [Burns 81] : tous les exécutables sont identiques. Cependant l'espace disque occupé est plus important sur les machines qui exécutent l'application.

Les mécanismes permettant la communication transparente entre deux tâches distantes reposent sur la notion de mandataires [Shapiro 86]. Un mandataire est un représentant local d'une tâche distante.

***Définition VII.1 :* Les Tâches Transparentes à la Répartition (TTR) - actives ou écran**

Chaque TTR englobe le corps d'une tâche préexistante de l'application et le code du mandataire. Elle possède deux modes (Figure VII.7) :

- une TTR est dite *Active* si elle exécute le corps qui lui a été associé;
- une TTR est dite *Ecran* si elle est mandataire d'une tâche *Active* située dans un autre exécutable.

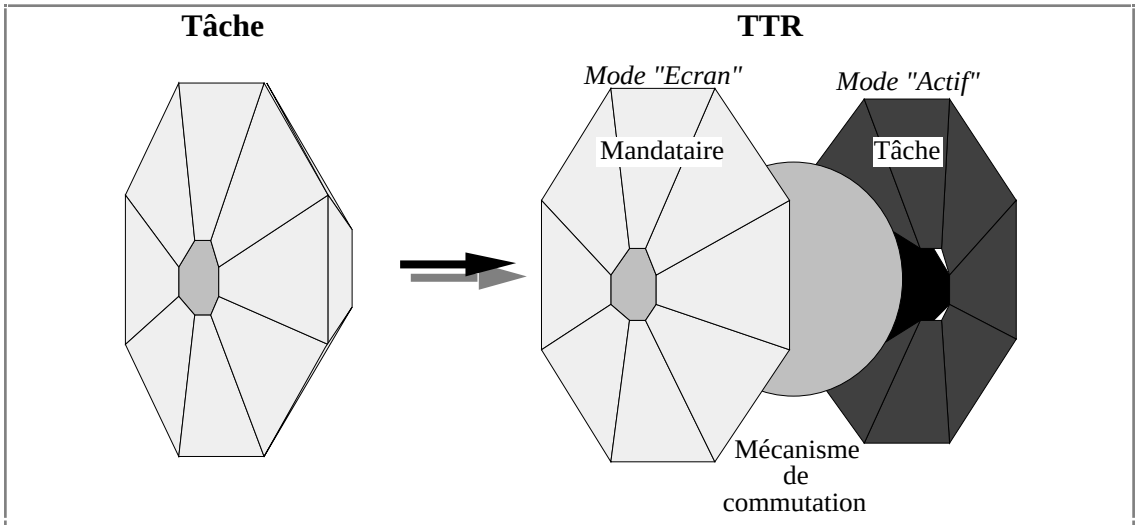


Figure VII.7 : Transformation d'une tâche de l'application en TTR (Tâche Transparente à la Répartition).

En mode *ecran*, une TTR possède les mêmes points d'entrée qu'en mode *actif*, plus des points d'entrée dédiés au contrôle. Cela permet à une TTR de masquer son mode à toute TTR appelante.

Ainsi, une TTR, sur un site donné, soit exécute le code qui lui est associé, soit est un mandataire d'une tâche distante. Dans ce cas, elle retransmet les

messages qui sont destinés à la tâche exécutant le code “originel” dans un autre exécutable, sur un autre site.

Principe de conception P2 : **Transformation des tâches de l’application**

Nous transformons toutes les tâches de l’application en “Tâches Transparentes à la Répartition” (TTR).

Conformément aux objectifs que nous nous sommes fixés, le placement des tâches de l’application est indiqué de manière externe. Chaque exécutable reçoit la liste des TTR en mode *actif*. Ces informations sont placées dans un fichier de configuration.

Les TTR en mode *actif* ne communiquent pas directement avec l’extérieur. Elles communiquent localement avec la TTR voulue. Celle-ci répond directement si elle est en mode *actif*; sinon, elle transmet la demande à l’exécutable contenant la TTR correspondante en mode *actif*.

Si l’application est répartie sur N exécutables, il y aura exactement N - 1 exemplaires de chaque tâche en mode *ecran*, et un seul exemplaire *actif* contenu dans l’exécutable où la tâche doit s’exécuter.

Exemple VII.1 : Considérons une application composée de 4 tâches (A, B, C et D) répartie dans trois exécutables (Figure VII.8). C et D sont Actives dans l’exécutable 1, B dans l’exécutable 2 et A dans l’exécutable 3.

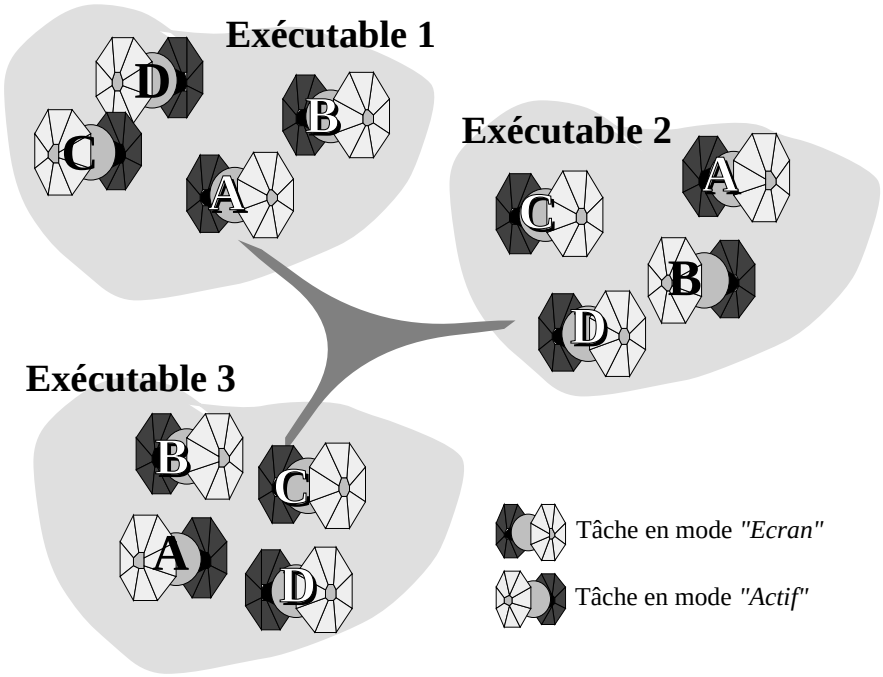


Figure VII.8 : Exemple de répartition d’une application composée de 4 tâches : A, B, C et D réparties dans trois exécutables.

Principe de conception P3 : **Le rendez-vous distant**

Deux TTR distantes réalisent un rendez-vous Ada distant à l’aide d’un protocole.

L'application de ce principe permet à deux TTR distantes de communiquer par appel de point d'entrée comme si elles se situaient sur le même site.

Pendant le traitement d'un rendez-vous, la tâche appelée réalise l'*accept* pendant que l'appelante attend que le traitement soit complètement effectué [Ada 83]. Ce comportement doit être reproduit lorsque les deux TTR ne sont pas dans le même exécutable.

Le réseau de Petri de la Figure VII.9 modélise le protocole de dialogue entre deux TTR actives sur des sites distants. Sa validation repose sur la contruction du graphe des marquages accessibles présenté dans la Figure VII.10.

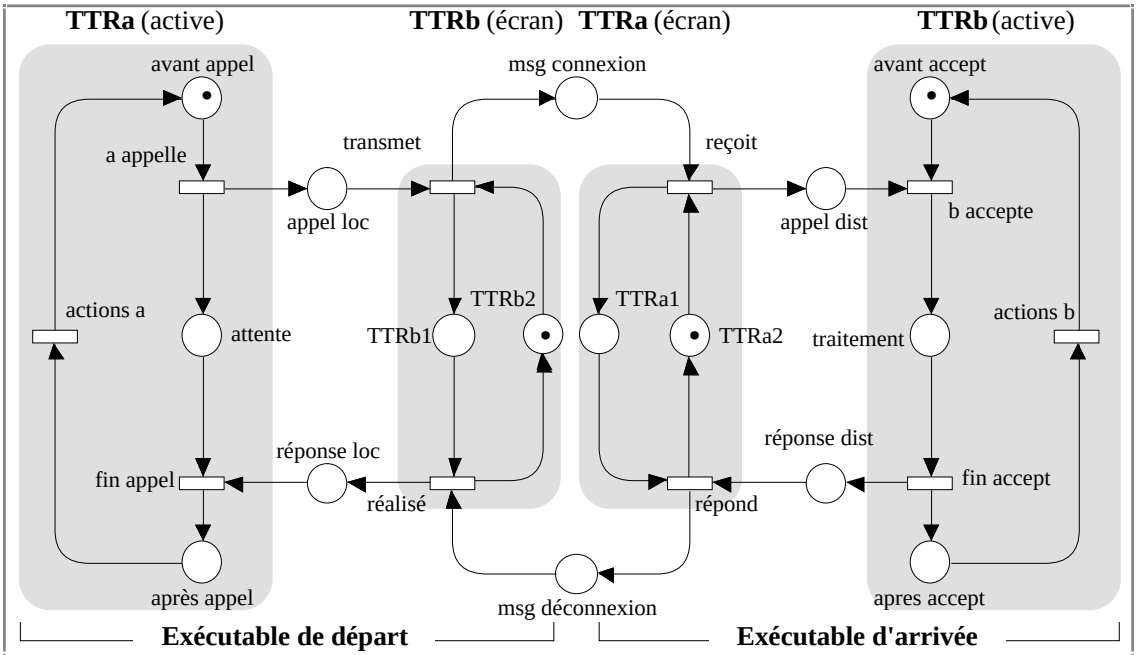


Figure VII.9 : Modélisation du protocole de dialogue entre deux TTR distantes.

Le mécanisme repose sur l'action des TTR en mode *écran* qui dialoguent à l'aide des messages de connexion et de déconnexion. Le message de connexion est envoyé par le représentant du destinataire sur le site de départ en direction du représentant de l'émetteur dans le site d'arrivée. Le message de déconnexion est transmis de la même façon dans l'autre sens.

Le fonctionnement d'une TTR en mode *écran* s'apparente, au niveau Ada, à celui des STUB [Stevens 90] dans la sémantique des Remote Procedure Call (RPC).

Le principe impose que, sur le site émetteur, la TTR appelée, en mode *écran*, se transforme en demandeur d'un service distant, auprès du représentant en mode *écran* de la TTR qui a demandé le rendez-vous, sur le site où la TTR appelée est active. Cependant, en Ada, il n'existe aucun moyen, pour une tâche appelée, de récupérer l'identité de la tâche appelante. Cette dernière doit systématiquement s'identifier à chaque appel d'un point d'entrée. Si les deux TTR sont actives sur le même site, l'information sera inutilisée.

Propriété attendue : Le protocole défini ne doit en aucun cas provoquer un blocage du système. Si la tâche appelée est prête à réaliser l'accept, le rendez-vous doit avoir lieu.

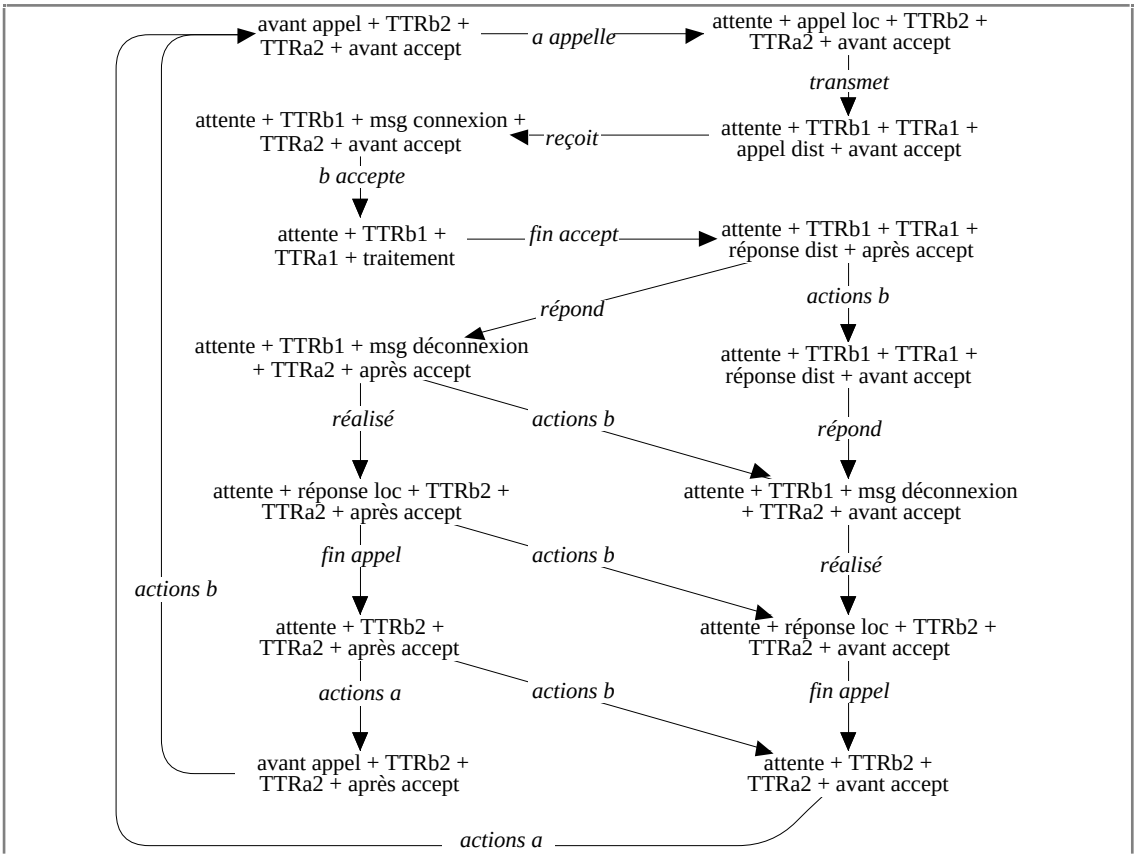


Figure VII.10 : Graphe des marquages accessibles du modèle de la Figure VII.9.

Validation : Le graphe des marquages accessibles de la modélisation du protocole de réalisation du rendez-vous distant est donné en Figure VII.10.
Le marquage initial constitue un état d'accueil, le modèle est donc vivant. Les deux TTR actives reprennent leur exécution après réalisation de l'accept et les TTR écran se retrouvent dans un état cohérent, c'est-à-dire prêtes à redevenir mandataires.

Ce mécanisme de rendez-vous distant permet à T1 et T2 de dialoguer comme si elles étaient toutes deux sur le même site. Le protocole défini par [Ada 83] est donc bien respecté.

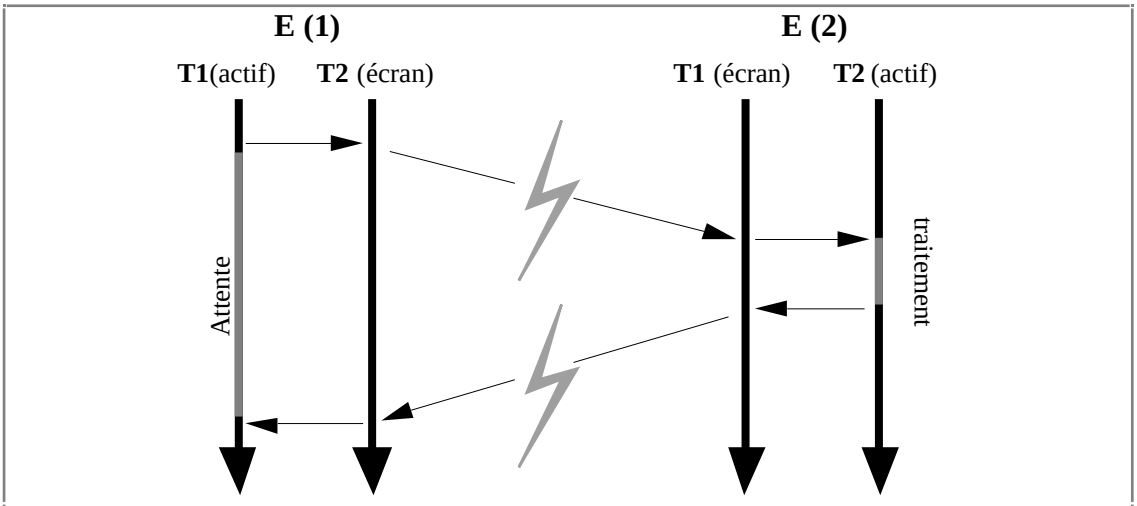


Figure VII.11 : Exemple de réalisation d'un rendez-vous entre deux TTR distantes.

Exemple VII.2 : Soit une application comportant un exécutable E, lancé sur 2 sites. Nous notons chacun des exécutables E(1) et E(2). Considérons deux TTR : T1, active dans E(1) et T2, active dans E(2). Supposons que T1 appelle un point d'entrée de T2 (Figure VII.11). T1 appelle localement le point d'entrée de T2 qui est en mode Ecran (donc, mandataire de T2 dans E(2)). T2 transmet donc la demande en direction T1 en mode Ecran sur E(2). Enfin, T1 appelle localement le point d'entrée avant de retransmettre la réponse vers E(1) (via T2 en mode écran).

2.5. DÉFINITION D'UNE BOÎTE À OUTILS DE RÉPARTITION

Les mécanismes que nous avons développés reposent sur l'échange de messages entre exécutables distants.

Ainsi, les seuls services dont nous avons besoin sont liés à la manipulation de messages. Nous supposons que les communications sont fiables.

***Hypothèse :* Communications fiables**

Les communications sont fiables, la durée de transmission d'un message sur le réseau est bornée.

Les services de répartition offerts sont les suivants [Mullender 89, Stevens 90, Marzullo 91] :

- lancement à distance d'un programme exécutable;
- établissement d'un lien de communication avec un exécutable dont l'identité est connue;
- signalisation et lecture des messages reçus;
- émission d'un message.

Ces services constitue une *boîte à outils* (Figure VII.12). Elle est basée, en ce qui concerne les services de communication, sur le mécanisme de *socket* dans le système UNIX™ [Stevens 90].

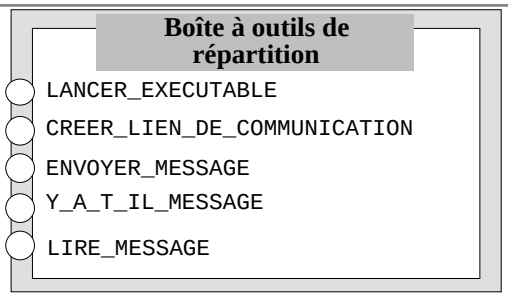


Figure VII.12 : Spécification de la boîte à outils permettant l'application de nos principes de répartition.

La primitive `LANCER_EXECUTABLE` est une fonction retournant une valeur permettant d'identifier l'exécutable créé. Cette identité sert par la suite dans l'établissement d'un lien de communication. Elle possède deux paramètres :

- la désignation du fichier exécutable;
- le nom de la machine où il doit être lancé.

Les systèmes d'exploitation gérant la communication entre différentes machines proposent tous des services permettant de lancer un exécutable à

distance. Sous UNIX™, une telle fonctionnalité est offerte par *rsh* [Stevens 90]. Par ailleurs, d'autres outils (sur-ensembles de systèmes d'exploitation), plus élaborés, ont été développés en prenant notamment en compte des critères de charge [Theimer 88, Folliot 89].

La primitive `CREER_LIEN_DE_COMMUNICATION` est une fonction retournant une valeur permettant d'identifier le lien de communication créé. Cette identité sert par la suite dans toute communication. Elle ne nécessite qu'un seul paramètre : l'identité de l'exécutable que l'on veut relier.

Le service `ENVOYER_MESSAGE` retourne une valeur indiquant si l'émission s'est déroulée sans problème. Il possède deux paramètres :

- l'identité du lien de communication;
- le message à transmettre.

Le service `Y_A_T_IL_MESSAGE` permet de savoir si un message est présent sur un lien de communication. Il possède un seul paramètre : l'identité du lien de communication que l'on veut sonder.

La primitive `LIRE_MESSAGE` ramène le message présent sur un lien de communication. Elle possède un seul paramètre : l'identité du lien de communication contenant le message à lire.

3. Réalisation d'une application répartie

Dans cette section, nous décrivons nos principes de réalisation d'une application Ada répartie sur plusieurs sites.

Nous proposons de centraliser, par souci de performances, la réception des messages définis dans le protocole de réalisation d'un rendez-vous distant. Pour le contrôle de l'exécution de l'application, nous discutons les solutions à contrôle centralisé et décentralisé.

3.1. PRINCIPES DE RÉALISATION

Deux problèmes se posent, lorsque nous voulons appliquer les principes de conception que nous avons énoncés.

Le premier est lié à la réception d'un message. Il faut que le mécanisme développé soit efficace. Un message reçu par un exécutable est potentiellement destiné à toute TTR en mode écran. Il faut cependant limiter l'activité de ces TTR au strict minimum. Un mécanisme doit être mis en place pour gérer la préemption des TTR en mode écran.

Le second concerne la localisation des informations décrivant l'application répartie, c'est-à-dire :

- le placement des tâches dans les différents exécutables,
- le contrôle de l'exécution de l'application : la disparition de l'ensemble des tâches d'un programme Ada provoque la terminaison de l'unité maître. Si les tâches sont réparties dans plusieurs exécutables (donc, plusieurs unités maîtres) contenant à la fois des tâches passives (les TTR en mode écran) et des tâches actives (les TTR en mode actif), il faut développer un mécanisme permettant de détecter la terminaison de toutes les TTR en mode Actif.

3.1.1. Réception d'un message

Pour la réception de messages, nous introduisons le principe de réalisation suivant [Sens 90] :

Principe de réalisation R1 : **L'aiguilleur**

> La réception des messages dans un exécutable est centralisée par une tâche Ada : "l'aiguilleur".

L'aiguilleur a pour rôle de collecter tous les messages provenant de l'extérieur et de les transmettre aux tâches destinataires. Outre la distribution des messages, l'aiguilleur contrôle les communications : il détecte une rupture d'un lien de communication avec les autres exécutables composant l'application.

La centralisation de la réception, au niveau d'une seule tâche, évite le réveil, à chaque réception d'un message, de toutes les TTR écran en attente d'une réponse.

L'aiguilleur est réveillé lorsqu'un message est présent sur un lien de communication. Il lit alors dans l'en-tête le nom de la TTR destinataire pour lui transmettre le contenu du message. Conformément au principe P3 (section 2.4), il s'agit de l'appelante (en mode écran) du point d'entrée dans l'exécutable émetteur.

De par le principe P3, l'aiguilleur ne reçoit jamais de messages destinés à une TTR en mode actif.

La TTR écran décode l'identité de la TTR dont elle doit appeler le point d'entrée et les arguments de l'appel. Elle réalise ensuite l'appel du point d'entrée correspondant et attend sa réalisation afin de retransmettre les résultats.

L'aiguilleur reçoit tous les messages en provenance de l'extérieur, de ce fait il représente un goulot d'étranglement. Il faut donc minimiser les traitements à effectuer (réception du message, décodage du destinataire, transmission du message). Ainsi le traitement du message, notamment le décodage des données, est laissé aux TTR en mode écran qui assurent l'appel du point d'entrée en local.

Pour cela, chaque TTR en mode écran possède un point d'entrée spécialisé permettant à l'aiguilleur de lui faire parvenir les messages.

3.1.2. Le contexte de l'application répartie

Nous avons adopté le principe de réalisation suivant :

***Principe de réalisation R2 :* Gestion du contexte global de l'application**

> Les exécutables ont une connaissance minimale du contexte de l'application (placement et activité des TTR). Cette connaissance est regroupée dans un processus dédié : le superviseur. L'initialisation et la terminaison de l'application sont gérées par le processus superviseur.

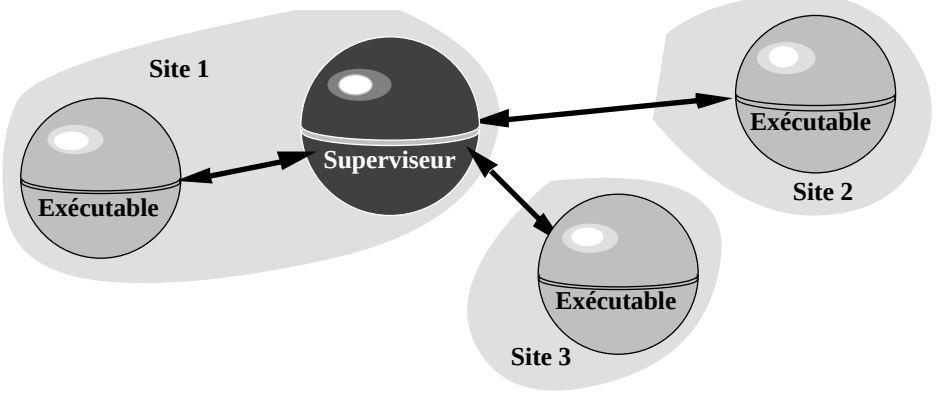


Figure VII.13 : Architecture centralisée, conditionnée par le principe de réalisation R2.

Le principe R2 conditionne une stratégie de type centralisé : le superviseur est au centre d'une configuration en étoile (Figure VII.13). Cela simplifie le traitement des communications au niveau des exécutables qui ne communiquent qu'avec le superviseur.

Le dual du principe R2 conditionne une politique complètement décentralisée (Figure VII.14). Chaque exécutable a connaissance de la topologie de l'ensemble de l'application. Pour résoudre les problèmes liés au contrôle de l'application, l'ensemble du contexte doit être dupliqué. Des mécanismes de mise à jour doivent être mis en place [Raynal 84, Raynal 88, Mullender 89].

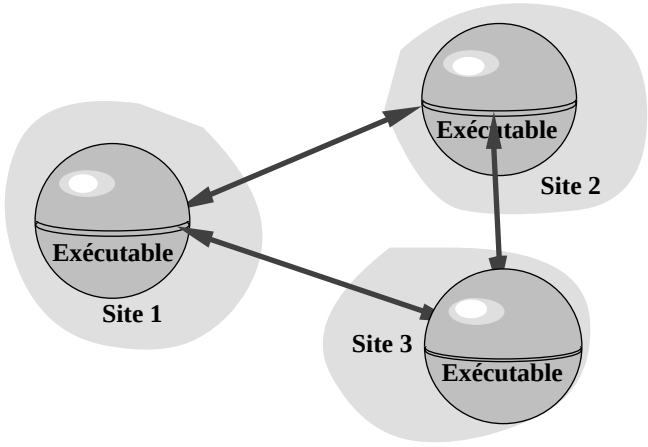


Figure VII.14 : Architecture centralisée, conditionnée par le dual du principe de réalisation R2.

3.2. ARCHITECTURE CENTRALISÉE

Nous décrivons ici les techniques que nous avons développées pour implémenter une architecture centralisée. Les problèmes que nous abordons sont :

- l'émission et le routage des messages,
- la gestion du contexte d'exécution de l'application répartie.

Nous présentons également l'adaptation d'une application parallèle déjà existante, en vue de la répartir sur un ensemble de machines.

3.2.1. Emission et routage des messages

L'émission de messages est gérée par les TTR en mode écran qui font office d'interface de communication. Elles construisent le message de connexion puis l'émettent en direction du superviseur. Elle se bloquent en attente du message de déconnexion.

Les services locaux de communication sont indépendants du nombre d'exécutables et de leur localisation. En effet, seul le superviseur connaît la configuration de l'application.

Pour acheminer un message à une TTR distante, il faut déterminer l'exécutable dans lequel elle est active.

En accord avec le principe de réalisation R2, seul le superviseur connaît la configuration de l'application. Elle est décrite dans une *table de localisation des TTR* qui, à chaque TTR, fait correspondre la position de l'exécutable où elle fonctionne en mode actif. La *table de localisation des TTR* est créée à l'initialisation de l'application.

Le superviseur contient une tâche, le *routeur*, qui assure l'envoi du message au bon exécutable. Le *routeur* est réveillé lorsqu'un message est présent sur un lien de communication. Il décode le nom de la TTR destinataire, consulte la *table de localisation* pour trouver l'exécutable où elle est active puis retransmet le message.

La diffusion est à la charge du superviseur : le message lui est transmis une seule fois par l'exécutable émetteur. Il est ensuite réémis en direction des autres exécutables.

Exemple VII.3 : La Figure VII.15 récapitule les mécanismes d'émission et de réception des messages dans l'application, à travers la réalisation d'un rendez-vous distant.
Supposons que la TTR A, active dans l'exécutable 1, souhaite effectuer un rendez-vous avec la TTR B, active sur l'exécutable N. Elle appelle le point d'entrée correspondant de B dans l'exécutable 1. B, étant en mode écran, construit le message et le transmet au superviseur. Il est réceptionné par le *routeur* qui le redirige vers l'exécutable N. Dans l'exécutable N, il est reçu par l'aiguilleur qui réveille la TTR A en mode écran. Cette dernière, après décodage des arguments du point d'entrée, appelle la TTR B en mode actif. Les paramètres en sortie emprunteront le chemin inverse.

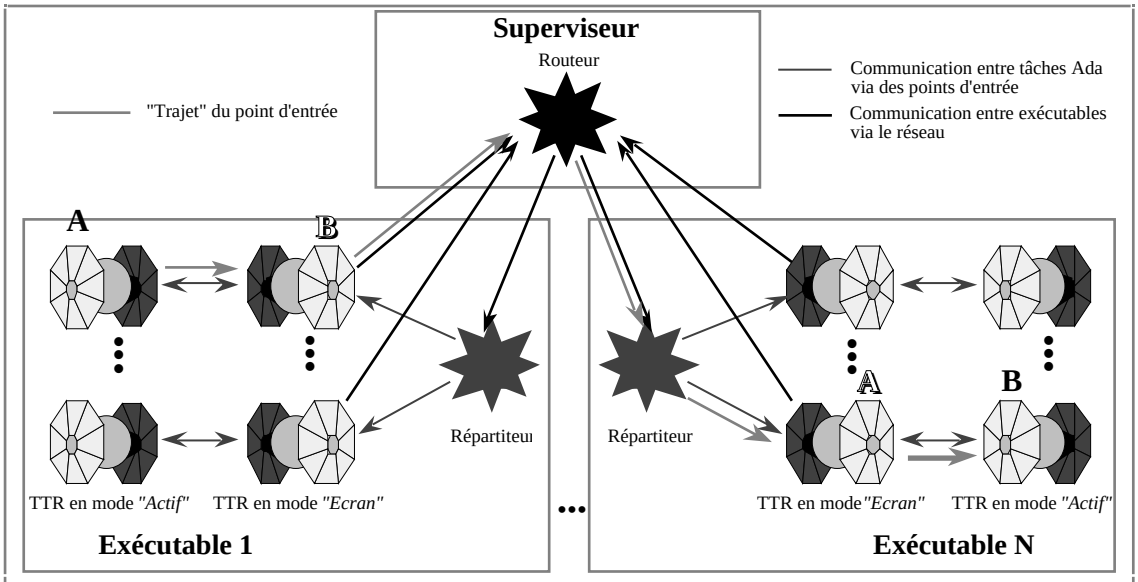


Figure VII.15 : Gestion des communications au niveau d'un exécutable. Nous donnons ici un exemple de rendez-vous entre deux TTR actives dans des exécutables différents.

3.2.2. Initialisation et terminaison

Dans le superviseur, une tâche, le *super-gardien*, pilote des tâches de contrôle situées dans chaque exécutable : les *gardiens locaux*.

Seul le *super-gardien* possède une vision complète de l'état de l'application (liste des TTR en activité). Il supervise l'initialisation et la terminaison de l'application (Figure VII.16).

Les *gardiens locaux* sont asservis au *super-gardien* et lui transmettent toute information concernant les tâches de l'exécutable. Ils se comportent comme des mandataires du *super-gardien*.

Le *super-gardien* et les *gardiens locaux* utilisent le même système de communication que les TTR en mode écran. Les *gardiens locaux* transmettent directement leurs messages en direction du processus superviseur et reçoivent leurs instructions via l'aiguilleur de leur exécutable. Le *super-gardien* émet directement ses messages vers l'exécutable concerné et reçoit des informations des *gardiens locaux* via le routeur qui, dans ce cas, se comporte comme l'aiguilleur du processus superviseur.

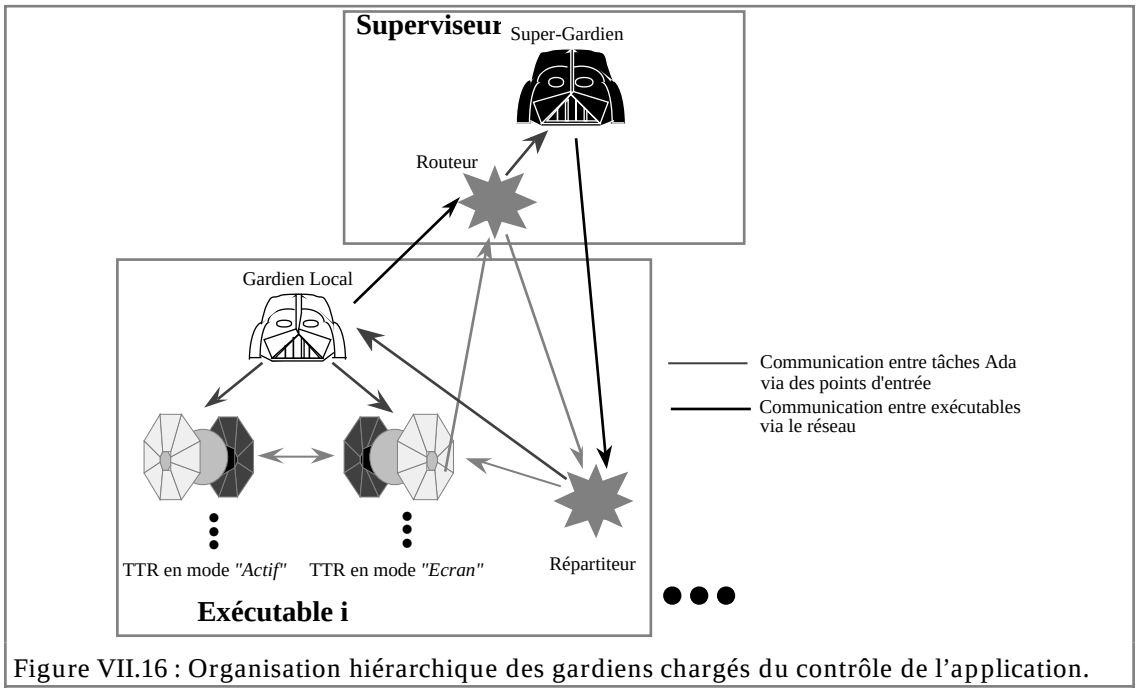


Figure VII.16 : Organisation hiérarchique des gardiens chargés du contrôle de l'application.

L'initialisation de l'application se déroule en deux étapes :

- la phase de lancement : il faut lancer à distance les exécutables sur les différents sites et créer des liens de communication avec le superviseur;
- la phase de placement : il faut activer, dans chaque exécutable, les TTR dans le mode adéquat.

Le fichier de configuration décrivant le placement des TTR et les sites d'exécution de l'application est consulté par le *super-gardien*.

La phase de lancement

Le lancement suit un protocole de type envoi-attente [Pujolle 85] :

- dans un premier temps, le superviseur lance les différents exécutables sur des sites distants, puis attend les réponses de chacun d'eux;
- une fois toutes les réponses obtenues, le superviseur transmet aux exécutables un message leur signalant que la phase de lancement est réussie.

Si le *super-gardien* n'arrive pas à lancer tous les exécutables, il envoie une requête de terminaison à ceux qu'il a déjà créés. En cas de panne du superviseur, les exécutables se terminent à l'expiration d'un temporisateur.

La phase de placement

A partir de la configuration indiquée par l'utilisateur, le *super-gardien* construit la *table de localisation des TTR* puis transmet aux *gardiens locaux* la liste des TTR à activer en mode actif. L'initialisation de l'application est alors terminée.

Il existe deux types de terminaisons : les terminaisons normales et les terminaisons sur erreur.

Terminaison normale

Une TTR en mode actif qui se termine se signale à son *gardien local*. Ce dernier notifie l'événement au *super-gardien* qui maintient une table des TTR encore en activité.

Lorsque toutes les TTR actives ont signalé leur terminaison, le super-gardien demande aux *gardiens-locaux* de tuer toutes les tâches restantes (TTR en mode écran et aiguilleur) afin de terminer les exécutables. Le *super-gardien* détruit alors le *routeur* avant d'achever son exécution.

Terminaison sur erreur

Deux cas d'erreurs sont possibles :

- Terminaison anormale (sur exception) d'une TTR : aucun mécanisme de reprise ne peut être réalisé au niveau du contrôle de la répartition. C'est au niveau de l'application que de tels traitements doivent être spécifiés
- Rupture d'un lien de communication : un mécanisme de reprise, en mode dégradé, sur un nombre restreint de sites [Avizienis 87, Nelson 90] peut être envisagé. Nous n'avons pas étudié cette éventualité.

Une TTR se terminant sur une exception le signale au *gardien local*. L'information est transmise au *super-gardien* qui provoque la terminaison de l'application sans attendre que toutes les TTR actives soient terminées.

Une rupture d'un lien de communication est détectée de chaque côté du partitionnement du réseau :

- exécutables : l'aiguilleur prévient le *gardien local* qui provoque la terminaison de l'exécutable;
- superviseur : le routeur prévient le *super-gardien* qui, après avoir ordonné aux exécutables qu'il peut encore joindre de se terminer, achève son exécution[16].

---

16 L'achèvement d'une exécution pose des problèmes de Droit Humanitaire International tel qu'ils sont régi par les Conventions de Genève de 1949 et les Protocoles additionnels de 1977 [Lanord 92].

3.2.3. Adaptation d'une application déjà existante

La notion de service de répartition doit permettre la transformation d'une application multi-tâches déjà écrite mais exécutable sur une machine mono-processeur, en une application répartie. Cet aspect est important dans le cadre du prototypage, si nous voulons étudier les modifications à envisager sur un prototype centralisé afin de le répartir.

La transformation consiste, d'une part à ajouter les mécanismes de communication et de contrôle, d'autre part à écrire le superviseur (Figure VII.17).

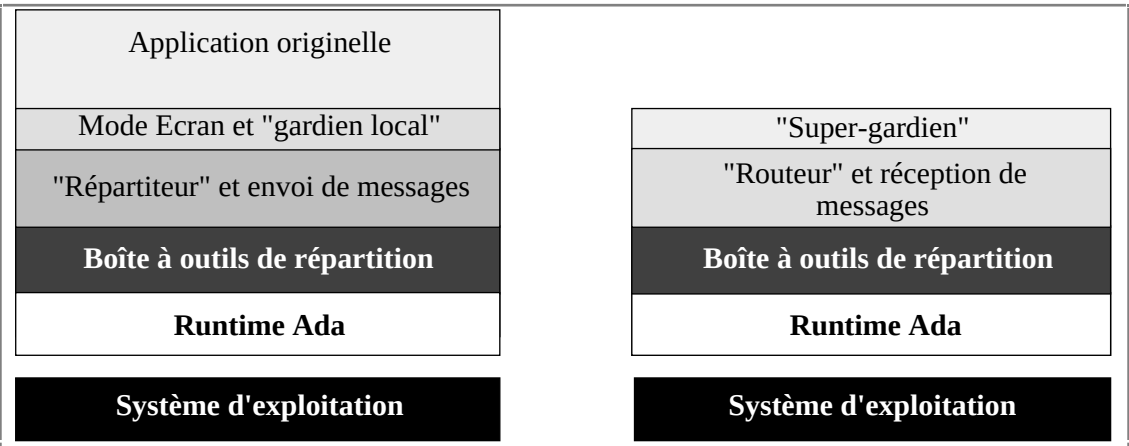


Figure VII.17 : Les modules composant une application répartie pour une architecture centralisée.

Le superviseur, comme l'exécutable, reposent sur la *boîte à outils de répartition*.

Le superviseur doit être conçu dans son intégralité la première fois qu'une application est répartie. Il est par la suite réutilisable pour n'importe quelle autre application :

- le super-gardien doit être paramétré en fonction du nombre de TTR composant l'application;
- le routeur doit être paramétré en fonction de la configuration du réseau de machines sur lequel fonctionne l'application.

Un exécutable est construit à partir du programme originel :

- il faut transformer les tâches en TTR, ce qui revient à écrire les modes écran et ajouter l'identification des tâches appelantes à chaque appel de point d'entrée; un tel travail doit être effectué chaque fois;
- il faut créer le gardien local : son comportement étant indépendant de l'application répartie, il est réutilisable;
- il faut créer l'aiguilleur : il peut aisément être obtenu à partir d'un modèle générique.

3.2.4. Avantages et inconvénients

Le principal avantage de cette approche, outre la simplicité de sa mise en œuvre, est que le traitement des services de répartition, au sein des

exécutables, est indépendant de leur nombre. Les tâches émettrices se contentent d'envoyer leurs messages à un unique destinataire : le superviseur.

Il n'y a pas de risque d'incohérence dans la description de l'état d'exécution de l'application puisque toute la connaissance est centralisée. L'initialisation des exécutables de l'application, comme la détection de sa terminaison, en sont grandement facilitées.

Le problème vient essentiellement du fait que tout message transmis transite forcément par le serveur de communication : en introduisant une indirection supplémentaire, le nombre de messages est doublé; il existe donc un goulot d'étranglement potentiel. La durée de transmission d'un message est directement liée à la capacité du serveur de communication à les traiter rapidement; en cas de flot important, l'application risque de s'en trouver particulièrement ralentie.

3.3. ARCHITECTURE DÉCENTRALISÉE

Nous décrivons ici les techniques que nous avons développées pour implémenter une architecture décentralisée.

Le dual du principe R2 implique que :

- dans chaque exécutable, un gardien contrôle l'exécution des TTR,
- un mécanisme de routage doit être intégré dans chaque exécutable.

Pour éviter les problèmes d'incohérence, chaque exécutable possède une copie du contexte. Il diffuse les changements apportés à ce contexte et coopère avec ses homologues pour piloter l'application.

3.3.1. Emission et routage des messages

L'émission d'un message est lié au routage. Chaque exécutable maintient une copie de la *table de localisation des TTR* associant une TTR à un exécutable. Les messages sont de la sorte transmis directement à l'exécutable destinataire.

La *table de localisation des TTR* est créée à l'initialisation de l'application. Elle est accessible en lecture par toutes les TTR ainsi que le gardien de l'exécutable.

Exemple VII.4 : La Figure VII.18 récapitule les mécanismes d'émission et de réception des messages dans l'application, à travers la réalisation d'un rendez-vous distant.
Supposons que la TTR A, active sur l'exécutable 1, souhaite effectuer un rendez-vous avec la TTR B, active sur l'exécutable N. Elle appelle le point d'entrée correspondant de B dans l'exécutable 1. B, étant en mode écran, construit le message et consulte la table de localisation afin d'identifier l'exécutable destinataire. Le message sera ainsi émis en direction de l'exécutable N. Il est réceptionné par l'aiguilleur qui réveille la TTR A en mode écran. Cette dernière, après décodage des arguments du point d'entrée, appelle la TTR B en mode actif. Les paramètres en sortie emprunteront le chemin inverse.

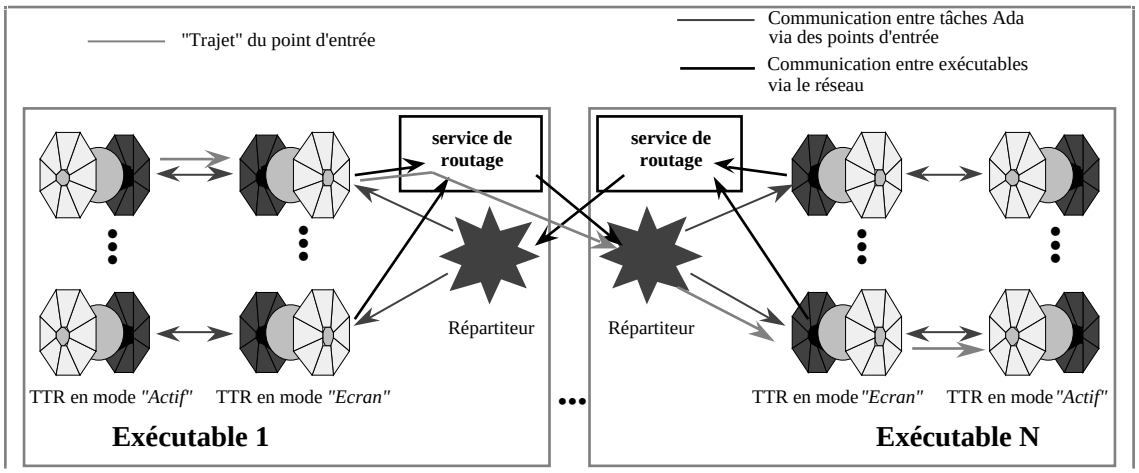


Figure VII.18 : Gestion des communications entre exécutables. Nous donnons ici un exemple de rendez-vous entre des TTR actives dans des exécutables différents

3.3.2. Initialisation et terminaison

Chaque exécutable contient un *gardien* qui coopère avec ses homologues. Comme précédemment, l'initialisation de l'application se déroule en deux temps : lancement et placement.

La phase de lancement

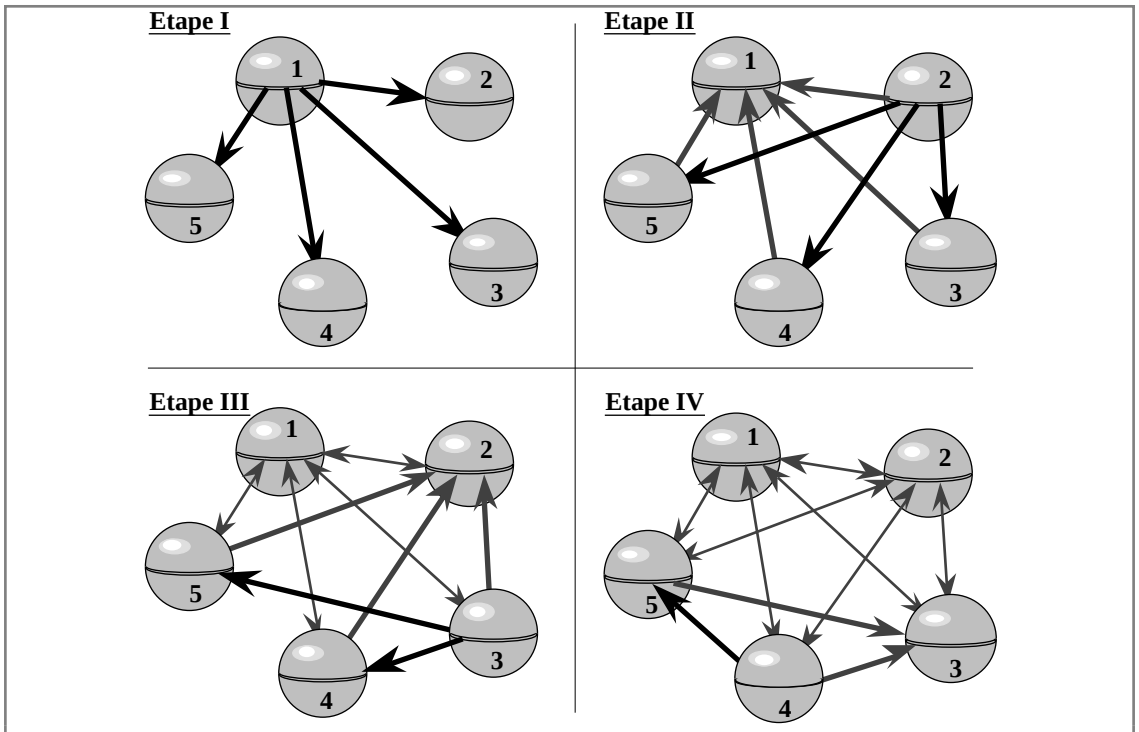


Figure VII.19 : Construction du réseau de communication maillé pour une application répartie sur 5 exécutables. Chaque gardien local débute la phase de placement lorsque *la table de localisation des exécutables* est à jour.

Le lancement des exécutables et l'établissement des liens de communication posent un problème : il faut établir une configuration en clique. Chaque exécutable possède un lien de communication avec tous ses homologues.

La phase de lancement est décrite en Figure VII.19. Elle est pilotée par le premier exemplaire de l'exécutable qui s'attribue le numéro 1 (Etape 1). Il lance successivement ses homologues en leur attribuant des numéros consécutifs. Tout exécutable doit attendre un message de connexion de ses homologues de numéro inférieur. A chaque réception de message, il met à jour une *table de localisation des exécutables*, associant une machine à chaque exécutable. Ensuite, il émet une demande de connexion en direction de chacun de ses homologues de numéro supérieur afin de les intégrer dans *la table de localisation des exécutables*. La phase de lancement se termine tout naturellement, lorsque chaque exécutable a complété sa *table de localisation des exécutables* [Cheng 90].

La phase de placement

A partir de la configuration indiquée par l'utilisateur et de la *table de localisation des exécutables*, chaque *gardien* construit la *table de localisation des TTR* et lance les TTR de l'exécutable. L'initialisation de l'application est alors terminée.

Il existe deux types de terminaison : les terminaisons normales et les terminaisons sur erreur.

Terminaison normale

La terminaison de toute TTR en mode Actif est signalée au gardien de l'exécutable. Ce dernier diffuse l'événement à ses homologues, qui mettent à jour leur copie de *la table de localisation des TTR*. Lorsque sa table de localisation des TTR est vide, le gardien tue l'aiguilleur de son site, puis se termine après avoir prévenu ses homologues.

Terminaison sur erreur

Deux cas d'erreurs sont possibles :

- Terminaison anormale (sur exception) d'une TTR : aucun mécanisme de reprise ne peut être réalisé au niveau du contrôle de l'application. C'est au niveau de l'application que de tels traitements doivent être spécifiés.
- Rupture d'un lien de communication : un mécanisme de reprise, en mode dégradé, sur un nombre restreint de sites [Avizienis 87, Nelson 90] peut être envisagé. Nous n'avons pas étudié cette éventualité.

Une TTR se terminant sur une exception le signale au *gardien* de son exécutable qui transmet l'information à ses homologues. Un tel événement implique la terminaison brutale de l'application.

Dès que la rupture d'un lien de communication a été détectée, le gardien prévient ses homologues encore connectés puis détruit toutes les tâches de l'exécutable avant d'achever son exécution.

3.3.3. Adaptation d'une application déjà existante

La notion de service de répartition doit permettre la transformation d'une application multi-tâches déjà écrite mais fonctionnant sur une machine mono-processeur, en une application répartie. Cet aspect est important dans le cadre du prototypage, si nous voulons étudier les modifications à envisager sur un prototype centralisé afin de le répartir.

La transformation consiste à ajouter les mécanismes de communication et de contrôle (Figure VII.20).

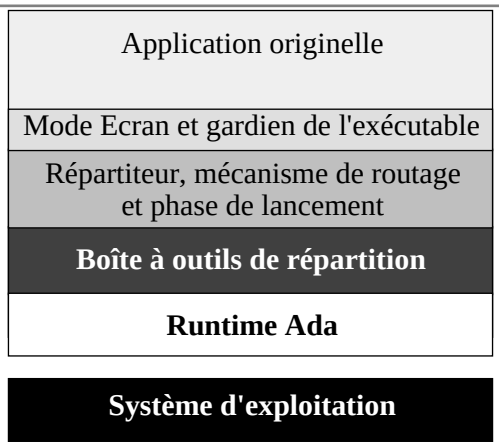


Figure VII.20 : Les modules composant une application répartie pour une architecture décentralisée.

L'application des principes repose sur la boîte à outils de répartition. Un exécutable est construit à partir du programme originel de la façon suivante :

- il faut transformer les tâches en TTR, ce qui revient à écrire les modes écran et ajouter l'identification des tâches appelantes à chaque appel de point d'entrée; un tel travail doit être effectué à chaque répartition d'un programme déjà existant;
- il faut créer le gardien et l'aiguilleur : comme dans le modèle centralisé, ils peuvent être aisément déduits d'un modèle type;
- il faut créer un mécanisme de routage : il peut être obtenu à partir d'un modèle paramétrable;
- il faut intégrer la *phase de lancement* : cette étape, indépendante de l'application, est par la suite réutilisable.

Une telle organisation est très facilement portable d'une architecture à une autre puisque tous les éléments non portables sont regroupés dans la boite à outils de répartition. Seule cette unité est à réécrire.

3.3.4. Avantages et inconvénients

L'architecture décentralisée permet d'éviter le goulot d'étranglement potentiel au niveau du superviseur. Les communications sont plus rapides et le réseau moins chargé puisque les messages ne transitent qu'une seule fois.

Cependant, cette approche est plus délicate à mettre en œuvre. L'existence de copies multiples du contexte d'exécution d'une application implique la mise en œuvre de protocoles de mise à jour. Des messages de contrôle particuliers doivent être créés à cet effet.

4. Application au prototypage

Nous avons exposé au paragraphe 3 comment appliquer les principes définis dans le paragraphe 2 en vue de répartir une application Ada multi-tâches centralisée.

Le prototype, tel qu'il a été défini dans le Chapitre V, correspond au profil exigé pour l'application de notre technique. Les Ressources ne posent pas de problème puisque leur manipulation est gérée par un serveur.

4.1. CHOIX STRATÉGIQUES

Dans cette section, nous exposons les choix que nous avons effectués dans la conception du prototype réparti.

Architecture

Dans la section précédente, nous avons présenté deux types d'architecture : centralisée et décentralisée. Des expériences préliminaires [Kordon 91c] ont été réalisées sur une architecture de type centralisé. L'objectif était essentiellement une étude de faisabilité, mais quelques tests ont montré un impact important du goulot d'étranglement représenté par le superviseur.

En conséquence, nous avons choisi de concevoir le prototype réparti sur le modèle d'une architecture décentralisée.

Regroupement des fonctions de contrôle

Il existe des points communs entre les fonctions du module de contrôle dans le prototype centralisé et les gardiens d'une application répartie, tels qu'ils ont été définis dans la section précédente :

- le module de contrôle est chargé de l'initialisation logique, puis de la surveillance du fonctionnement du prototype centralisé;
- les gardiens d'une application répartie sont chargés de l'initialisation physique (lancement et placement) de l'application, puis de la surveillance de son fonctionnement.

Une fusion des deux mécanismes est souhaitable. L'architecture à plusieurs gardiens, décrite dans le Chapitre IV (section 5.4.2), s'adapte parfaitement au contrôle d'une application à architecture décentralisée si nous considérons qu'il existe un gardien par exécutable.

Chaque gardien réalise les fonctions définies pour un prototype et pour le contrôle d'une application répartie, c'est-à-dire :

- Initialisation physique : elle est constituée des phases de lancement et de placement. Chaque gardien construit la table de localisation des TTR et met en route les mécanismes liés à la répartition :
  - initialisation de l'aiguilleur,
  - lancement des TTR dans le mode adéquat.

- Initialisation logique : dans chaque exécutable, il faut transmettre les paramètres de démarrage aux TTR en mode actif.
- Contrôle : il faut gérer le contexte global du prototype et signaler aux autres exécutables toute modification de l'état des TTR actives. Chaque gardien possède une copie du contexte, il est donc capable de décider de la terminaison de l'exécutable.

La Figure VII.21 décrit les services offerts par le module de contrôle d'une application pour une application répartie. Les services déjà présentés dans le Chapitre IV sont indiqués en italique.

On notera que la nouvelle spécification constitue une extension naturelle de celle définie en Figure IV.43. Ainsi, en terme de spécification, le contrôle d'un eapplication répartie se résume à un enrichissement (ou héritage en terminologie objet) du contrôle centralisé.

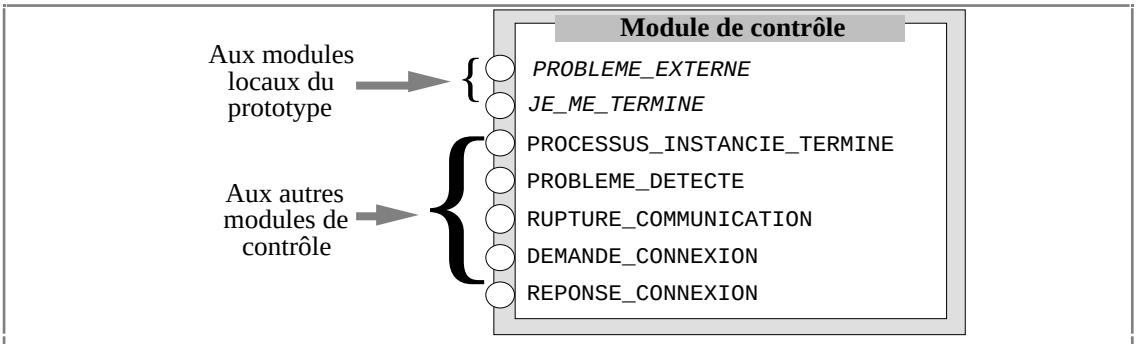


Figure VII.21 : Services offerts par le module de contrôle du prototype réparti.

Le service `PROCESSUS_INSTANCIE_TERMINE` permet à un gardien de propager la terminaison d'une TTR correspondant à un processus instancié à ses homologues afin qu'ils mettent à jour leur copie du contexte global du prototype.

Les services `PROBLEME_DETECTE` et `RUPTURE_COMMUNICATION` permettent de propager un problème. La différenciation permet d'envisager, dans le cas d'une rupture d'un lien de communication, une reprise en mode dégradé sur un nombre réduit de sites. La reprise d'une erreur provenant d'un module externe n'est pas de notre ressort.

Enfin, les services `DEMANDE_CONNEXION` et `REPONSE_CONNEXION` sont utilisés durant la phase de lancement pour établir les connexions entre les différents exécutables.

Dans un premier temps, nous proposons une application directe des principes développés dans le paragraphe 2. Cependant, certaines tâches du prototype ne se prêtent pas à une telle approche. Nous proposons donc un raffinement permettant d'obtenir de meilleurs résultats.

4.2. PREMIÈRE APPROCHE DU PROTOTYPE RÉPARTI

Le prototype associé à un réseau de Petri comporte deux types de tâches :

- Les tâches transformées en TTR :

  - les tâches correspondant aux processus instanciés du modèle : c'est à leur niveau que se situe le parallélisme potentiel du réseau de Petri décrivant le système;
  - les tâches de service : le serveur de Ressources, les serveurs de synchronisations et les tâches d'interface réalisent les communications entre les processus instanciés du modèle; il y a également parallélisme potentiel à ce niveau;
- Les tâches qui ne sont pas transformées en TTR :
  - les tâches de service associées aux serveurs de synchronisations : il s'agit de tâches dont la durée de vie est limitée dans le temps; elles sont toujours créées dans l'exécutable où se trouve le serveur de synchronisations correspondant.

L'application directe des principes de répartition permet d'obtenir un prototype réparti. Cependant, certains phénomènes sont générateurs de problèmes :

- la gestion des Ressources,
- le mécanisme de communication serveur-client.

La gestion des Ressources

Les demandes des clients qui ne se trouvent pas dans le même exécutable que le serveur de Ressources sont beaucoup plus longues à traiter. Les clients placés sur le site du serveur de Ressources sont donc avantagés.

Par ailleurs, il est impossible de profiter du parallélisme potentiel entre Ressources, c'est d'autant plus dommage que l'on dispose d'un parallélisme réel au niveau de l'exécution.

Le problème le plus important apparaît avec les contraintes de placement liées aux composants externes. Comme le serveur de Ressources effectue lui-même l'appel des primitives de manipulation, si deux composants externes doivent être situés sur deux machines différentes, il est impossible de trouver un placement correct.

L'architecture à un seul serveur de Ressources n'est pas adéquat dans le cas d'un prototype réparti. Nous lui préférons l'architecture décrite au Chapitre IV, section 3.5.3 : plusieurs serveurs gèrent l'ensemble des ressources. Chacun d'eux est placé dans un exécutable. Ainsi, le goulot d'étranglement n'existe que s'il est potentiellement présent dans le modèle.

Avec la *délocalisation* des Ressources, des problèmes liés à la gestion du marquage apparaissent. Ils s'apparentent à la gestion de données réparties [Bernstein 78, Bernstein 81, Raynal 84, Raynal 91]. Des mécanismes doivent être mis en place pour préserver :

- l'équité dans la réalisation des demandes, c'est-à-dire l'absence de famine ou d'interblocage [Krakowiak 85] dans la réalisation des demandes bloquantes (évaluation d'une précondition);
- l'intégrité des données contenues dans les Ressources (absence de perte ou d'incohérence) [Bernstein 78, Bernstein 80, Raynal 91].

Le mécanisme de communication client-serveur

Lorsqu'un client et un serveur sont situés dans des exécutables différents, le mécanisme de communication mis en œuvre (tâches d'interfaces) devient très lourd (Exemple VII.5).

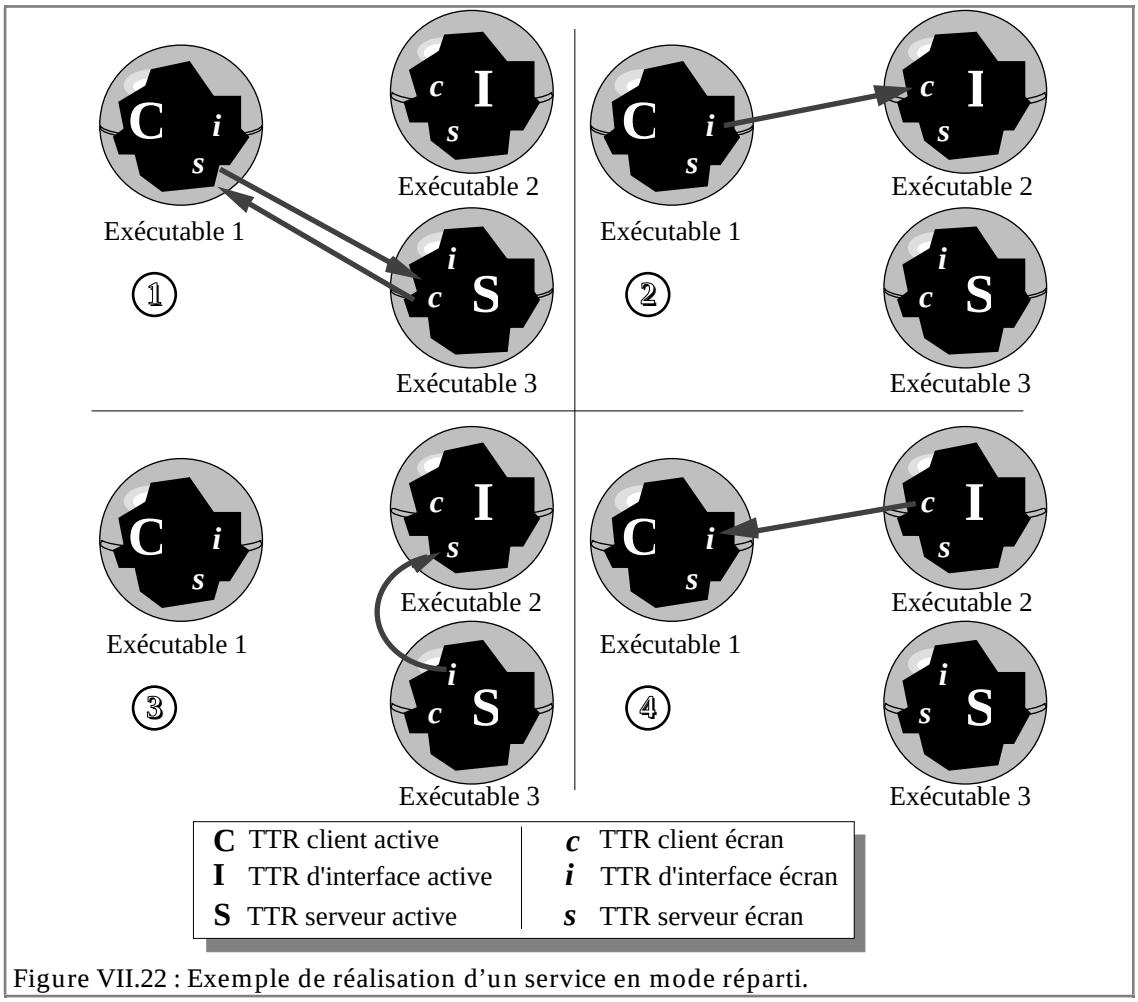


Figure VII.22 : Exemple de réalisation d'un service en mode réparti.

Exemple VII.5 : Considérons un prototype réparti sur trois exécutables (Figure VII.22). Un client est placé dans l'exécutable 1, la tâche d'interface qui lui est associée dans l'exécutable 2 et le serveur auquel il demande un service dans l'exécutable 3. La réalisation d'un service se déroule comme suit :

1 le client effectue une demande au serveur : la réponse lui parvient via le protocole défini par le principe P3 (2 messages);
2 le client se connecte à la tâche d'interface qui lui est associée (1 message);
3 le serveur réveille la tâche d'interface (1 message);
4 la tâche d'interface réveille le client (1 message).

Cinq messages circulent sur le réseau. S'il y avait eu une demande d'annulation refusée par le serveur, il y en aurait eu neuf (2 pour la demande de service, 2 pour l'attente sur temporisation via la tâche d'interface, 2 pour demander l'annulation au serveur puis 3 pour recevoir le résultat du service). Cela correspond à un mauvais placement de la tâche d'interface.

Considérons désormais que les tâches d'interface soient, par définition, placées sur le même site que le client qu'elles représentent. Le nombre de messages échangés sur le réseau diminue.

Exemple VII.6 : Considérons un prototype réparti sur deux exécutables. Un client et la tâche d'interface qui lui est associée sont placés dans l'exécutable 1, le serveur auquel il demande un service dans l'exécutable 2. Une demande de service simple (similaire à celle décrite dans Exemple VII.5) ne nécessite que 3 messages. De même, une demande de service avec annulation refusée n'implique que 5 messages.

Les serveurs de synchronisations peuvent cependant être distants. Le dialogue client-serveur doit être disponible partout, y compris pour des clients en mode écran. Nous considérons donc que le mécanisme de dialogue est actif dans chacun des exécutables.

Dans le cas d'un dialogue entre clients et serveurs distants, le mécanisme sera activé à deux reprises : dans l'exécutable du client et dans celui du serveur. Cela n'est pas grave puisqu'il s'agit de rendez-vous locaux rapides en comparaison des rendez-vous distants.

C'est pourquoi, dans une seconde approche, nous ne transformons pas en TTR les tâches d'interface ni le serveur de Ressources. Le prototype étant généré automatiquement, cette différenciation ne pose pas de problème particulier.

4.3. SECONDE APPROCHE DU PROTOTYPE RÉPARTI

Le prototype associé à un réseau de Petri comporte deux types de tâches :

- les tâches transformées en TTR : le mode écran doit cependant être enrichi pour avoir, sur le site destinataire, un comportement symétrique à celui du site émetteur (section 4.3.1). Il s'agit :
  - des tâches correspondant aux processus instanciés du modèle;
  - des serveurs de synchronisations.
- Les tâches qui ne sont pas transformées en TTR :
  - Les tâches de service associées aux serveurs de synchronisations : comme dans la première approche, elles sont créées dans l'exécutable où se trouve le serveur de synchronisations correspondant;
  - Les serveurs de Ressources : puisque nous considérons désormais l'approche décrite en section 3.5.3 du Chapitre IV, il existe un serveur de Ressources par exécutable;
  - Les tâches d'interface : pour faciliter les communications client-serveur, elles sont actives dans tous les exécutables.

4.3.1. Le mode écran des TTR

Nous avons décidé que les tâches d'interface n'étaient pas soumises au mécanisme des TTR. Cela implique une modification du comportement des TTR associées aux clients et aux serveurs de synchronisations si l'on veut éviter des phénomènes de blocage.

Exemple VII.7 : Considérons un serveur et un client situés sur deux exécutables différents (Figure VII.23). Dans un premier temps, le client effectue une demande au serveur. Ensuite, il se connecte sur la tâche d'interface associée à son exécutable. Le serveur, quant à lui, réveille la tâche d'interface présente sur son site. Dans un exécutable, la

tâche d'interface va attendre un événement provenant du serveur de synchronisations en mode écran, qui ne peut agir tandis que dans l'autre, un message sera adressé au client en mode écran qui ne peut le traiter.

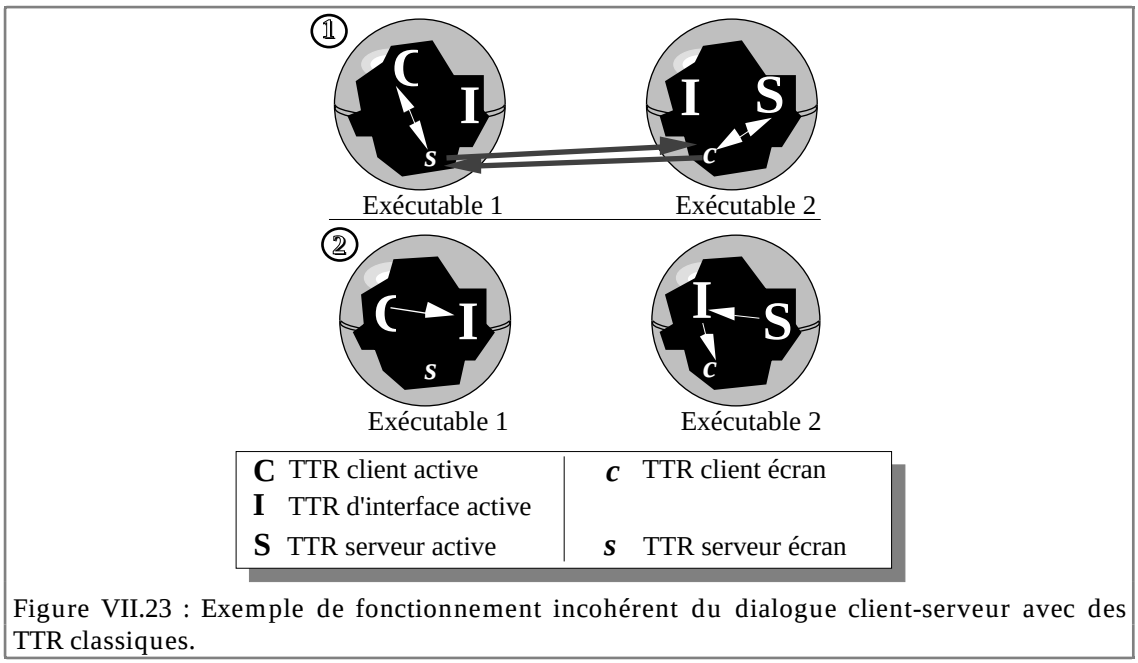


Figure VII.23 : Exemple de fonctionnement incohérent du dialogue client-serveur avec des TTR classiques.

Pour résoudre ce problème, il faut que les tâches transformées en TTR, c'est-à-dire, les processus instanciés et les serveurs de synchronisations, puissent, en mode écran, effectuer des traitements particuliers (Figure VII.24) :

- Les TTR des processus instanciés doivent, lorsqu'elles effectuent une demande de service à un serveur, se bloquer en attente d'un événement de la tâche d'interface. En cas de réveil, elles transmettent à l'exécutable où le processus instancié qu'elles représentent est actif, les résultats reçus.
- Les TTR des serveurs de synchronisations reçoivent le message et réveillent la tâche d'interface concernée.

La gestion des attentes temporisées est résolue trivialement puisqu'un client en mode écran ne prend jamais la décision d'annuler une demande. Si la temporisation se termine trop tôt, c'est également le cas dans l'exécutable du client actif qui effectue alors une demande d'annulation. Deux cas sont à envisager :

- Le serveur refuse l'annulation : la réponse transite par le client en mode écran qui se reconnecte indéfiniment à la tâche d'interface. Lorsque la réalisation survient, il transmet le résultat à l'exécutable ou le client qu'il représente est actif.
- Le serveur accepte l'annulation : le client en mode écran se contente de faire suivre la réponse.

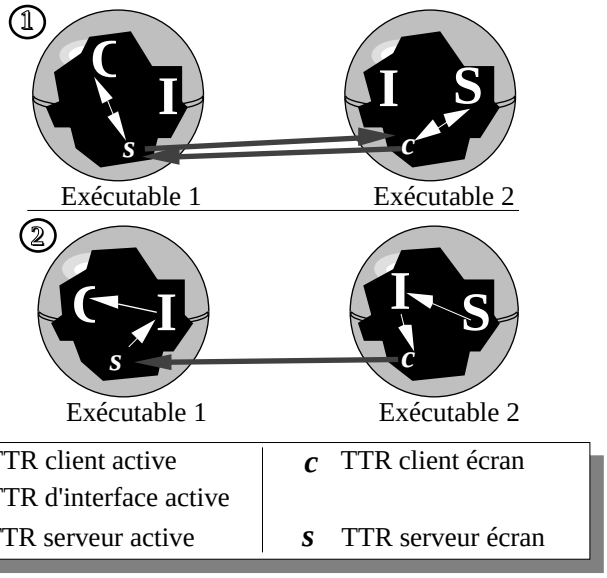


Figure VII.24 : Protocole modifié de transmission du résultat d'un service.

Si le client et le serveur sont situés dans le même exécutable, le mécanisme de communication reste le même que dans le prototype centralisé.

4.3.2. La gestion délocalisée des Ressources

Pour assurer la gestion délocalisée des Ressources par plusieurs serveurs nous considérons les points suivants :

- l'estampillage des demandes : nous devons ordonner des demandes émanant de sites différents afin d'éviter les phénomènes de famine;
- la description des Ressources : nous devons trouver une représentation, ainsi que des protocoles de consultation, permettant d'assurer l'intégrité du marquage;
- la production d'un marquage postcondition : cette opération peut être effectuée composante par composante;
- l'évaluation d'un marquage précondition : une telle opération nécessite la connaissance du marquage de l'ensemble des Ressources référencées;
- la gestion des Ressources externes : leur utilisation doit être transparente, quelles que soient les contraintes de placement auxquelles elles sont soumises.

Estampillage des demandes

Une précondition est toujours estampillée : cela permet, en cas d'échec d'une première évaluation, de réveiller la plus ancienne demande lorsqu'un événement positif se produit. Centraliser la gestion des estampilles est absurde : cela introduit un goulot d'étranglement inutile. Chaque exécutable gère une estampille locale. Composée avec le numéro de l'exécutable, nous obtenons une estampille unique [Lamport 78] :

Numéro de l'exécutable	Estampille locale

En cas d'égalité de deux estampilles locales, le numéro d'exécutable est utilisé pour ordonner deux demandes.

Afin de préserver l'équivalence des estampilles sur chaque site, le serveur de Ressources remet à jour son compteur si l'estampille du message reçu possède une valeur supérieure à celle du site. Cette opération est effectuée dès réception d'un message provenant d'un autre serveur de Ressources.

Description des Ressources

Le problème posé par la répartition de la gestion des Ressources est leur accès. Dans notre cas, il est soumis aux contraintes suivantes :

- exclusion mutuelle : la modification du marquage des Ressources ne peut se faire qu'en section critique;
- équité : les demandes d'évaluation et de production doivent être traitées dans un temps fini;
- évaluation groupée : l'évaluation d'une précondition implique la consultation (et, par conséquent, le verrouillage) de l'ensemble des Ressources qui la composent.

Ces problèmes sont classiques des bases de données réparties [Bernstein 80, Marzullo 91]. Il existe plusieurs types de solutions :

- l'utilisation de copies multiples [Bernstein 78, Birman 85, Garcia-Molina 85, Kumar 91, Raynal 91],
- l'utilisation de serveurs de données responsables de certaines ressources,
- l'utilisation de jetons représentant la donnée [Le Lann 77, Lamport 78].

Les solutions basées sur l'existence de copies multiples sont en général adoptées pour résoudre des problèmes de tolérance aux pannes, ou lorsque les durées de communication sont trop importantes. Dans notre cas, chaque exécutable possède une représentation de l'ensemble des Ressources. Lorsqu'un serveur effectue une mise à jour, il la diffuse à l'ensemble de ses homologues.

Des mécanismes de type électif [Garcia-Molina 85, Kumar 91, Raynal 91], ou basés sur un ordonnancement des estampillages [Lamport 78, Garcia-Molina 89] permettent de préserver l'intégrité du contenu des Ressources. De telles solutions ne sont pas adaptées dans notre cas : soit elles impliquent de "défaire" des Actions indûment activées, soit le nombre de messages transmis devient trop important.

L'utilisation de serveurs responsables de la gestion du contenu de Ressources n'est pas non plus satisfaisante. En effet, l'évaluation d'une précondition suppose la connaissance de *tout* son marquage. Un mécanisme de verrouillage deux-phases [Eswaran 76] doit être considéré. Sa mise en œuvre est excessivement complexe. Le nombre de messages échangés risque d'être énorme et les performances médiocres :

- Si un serveur accède à plusieurs Ressources disposées sur différents exécutables, il bloque leur accès, le temps d'une évaluation.

- La gestion des événements externes peut relancer des évaluations. Ainsi, outre le marquage des Ressources dont il a la charge, chaque serveur doit enregistrer les demandes non encore évaluées en attente d'un événement positif sur des Ressources dont il a la charge.
- Les suppressions des demandes d'évaluation doivent être gérées dans l'ensemble des exécutables, après une annulation de la part d'un client.

La résolution de l'ensemble de ces contraintes ne peut se faire qu'au prix d'un protocole excessivement coûteux en messages. Nous écartons donc cette deuxième solution.

Les contraintes de gestion des Ressources imposent l'existence d'un unique exemplaire accessible, en lecture comme en écriture, depuis différents exécutables. Ainsi, une ressource est représentée par un jeton sans "domicile fixe" [Le Lann 77, Lamport 78]. Un serveur est considéré comme responsable d'une Ressource si le jeton est présent dans son exécutable. Les jetons décrivant les Ressources se promènent au gré des demandes qui les concernent.

Dans le prototype réparti, chaque Ressource est représentée par un *Jeton R* qui contient son marquage et l'ensemble des demandes qui la concerne. Le serveur qui possède un jeton R a le privilège de modifier le marquage de la Ressource. L'estampillage global des demandes assure l'absence de famine des serveurs pour des jetons R.

Lorsqu'un serveur à besoin d'une Ressource qui n'est pas présente sur son site, il diffuse une demande à ses homologues. Tout serveur recevant une demande l'ignore s'il n'est pas détenteur du jeton R correspondant. Dans le cas contraire, le jeton est transmis au serveur dont l'estampille est la plus prioritaire. Le mécanisme est le même pour l'ensemble des requêtes n'impliquant pas le déplacement du jeton (demande d'annulation, production d'une postcondition).

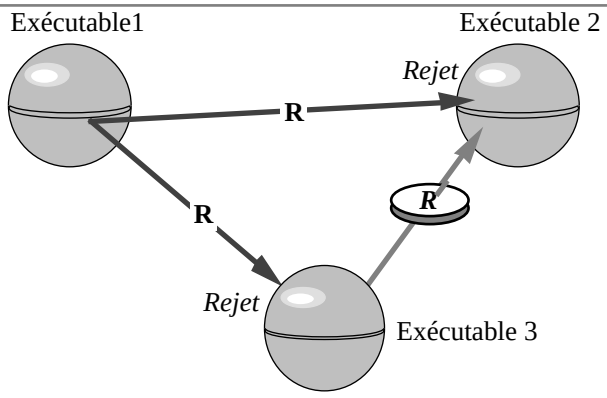


Figure VII.25 : Perte d'une requête concernant la Ressource R pendant le déplacement du jeton la décrivant.

Un jeton R est toujours sous la responsabilité d'un serveur sauf lorsqu'il est en cours de transmission : aucun serveur ne possède la Ressource. Si une requête survient, tous l'ignorent et elle est perdue (Figure VII.25).

Cela justifie l'existence d'un second jeton : le jeton T qui accorde à son possesseur (et qui doit auparavant posséder le jeton R correspondant) le privilège de transmettre le jeton R au serveur qui possède la demande la plus prioritaire. Ce jeton permet de rediriger les demandes susceptibles d'avoir été ignorées durant le trajet du jeton R de la Ressource. Les serveurs doivent respecter les règles suivantes :

- Un serveur de Ressources ne peut transmettre le jeton R correspondant à une ressource que s'il possède également le jeton T associé.
- Un serveur de Ressources reste responsable d'une Ressource jusqu'à réception d'un accusé de réception du jeton R. Il transmet alors le jeton T correspondant.
- Un serveur ne possédant que le jeton T d'une Ressource stocke dans ce dernier toutes les requêtes concernant la Ressource. Elles seront transmises avec le jeton.
- Un serveur ne possédant que le jeton R doit traiter les requêtes concernant la Ressource associée mais en conserver la trace. Dès réception du jeton T, il fait le tri de celles qui ont été perdues ou déjà réalisées.

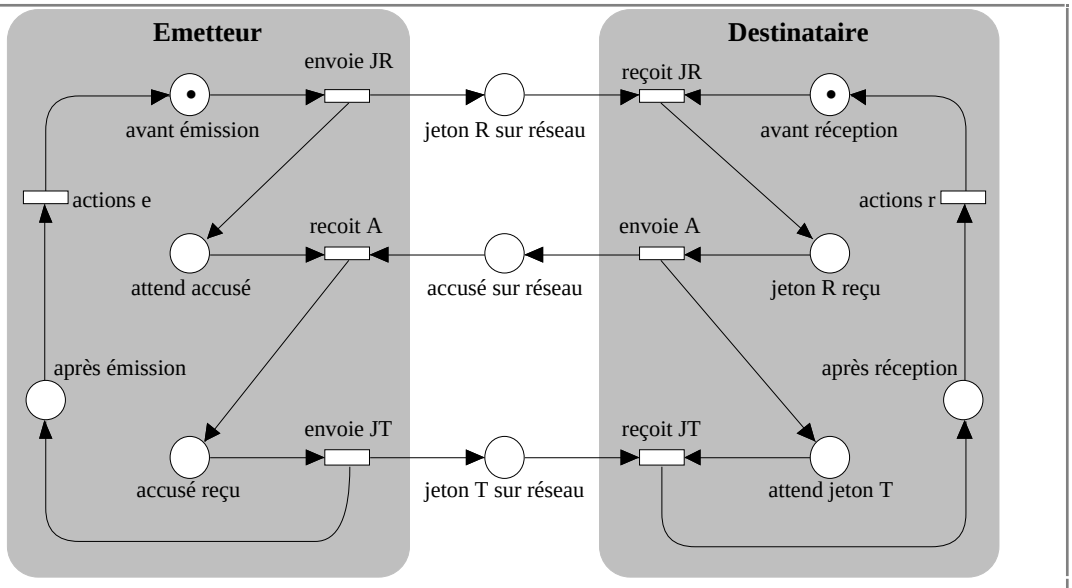


Figure VII.26 : Réseau de Petri modélisant le protocole de transmission d'une Ressource.

L'échange de messages permettant le déplacement d'un jeton R est décrit à l'aide du modèle de la Figure VII.26. Si les communications sont fiables, un serveur recevant le jeton R d'une Ressource peut le retransmettre au bout d'un temps fini.

Exemple VII.8 : Considérons le transfert présenté en Figure VII.27 : les demandes D1, D2 et D3, survenues avant l'émission du jeton T, sont redirigées par le serveur émetteur. Ici, à réception du jeton T, le serveur destinataire ignore D3 qu'il a reçu et traité mais traite D1 et D2 qu'il a ignorées. Dès réception du jeton T, il devient seul responsable de la Ressource et peut à nouveau émettre le jeton R associé.

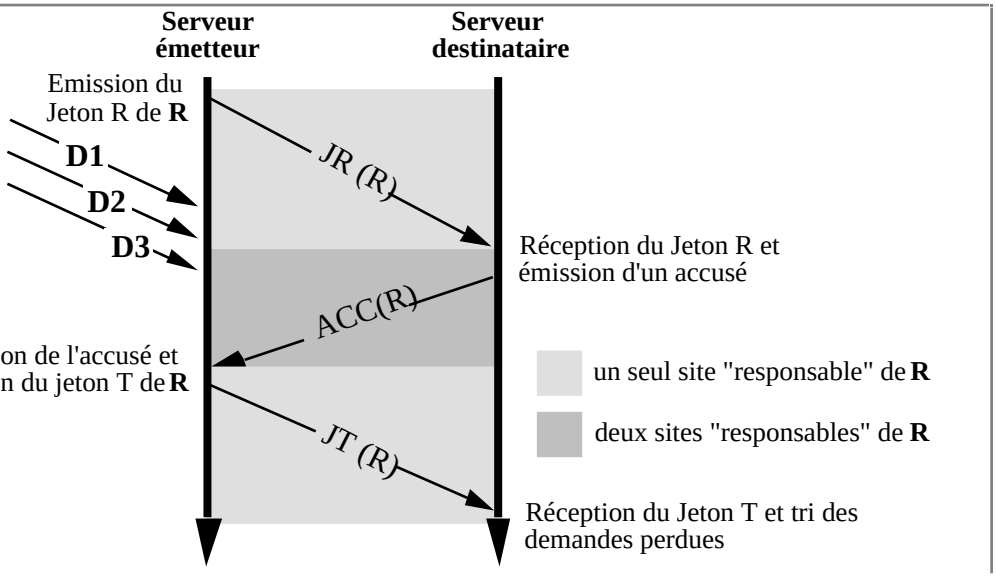


Figure VII.27 : Exemple de transmission d'une Ressource entre deux serveurs.

Evaluation d'une précondition

Les serveurs de Ressources gèrent des descripteurs pour chacune des préconditions qu'ils évaluent. Le descripteur est le même que pour un serveur centralisé.

Pour évaluer une précondition ressource, un serveur doit rapatrier l'ensemble des jetons associés aux Ressources internes référencées. Ainsi, l'échec d'une évaluation peut survenir dans plusieurs cas :

- Si le serveur ne possède pas tous les jetons R : il diffuse alors les demandes correspondantes à ses homologues. La précondition est évaluée lorsque tous les jetons sont arrivés.
- S'il y a défaut de marquage dans une Ressource, ou incompatibilité des marques disponibles : la précondition ne ne peut être réévaluée qu'après un événement positif.

Des interblocages peuvent survenir lorsque l'évaluation d'une précondition échoue. Ils sont de deux types :

- l'interblocage par échec d'évaluation (Exemple VII.9);
- l'interblocage indirect par échec d'évaluation (Exemple VII.10).

Exemple VII.9 : Soient N Processus communiquant de manière asynchrone via un ensemble de Ressources selon la configuration indiquée en Figure VII.28. Chaque Processus est placé dans un exécutable différent; leurs demandes sont donc traitées par des serveurs différents. Supposons que l'ordonnancement des requêtes d'évaluation soit tel que les estampilles des préconditions ressources des Actions T1 à TN respectent la relation suivante[17] :

$$i1 > i2 > i3 ... > iN$$

Conformément à l'algorithme d'émission des jetons R, les Ressources $R_{a(N-1)}$ et $R_{b(N-1)}$ sont transmises vers le serveur$_N$. L'évaluation échoue par défaut de marquage dans

[17] Plus la valeur de l'estampille est faible, plus la demande à laquelle elle est associée est prioritaire.

$R_{b(N-1)}$. Si le serveur$_N$ ne relâche pas les jetons qu'il détient, le système se bloque car aucun serveur ne peut rassembler les jetons nécessaires pour l'évaluation des préconditions.
Pourtant, toutes les Actions composant un tel modèle sont activables si l'on considère un ordre d'évaluation global adéquat. Nous appelons cet état *interblocage par échec d'évaluation*.

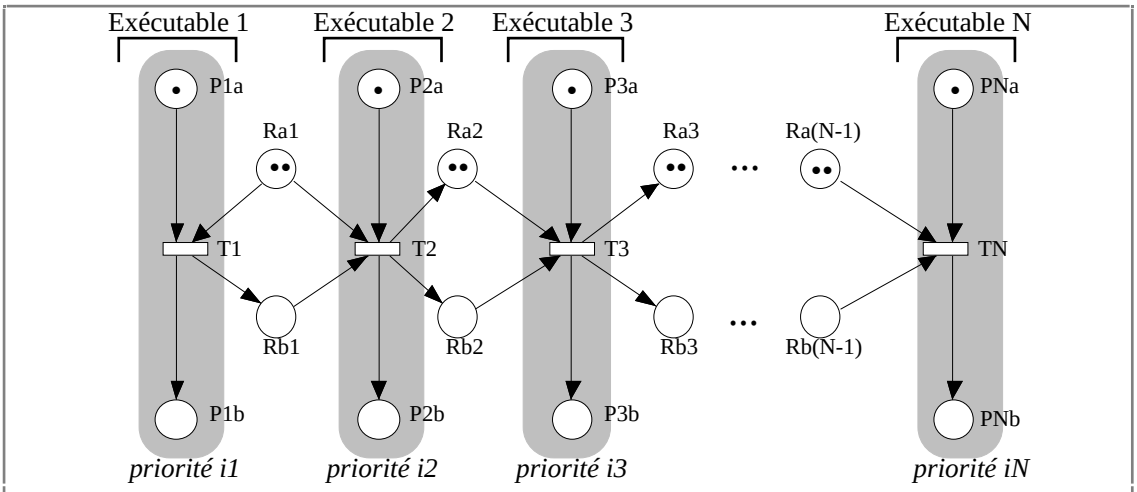


Figure VII.28 : Configuration susceptible de provoquer un interblocage par échec d'évaluation.

Exemple VII.10 : Soient N Processus communiquant de manière asynchrone via un ensemble de Ressources selon la configuration indiquée en Figure VII.29. Chaque Processus est placé dans un exécutable différent; leurs demandes sont donc traitées par des serveurs différents. Supposons que l'ordonnancement des requêtes d'évaluation soit tel que les estampilles des préconditions ressources des Actions T1 à TN respectent la relation suivante :

$$i1 > i2 > i3 \ldots > iN$$

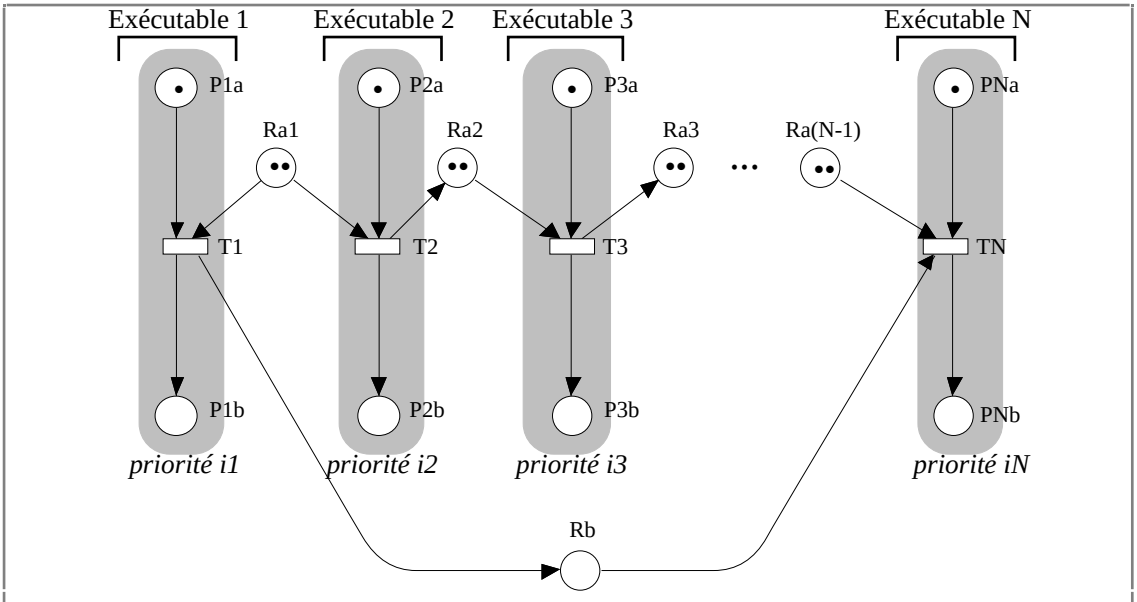


Figure VII.29 : Configuration susceptible de provoquer un interblocage indirect par échec d'évaluation.

Conformément à l'algorithme d'émission des jetons R, les Ressources $R_{a(N-1)}$ et R_b sont transmises vers le serveur$_N$. L'évaluation échoue par défaut de marquage dans R_b. Si le serveur$_N$ ne relâche pas les jetons qu'il détient, le système se bloque car aucun serveur ne peut rassembler les jetons nécessaires pour l'évaluation des préconditions.

Pourtant, toutes les Actions composant un tel modèle sont activables si l'on considère un ordre d'évaluation global adéquat. Nous appelons cet état *interblocage indirect par échec d'évaluation*.

Ces deux types d'interblocage sont liés au fait que le serveur détenant un jeton R ne peut le transmettre car il possède une demande (dont l'évaluation a échoué) de priorité élevée. Seul un événement positif, ou une demande d'annulation, peut débloquer le système, ce qui n'est pas toujours le cas.

La solution consiste à *inhiber* les demandes de jeton pour lesquelles une évaluation a échoué. Une demande inhibée cesse d'être prise en compte jusqu'à ce qu'un événement positif survienne pour l'une des Ressources responsables de l'échec.

Ainsi, un serveur transmet le jeton R associé à une Ressource au serveur possédant la demande non inhibée de priorité la plus élevée.

Lorsqu'un serveur réalise un événement positif sur une Ressource, il réactive les demandes référencées dans le jeton R et, pour provoquer le rapatriement de toutes les ressources référencées dans les préconditions inhibées, diffuse un message de réactivation.

Chaque serveur recevant ce message consulte les demandes inhibées des jetons qu'il possède et, le cas échéant, les réactive avant de tenter d'émettre le jeton R.

Exemple VII.11 : Considérons la configuration décrite dans Exemple VII.9. Lorsque l'évaluation de la précondition associée à TN échoue, $serveur_N$ inhibe la demande sur les jetons de $R_{a(N-1)}$ et $R_{b(N-1)}$. Le jeton R de $R_{b(N-1)}$ est donc transmis au $serveur_{N-1}$ qui tente à son tour une évaluation qui échoue... ceci jusqu'à ce que $serveur_1$ possède le jeton de R_{a1}. L'événement positif sur R_{b1} provoque la réactivation des demandes la concernant. Le $serveur_2$ récupère les jetons associés à R_{a1} et R_{b1}; il peut enfin évaluer positivement T2. Le processus se poursuit de proche en proche jusqu'à ce que l'ensemble des demandes soient résolues.

Exemple VII.12 : Considérons la configuration décrite dans Exemple VII.10. Lorsque l'évaluation de la précondition associée à TN échoue, $serveur_N$ inhibe la demande sur les jetons de $R_{a(N-1)}$ et R_b. Le jeton R de $R_{a(N-1)}$ est donc transmis au $serveur_{N-1}$ qui, lui-même inhibe sa demande concernant $R_{a(N-2)}$ ... ceci se reproduit, de proche en proche, jusqu'à ce que la précondition ressource de T1 se réalise. L'événement positif sur R_{a1} et R_b provoque la réalisation de T2 et de TN. Ensuite, les réactivations se propagent au gré de l'activation des Actions.

Production d'un marquage postcondition

La production de marques ne pose pas de problème particulier. Elle est décomposable en requêtes unitaires transmissibles au serveur détenant le jeton R. Plusieurs cas sont à envisager lorsqu'un module client demande la production d'un marquage postcondition :

- Le Jeton R est présent : le serveur traite la demande et réactive les demandes de jetons réveillées par l'événement positif. Ensuite, il diffuse à ses homologues des demandes de réactivation pour les préconditions réveillées afin que les composantes référencées puissent revenir sur le site d'où provient la demande d'évaluation.

Le jeton R est envoyé si le serveur possède le jeton T correspondant et si la demande la plus prioritaire non inhibée concerne un autre exécutable.

- Le serveur ne possède que le jeton T : la demande est estampillée, puis déposée dans le jeton T.
- Le serveur ne possède aucun jeton : la demande est estampillée, puis diffusée à l'ensemble des serveurs de Ressources. L'estampille permet à un serveur recevant un jeton T d'identifier les productions inutilement redirigées.

Lorsqu'un serveur reçoit une demande de production provenant de l'un de ses homologues, il doit, selon les cas, soit ignorer la demande (il ne possède aucun des jetons), soit mettre à jour le jeton R (il possède le jeton R) soit la rediriger dans le jeton T.

Annulation d'une demande d'évaluation

Lorsqu'un serveur accepte l'annulation de l'évaluation d'une précondition demandée par un module client, il supprime les estampilles correspondantes sur l'ensemble des jetons R concernés. Si certain d'entre eux ne sont pas en sa possession, il diffuse l'annulation à ses homologues, afin d'éviter des déplacements inutiles.

S'il ne possède que le jeton T de la Ressource, l'annulation est insérée dans la liste des événements perdus. Elle sera transmise avec le jeton T.

Lorsqu'un serveur reçoit un message d'annulation provenant de l'un de ses homologues, il traite les demandes concernant les jetons en sa possession et ignore les autres.

Réception d'un message de réactivation

Lorsqu'un serveur reçoit un message de réactivation provenant de l'un de ses homologues, il met à jour les demandes sur les jetons concernés en sa possession. Une telle demande ne provient jamais d'un module-client.

Description des jetons R et T

Le jeton R d'une Ressource interne contient les informations suivantes :

- le nom de la Ressource qu'il décrit;
- son marquage;
- la liste des demandes de jeton, inhibées ou non, classées par estampille;
- la liste des demandes d'évaluation en attente d'un événement positif pour la Ressource;
- la liste des productions effectuées en l'absence du jeton T : cette liste permet de détecter les demandes de production réellement perdues pendant le déplacement du jeton. Lorsque le jeton est émis, cette liste est toujours vide.

Le jeton T d'une Ressource interne contient les informations suivantes :

- le nom de la Ressource à laquelle il est associé;

- les demandes d'évaluation redirigées;
- les demandes de production redirigées;
- les demandes d'annulations redirigées;
- les demandes de réactivation redirigées.

Les Ressources externes

Nous ne considérons ici que les Ressources externes associées aux composants externes non-réversibles.

Les restrictions applicables nous permettent de les traiter de façon particulière :

- Les primitives associées à un composant externe non-réversible sont gérées, au niveau d'un exécutable donné, par le serveur de Ressources concerné.
- La production d'une marque dans une Ressource externe en entrée respecte les règles énoncées pour les Ressources. Les demandes sont transmises au serveur chargé de la gestion du composant externe non-réversible concerné.
- Les restrictions sur les Ressources externes en sortie permettent la consommation de marques à distance. En effet, une précondition n'en référence jamais plus d'une seule. Par diffusion, un serveur demande à ses homologues de lui fournir une marque "puisée" dans la Ressource externe. Le mécanisme ressemble à celui décrit pour les demandes de jeton : le serveur assurant la gestion des Ressources externes répond.

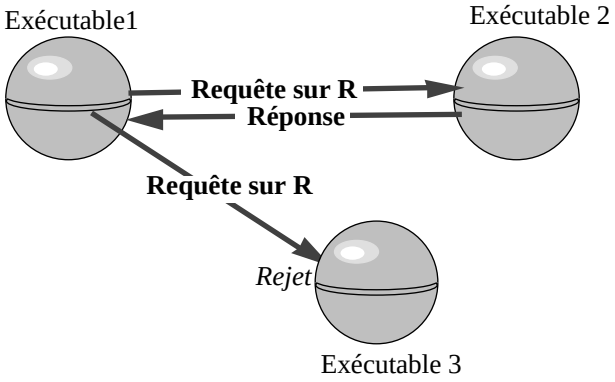


Figure VII.30 : Exemple de traitement réparti de la consommation d'une marque dans une place d'interface en sortie d'un composant externe non-réversible.

Exemple VII.13 : Considérons un prototype réparti sur trois exécutables (Figure VII.30). Le serveur de Ressources contenu dans l'exécutable 1 reçoit une demande d'évaluation du marquage contenu dans une interface en sortie d'un composant externe non-réversible dont il n'est pas responsable. Il diffuse la demande à ses homologues. L'exécutable 2, qui est concerné, la traite puis renvoie un résultat alors que l'exécutable 3 l'ignore.

Une remarque concerne la gestion des événements positifs liés aux Ressources externes. Ils sont signalés à l'ensemble des serveurs de Ressources à l'aide d'un message de réactivation.

4.3.3. Services offerts par le nouveau gestionnaire des Ressources

La Figure VII.31 récapitule l'ensemble des services offerts par le nouveau gestionnaire des Ressources. Les services déjà décrits dans le Chapitre IV sont indiqués en italique.

Comme nous l'avons souligné dans la section 4.1, la nouvelle spécification constitue une extension naturelle de celle définie en Figure IV.29. Ainsi, en terme de spécification, la gestion délocalisée des Ressources se résume à un enrichissement (ou héritage en terminologie objet) de la gestion centralisée.

Les services PRODUCTION_DIFFUSEE, DEMANDE_JETON_DIFFUSEE et ANNULATION_DIFFUSEE permettent de transmettre des requêtes ne nécessitant pas le déplacement des jetons associés aux Ressources concernées.

Les services REACTIVATION_DIFFUSEE, ACCUSE_JETON_R, ENVOIE_JETON_R, ENVOIE_JETON_T DEMANDE_SUR_RESS_EXT et REPONSE_POUR_RESS_EXT correspondent à des échanges de contrôle entre serveurs de Ressources.

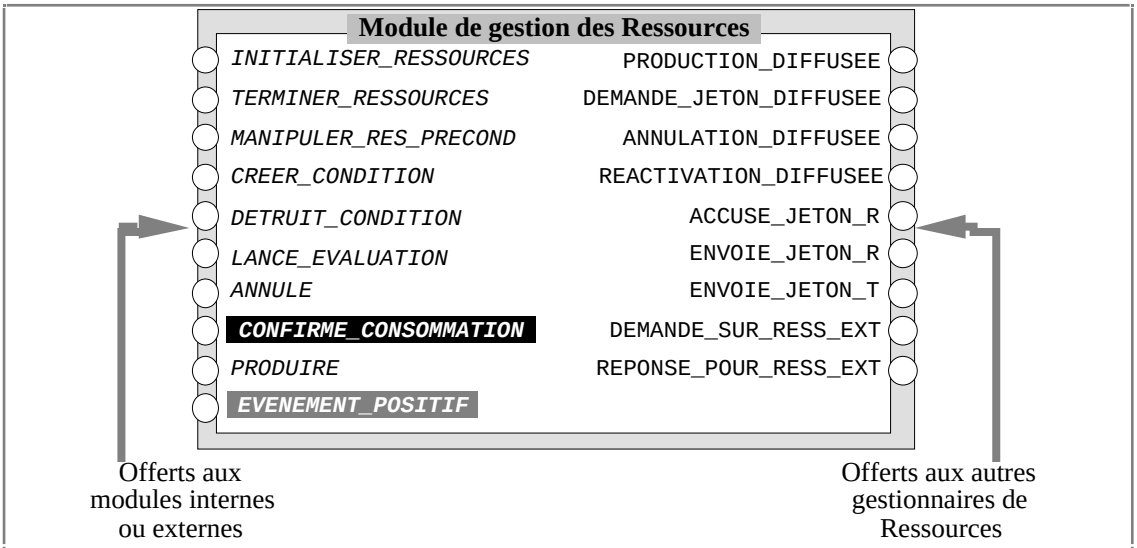


Figure VII.31 : Services offerts par le gestionnaire de Ressources délocalisé.

5. Observations sur le placement

Cette section présente quelques règles de placement permettant d'obtenir un bon facteur d'accélération.

Si nous considérons le prototype réparti en Ada, les objets à répartir sur une architecture sont :

- les processus instanciés du modèle;
- les jetons décrivant les Ressources internes du modèle;
- les serveurs de synchronisations.

En l'absence d'un mécanisme de migration de l'activité des TTR, le placement est effectué a priori par la phase de placement, sur la base d'une description du modèle en termes de processus communicants. Il faut placer correctement les jetons R, les processus instanciés et les serveurs de synchronisations.

L'algorithme de placement optimal doit étudier toutes les combinaisons possibles, en considérant à la fois les contraintes de communication et les durées d'exécution des différentes tâches. Cet algorithme est NP-complet; son application est donc impossible.

Des heuristiques, connaissant la durée d'exécution des tâches et les coûts des communications entre unités fonctionnelles, proposent cependant des solutions satisfaisantes [Billionnet 89].

Dans notre cas, il est difficile d'évaluer la durée d'exécution des tâches composant le prototype qui dépend des traitements associés aux Actions du modèle. L'heuristique développée n'optimise donc que les coûts de communication, c'est-à-dire :

- Le placement des Ressources internes du prototype, sachant que la production de marques peut être réalisée à distance, ce qui n'est pas le cas pour la consommation de marques.
- Le placement des serveurs de synchronisations, sachant que beaucoup de messages peuvent être échangés avec les clients susceptibles d'annuler une demande de connexion.

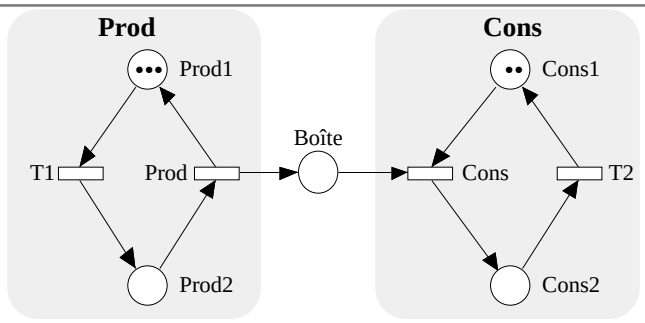
En assimilant les Ressources à des tâches particulières, nous nous ramenons à des méthodes de résolution classiques, basées sur des représentations de l'application sous la forme de graphes ou d'arborescences [Sinclair 87, Lo 88].

Connaissant la structure du prototype réparti, nous définissons quatre règles de placement.

Règle de placement 1 : **Les Ressources avec les consommateurs**

> Dans un modèle producteur-consommateur impliquant des Ressources, il est plus performant de placer les jetons R sur le même site que les Processus consommateurs.

Exemple VII.14 : Considérons le réseau de Petri de la Figure VII.32. Il comporte deux processus communicants : *Prod* produit des données consommées par *Cons*. Il est intéressant de placer la Ressource *Boîte* sur le site contenant les consommateurs. Un placement optimal s'effectue sur deux sites différents : le premier contient tous les processus instanciés de type *Prod*, l'autre ceux de type *Cons* et la Ressource *Boîte*.



Placement des unités fonctionnelles liées aux G-objets

Jeton R et T	Processus instancié	Serveurs de synchronisations	Exécutable
Boîte			Site 1
	Prod		Site 1
	Cons		Site 2

Figure VII.32 : Exemple d'application de la règle de placement N°1.

L'activation d'une Action synchronisée est également un point délicat. Il est intéressant de maximiser les communications locales, surtout s'il y a possibilité d'annulation.

***Règle de placement 2 :* Les serveurs de synchronisations avec les clients pouvant se retirer**

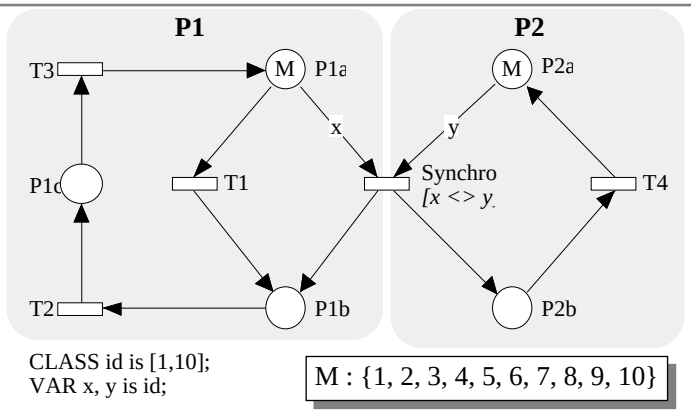
Lorsque N processus se synchronisent via une Action synchronisée, il est préférable de placer le serveur de synchronisations associé sur le même processeur que les clients pouvant effectuer des retraits.

D'une manière générale, il est intéressant de rapprocher le serveur de l'ensemble de ses clients.

Exemple VII.15 : Le prototype associé au réseau de Petri de la Figure VII.33 comporte, outre les tâches dédiées au contrôle du prototype :

- 10 processus instanciés de type *P1*,
- 10 processus instanciés de type *P2*,
- le serveur de l'Action synchronisée *Synchro.*

Comme *P1* peut se retirer, il vaut mieux placer ses instances avec le serveur de *Synchro.* Les instances de *P2* pourront être placées sur un autre site. Si ce réseau de Petri est un sous-ensemble d'un modèle plus important, il sera préférable de placer les instances de *P1* et de *P2* avec le serveur de *Synchro.*



Placement des unités fonctionnelles liées aux G-objets

Jeton R et T	Processus instancié	Serveurs de synchronisations	Exécutable
		Synchro	Site 1
	P1		Site 1
	P2		Site 2

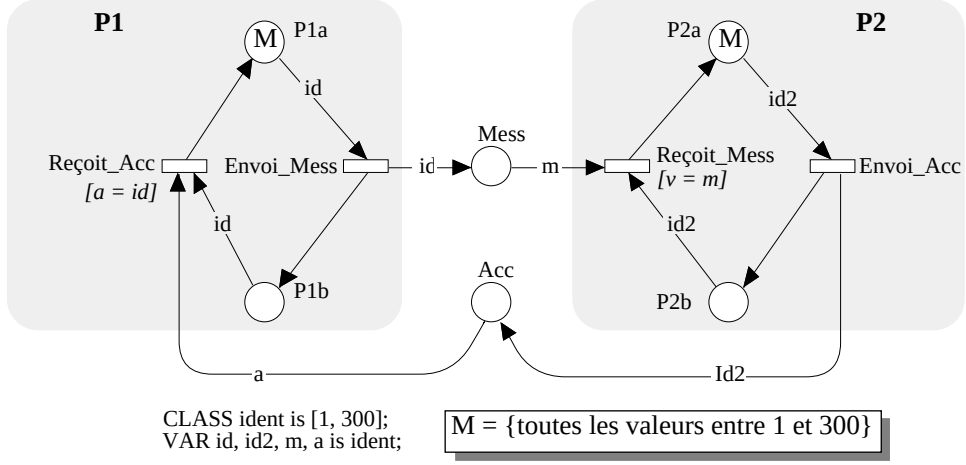
Figure VII.33 : Exemple d'application de la règle de placement N°2.

Certains modèles peuvent se déplier aisément. Ils correspondent à une organisation générique instanciée en fonction d'une couleur. Chaque instance ne communique pas avec ses voisines.

***Règle de placement 3 :* Répartition de modèles ou sous-modèles pliés**

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'y a pas d'interaction entre les différentes "instances", certaines Ressources, ou certaines Actions synchronisées, peuvent être dupliquées.

Exemple VII.16 : Le prototype associé au réseau de Petri de la Figure VII.34 comporte, outre les tâches dédiées au contrôle du prototype, 300 instances des processus *P1* et *P2*. La I^{ème} instance de *P1* communique toujours avec la I^{ème} instance de *P2* : il est donc intéressant de diviser l'application en paquets d'instances de *P1* et de *P2* qui communiqueront entre elles. Ici, les Ressources *Mess* et *Acc* peuvent être instanciées pour chacun des exécutables. Nous donnons, dans la Figure VII.34, un exemple de placement sur deux sites.
Pour ce modèle, nous pouvons également, dans le cas d'un placement sur deux machines, appliquer à ce modèle la règle numéro 1 :
- toutes les instances de *P1* seront placées dans l'exécutable 1, avec la Ressource *Acc*,
- les instances de *P2* seront placées dans l'exécutable 2 avec la Ressource *Mess*.



Placement des unités fonctionnelles
liées aux G-objets

Jeton R et T	Processus instancié	Serveurs de synchronisations	Exécutable
Mess (1)			Site 1
Acc (1)			Site 1
	P1 (1..150)		Site 1
	P2 (1..150)		Site 1
Mess (2)			Site 2
Acc (2)			Site 2
	P1 (150..300)		Site 2
	P2 (150..300)		Site 2

Figure VII.34 : Exemple d'application de la règle de placement N° 3.

La gestion des interfaces avec les composants externes non-réversibles ne pose aucun problème au niveau du placement. Il est évidemment intéressant de regrouper dans un même exécutable les tâches manipulant un composant externe. En cas de conflit avec d'autres contraintes de placement, une solution, certes pas optimale, reste possible.

***Règle de placement 4* : Répartition des composants externes**

Les contraintes de localisation des composants externes peuvent ne pas être prises en compte dans la recherche d'une solution.

Enfin, pour que la répartition soit intéressante, les deux critères suivants doivent être respectés.

Critère 1 : *Décomposition du prototype en composantes.*

Le prototype doit pouvoir être partitionné en composantes faiblement couplées de Processus fortement couplés.
Cela permet de minimiser les communications entre exécutables, plus coûteuses que les communications locales.

Critère 2 : *Durée des traitements associés aux Actions.*

Les délais de communication entre exécutables doivent être faible, ou du même ordre, par rapport à la durée moyenne des traitements associés aux Actions d'un prototype.
Cela permet de relativiser le coût du contrôle du prototype par rapport au traitements qu'il effectue.

6. Expérimentation

A partir d'un prototype centralisé généré automatiquement, nous avons appliqué les principes énoncés dans ce chapitre [Boussand 92].

Dans un premier temps, nous décrivons le réseau de Petri à partir duquel nous avons généré automatiquement (à l'aide de l'outil CPN/TAGADA) le prototype centralisé qui a servi de base à notre étude. Nous étudions sa structure en nous attachant au parallélisme potentiel et commentons les performances que nous avons mesuré.

6.1. LE MODÈLE À PROTOTYPER

Pour définir le modèle permettant de concrétiser nos études sur la répartition par une expérimentation, nous nous sommes posé le problème décrit ci après.

Exemple VII.17 : Considérons la chaîne d'assemblage d'une carrosserie automobile (Figure VII.35). 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.

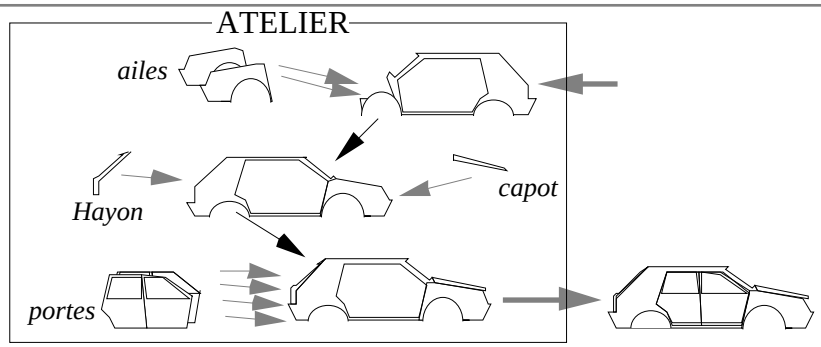


Figure VII.35 : Etapes de construction d'une carrosserie automobile.

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 des 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).

L'ensemble du système est modélisé par le réseau de Petri de la Figure VII.36. Le modèle présenté s'arrête automatiquement après production de 50 carrosseries, soit 50 capots et hayons, 100 ailes et 200 portières. Son exécution complète implique l'activation de 1819 transitions

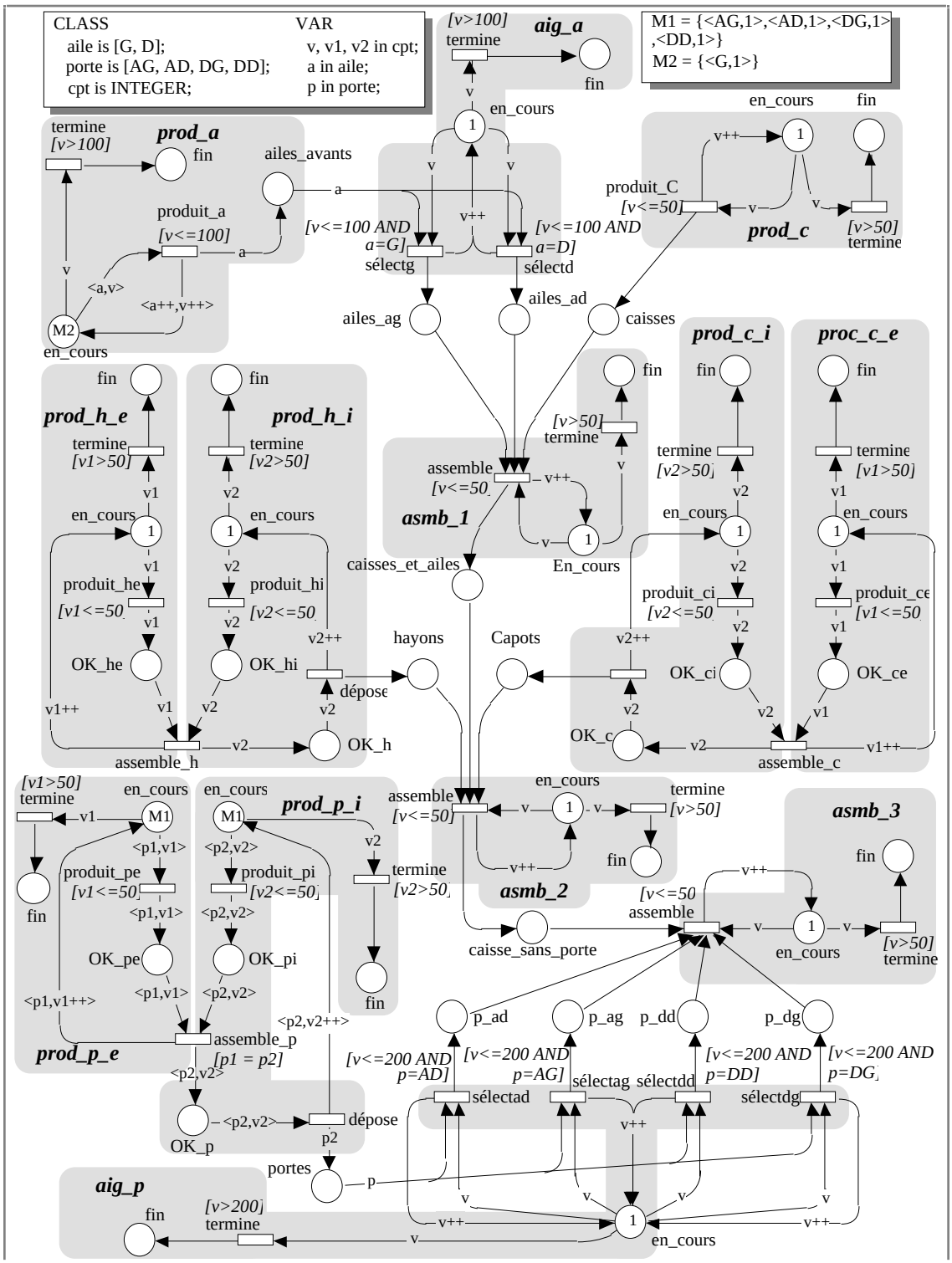


Figure VII.36 : Modélisation de l'atelier de montage de carrosseries automobiles.

Le modèle décrivant le comportement de l'atelier d'assemblage est décomposable en 13 Processus. 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*).

Le prototype centralisé, généré automatiquement, comporte 33 tâches Ada :

- le gardien,
- trois serveurs de synchronisations associées aux Actions *assemble_h*, *assemble_c* et *assemble_p*,
- le serveur de Ressources,
- une tâche par Processus pour *prod_a*, *aig_a*, *prod_c*, *prod_h_e*, *prod_h_i*, *asmb_1*, *prod_c_i*, *prod_c_e*, *asmb_2*, *asmb_3* et *aig_p*,
- quatre tâches pour les Processus *prod_p_e* et *prod_p_i*,
- dix-neuf tâches d'interface associées aux processus-instanciés du prototype.

Nous pouvons faire différentes observations *a priori* sur la structure du prototype obtenu. Ces observations constituent des informations importantes si l'on veut obtenir un placement efficace et, par conséquent, un gain optimal.

L'idée est de découper le prototype en "composantes" de tâches Ada au sein desquelles le tau de communication est élevé. Les communications entre les différentes composantes sont moins fréquentes.

Observation a priori N° 1 : Conformément à la règle de placement 2, les trois Actions synchronisées et leurs clients doivent, si possible être regroupés dans le même exécutable. Nous obtenons ainsi trois composantes regroupant :
- le serveur de *assemble_h*, *prod_h_i* et *prod_h_e*,
- le serveur de *assemble_c*, *prod_c_i* et *prod_c_e*,
- le serveur de *assemble_p*, et les processus instanciés de *prod_p_i* et *prod_p_e*.

Observation a priori N° 2 : Considérons la première approche de la répartition. La gestion des Ressources est centralisée dans l'exécutable ou le serveur de Ressources est Actif. Ici, les précondition Ressources portent sur des Actions qui sont systématiquement postcondition d'Etats_Processus alternatifs. Des demandes d'annulation sont à prévoir : le tau de communication entre les Processus "consommateurs" et le serveur de Ressource est élevé. Nous déduisons ainsi une quatrième composante dans le prototype. Elle regroupe le serveur de Ressources, les Processus *aig_a*, *asmb_1*, *asmb_2 asmb_3* et *aig_p*. Le placement des Processus *prod_a* et *prod_c* ne dois pas influer de façon importante sur les performances.

Observation a priori N° 3 : Considérons la seconde approche de la répartition. Il existe un serveur de Ressource par exécutable et le placement des jetons importe peu. En effet, Chaque Ressource est liée à un unique Processus "consommateur". Ainsi, les jetons initialement mal placés, ils seront rapatriés dans l'exécutable ou se situe le consommateur dès la première demande d'évaluation d'une précondition. Par ailleurs, nous pouvons caractériser un pipe-line de 5 étages :

étage 1 ⊳ prod_a, prod_c, prod_h_e, assemble_h, prod_h_i, prod_c_i, assemble_c et prod_c_e, prod_p_e, assemble_p, prod_p_i;
étage 2 ⊳ aig_a, aig_p;
étage 3 ⊳ asmb_1
étage 4 ⊳ asmb_2
étage 5 ⊳ asmb_3

Sur la base de ces observations, nous pouvons déduire, en fonction des approches, les effets suivants :

- Le gain optimal, pour la première approche, devrait être obtenu lorsque le prototype s'exécute sur quatre machines. Au delà, il faut envisager de

"casser" une composante et le coût du contrôle induit risque de devenir élevé.

- Le gain optimal, pour la seconde approche, doit pouvoir être obtenu avec plus de machines.

6.2. LES MESURES

Nous avons réparti "à la main", le prototype centralisé généré pour le modèle de la Figure VII.36 [Boussand 92].

Nous avons associé à chaque Action une procédure effectuant un calcul arithmétique flottant d'une durée d'environs 0,5 secondes CPU. Comme l'exécution de ce modèle implique 1819 activations de transitions, la durée théorique d'exécution sur une machine est d'environs 910 secondes.

Nous avons ensuite étudié le comportement de l'application répartie ainsi obtenue sur un ensemble homogène de machines - des SparcStation 1 à 8 Mo de mémoire centrale connectées en réseau Ethernet - dédiées au fonctionnement de l'application.

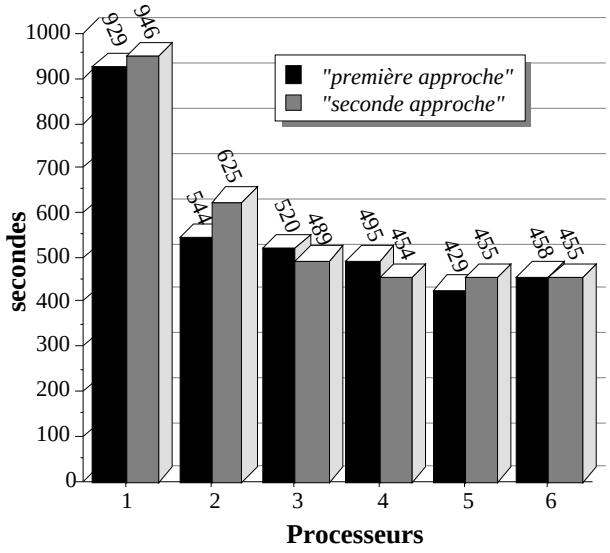


Figure VII.37 : Exécution comparée des deux approches de la répartition en fonction du nombre de machines utilisées.

La Figure VII.37 indique, pour chacune des approches de la répartition, les performances obtenues. La Figure VII.38 donne la courbe des gains observés en fonction du nombre de machines dédiées à l'application. Enfin, la Figure VII.39 indique l'évolution du nombre de messages échangés entre exécutables. Dans les trois cas, il s'agit de moyennes sur plusieurs exécutions effectuées pour des placements "optimaux"[18].

[18] C'est à dire, ceux pour lesquels nous avons observé les meilleures performances. Notre méthode de placement étant tout à fait empirique, il n'est pas exclus que de meilleurs résultats soient obtenus.

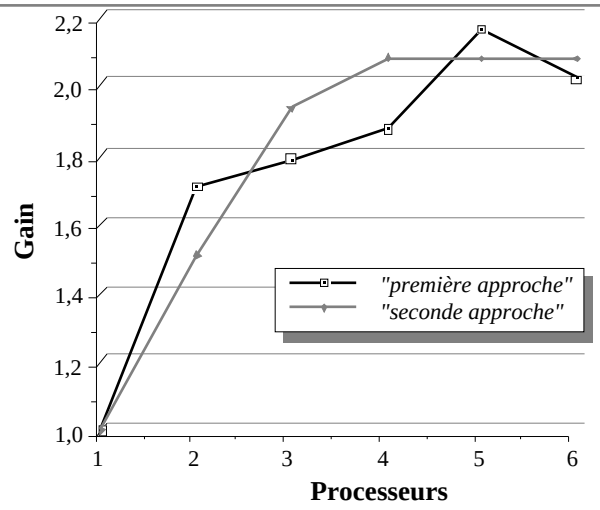


Figure VII.38 : Gain comparé des deux approches en fonction du nombre de machines utilisées.

Notre objectif était de démontrer la faisabilité et l'intérêt de la répartition. Cependant, la production d'un prototype réparti n'étant pas encore automatisée (CPN/TAGADA ne génère que des prototypes centralisés), nous n'avons pu étudier l'impact de la répartition en fonction des différents types de configuration de réseau de Petri.

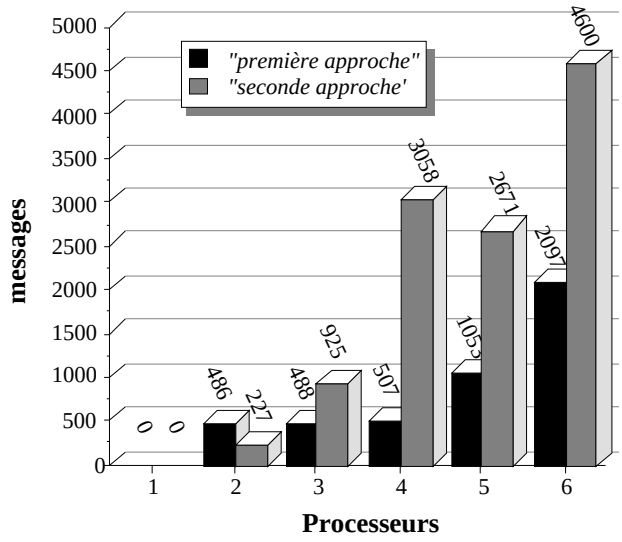


Figure VII.39 : Nombre de messages échangés entre les différents exécutables du prototype réparti.

Les résultats obtenus tendent à montrer que cette approche est intéressante. Nous proposons quelques observations *a posteriori* sur la base des mesures effectuées.

Observation a posteriori N° 1 : Si l'exécution du prototype réparti selon la première approche est meilleure sur cinq machines que sur quatre machines (?), il y a dégradation des performances à partir de six machines. Nous observons cependant que le nombre de messages échangés augmente rapidement dès que l'on passe sur plus de quatre processeurs. Cela est bien du à la nécessaire division d'au moins une composante de tâches. Nous augmentons de la sorte le nombre de rendez-vous distants effectués.

Observation a posteriori N° 2 : Contrairement à ce que l'on pouvait attendre, la seconde approche ne donne pas toujours les meilleurs résultats. Cependant, certains choix d'implémentation, dans la gestion des jetons par les serveurs de Ressources, pourraient être à l'origine des "écarts" observés. Conformément à ce que nous attendions, le nombre de message échangés devient rapidement très élevé.

6.3. QUELQUES REMARQUES

Afin d'obtenir de bonnes performances d'exécution, nous avons pleinement utilisé une approche par raffinement. En effet, sur la base des informations fournies par le prototype réparti, nous avons, par étapes successives, optimisé le placement du prototype.

Les paramètres ayant un impact important sur la durée d'exécution d'un prototype sont différents selon l'approche considérée :

- Pour la première approche, les meilleurs gains sont obtenus lorsque le nombre de message est minimal. Cela s'explique par la relation directe liant le nombre de messages échangés au nombre de rendez-vous distants.
- Pour la seconde approche, les meilleurs temps d'exécution sont systématiquement obtenus pour des placements ne minimisant pas le nombre de messages. Le temps d'exécution des Actions, ainsi que le parallélisme potentiel (défini par le pipe-line) au niveau des Ressources, ont constitué un critère important.

A l'heure actuelle, il semble que les performances des deux approches soient équivalente. Ainsi, le choix de l'une ou l'autre peut dépendre de la structure du prototype (le choix est alors effectué par la phase de placement) ou des directives données par le concepteur du modèle.

Cela n'est cependant vrai qu'à une seule condition : il ne doit pas y avoir de contraintes sur le placement des composants externes manipulés par le modèle. En effet, si tel est le cas, des contradictions peuvent interdire tout placement cohérent du serveur de Ressources.

Exemple VII.18 : Considérons un système manipulant deux composants externes A et B et dont l'exécution s'effectue sur deux processeurs : P1 et P2. Les contraintes suivantes sont définies sur le placement des composants externes :

- A est lié à P1,
- B est lié à P2.

Si l'on considère la première approche, il est impossible de placer le serveur de Ressources. En effet, ce dernier assure le dialogue entre le système et deux dispositifs distants. Seule la seconde approche est envisageable dans un tel contexte puisque, sur chaque Processeur, le serveur de Ressources local assumera les appels au primitives associées aux Ressources externes..

Nous pensons que des informations plus précises pourront être obtenues lorsque le générateur de code produira des prototypes répartis. L'étude des propriétés de modèles prototypés fournira d'importantes informations sur les critères de répartition à prendre en compte et le nombre maximal de processeurs pour lequel un gain est théoriquement possible. De telles

propriétés doivent pouvoir être déduites de l'analyse du réseau de Pétri décrivant le système.

7. Conclusion

Nous avons exposé trois principes permettant la répartition d'un exécutable Ada sur un ensemble de machines connectées en réseau.

L'architecture que nous avons décrite est facilement portable d'un environnement à un autre. Les éléments dépendants du système d'exploitation sont inclus dans une *boîte à outils de répartition* contenant un certain nombre de services permettant la réalisation des principes de répartition. Pour porter l'application, seule cette unité est à réécrire.

Le principe P2, basé sur l'utilisation d'un nouveau composant logiciel (les "Tâches Transparentes à la Répartition") autorise une répartition transparente des tâches sur un nombre quelconque de machines. Il est appuyé par deux autres principes auxiliaires :

- P1 : il spécifie que l'application ne comporte qu'un seul exécutable, contenant l'ensemble des tâches de l'application, lancé sur tous les sites impliqués dans la répartition. Ce principe autorise une grande souplesse dans le placement des tâches et permet d'envisager leur migration à un coût raisonnable.
- P3 : il spécifie le fonctionnement d'un rendez-vous entre tâches Ada distantes, suivant la sémantique du langage.

A partir de ces trois principes, deux architectures sont envisageables en fonction de la nature centralisée ou répartie de la gestion du contexte global de l'application. Une première étude [Kordon 91c] a mis à jour les problèmes engendrés par le contrôle centralisé de l'application.

Nous avons ensuite appliqué notre méthode au prototypage. Deux approches sont proposées :

- une application stricte des principes énoncés, qui pose des problèmes de performance au niveau de la gestion des Ressources;
- une application moins complète, avec la définition d'une nouvelle politique de réalisation du gestionnaire des Ressources autorisant un meilleur parallélisme.

Des expériences ont été menées, comparant les deux approches [Boussand 92]; des mesures du facteur d'accélération montrent que, pour des modèles respectant un certain nombre de contraintes, la répartition selon nos principes est pertinente.

Des extensions du service de répartition sont cependant envisageables :

- Nous avons focalisé notre étude sur la réalisation d'un rendez-vous distant entre tâches Ada. D'autres services ont cependant été décrits en Section 2.3 :
  - création et destruction dynamique de tâches : elles doivent être réalisées par les gardiens; en lieu et place des instructions Ada `new` et `abort`, des primitives équivalentes doivent être proposées

- calcul de la charge d'un site : elle est réalisée par les gardiens qui diffusent à leurs homologues la charge de leur machine à intervalles réguliers. Une *table de charge des sites* doit être maintenue par chacun d'eux.

• La migration de tâches : elle permet une adaptation dynamique du contexte d'exécution en tenant compte de critères de charge et de communication entre les TTR [Goscinski 91].

  La migration ne concerne pas l'ensemble des tâches du prototype : seuls les processus instanciés du modèle et les serveurs de synchronisations y sont soumis.

			Introduction générale
	Chapitre I	:	*Prototypage de systèmes parallèles*
	Chapitre II	:	*Interprétation sémantique d'un réseau de Petri*
	Chapitre III	:	*Méthodologie de prototypage*
	Chapitre IV	:	*Structure du prototype généré*
	Chapitre V	:	*Caractéristiques du prototype Ada*
	Chapitre VI	:	*Le prototype centralisé*
	Chapitre VII	:	*Le prototype réparti*
+	Chapitre VIII	:	*CPN/TAGADA : L'outil de prototypage*
			Conclusion générale
			Bibliographie
			Annexes & Appendices

Chapitre VIII :

CPN/TAGADA : L'outil de Prototypage

1 Introduction 299

2 L'environnement logiciel AMI 301

2.1 La plate-forme de dialogue 301
2.2 L'interface utilisateur Macao 303
2.3 Les outils 305
2.4 Le formalisme de description utilisé 306

3 Le générateur de code 308

3.1 Architecture de l'outil 308
3.2 Hypothèses 309
3.3 Les phases d'Identification et de Caractérisation 312
3.4 La phase de Placement 317
3.5 La phase de Génération 318
3.6 La gestion des composants externes 319

4 Conclusion 324

1 Introduction

Pour valider nos travaux, nous nous sommes attachés à les concrétiser par un produit logiciel. Cette approche pragmatique, reposant sur une réalisation d'outils, constitue, à notre sens, le seul moyen d'évaluer objectivement l'apport de notre contribution.

Les travaux décrits dans ce document ont abouti à la réalisation d'un outil : CPN/TAGADA (Traduction, Analyse, Génération Automatique de code aDA) qui génère automatiquement un prototype Ada. La version actuelle génère un prototype centralisé.

Le projet MARS [Estraillier 91, Estraillier 92], dans lequel s'inscrivent nos travaux (voir Chapitre I, section 1.1), est supporté par l'Atelier de Modélisation Interactive (AMI) [Bernard 88, Bernard 89, Bernard 90]. Dans un premier temps, un cahier des charges a été élaboré à partir des besoins de notre équipe. Nous avons ainsi intégré les fonctionnalités suivantes :

- interface utilisateur standard et conviviale;
- extensibilité des représentations graphiques;
- partage et communication entre logiciels;
- ouvertures pour l'intégration de nouvelles applications;
- conception rapide de nouvelles applications.

Notre générateur de code repose donc sur l'utilisation de cet atelier et bénéficie ainsi des services offerts par l'atelier AMI à toute application intégrée, concernant la gestion de l'interface utilisateur (saisie des données, visualisation des résultats, lancement des traitements), le stockage des données et la communication entre les différents outils. Ces fonctionnalités sont introduites dans le paragraphe suivant.

L'utilisation de AMI permet ainsi de concentrer l'écriture des applications sur la partie algorithmique propre, en se dégageant des problèmes liés à toute infrastructure système (gestion des utilisateurs, stockage des données...). Ainsi, pour notre outil de prototypage, AMI apporte les avantages suivants :

- L'interface utilisateur prend en charge la saisie des modèles réseau de Petri (qu'ils décrivent des composants externes ou la spécification d'un système), des directives de placement et de toute l'interactivité avec le concepteur.
- L'utilisation d'un formalisme homogène - les réseaux de Petri Colorés - autorise l'utilisation combinée des différents outils associés, disponibles sous AMI. Ensuite, la génération un prototype peut être effectuée sans avoir à traduire ou redessiner les spécifications.
- La communication entre applications permet à un outil indépendant du générateur de code d'assurer le calcul de la famille génératrice de semi-flots de support minimal.

Dans un premier temps, nous allons brièvement décrire l'environnement, en particlier la plate-forme système et l'interface utilisateur. Nous introduisons

ensuite brièvement les différents outils actuellement intégrés. Nous présentons ensuite un formalisme assurant une représentation homogène des différentes classes de réseaux de Petri. Enfin, nous détaillons l'architecture et les principes de réalisation de CPN/TAGADA.

2 L'environnement logiciel AMI

L'objectif est de fournir une plate-forme système couvrant à la fois la spécification de systèmes, leur vérification et le développement des logiciels. Dans ce cadre, un énorme travail préalable a essentiellement porté sur l'infrastructure d'un atelier de spécification, c'est-à-dire la structure système accueillant les outils de spécification et l'interface utilisateur [Bernard 90]. Une première version, AMI, intègre :

- Une plate-forme de dialogue : AMI. Elle permet à une application travaillant sur un modèle graphique, de dialoguer avec l'utilisateur afin de réaliser des services. Cette plate-forme fonctionne sur station de travail UNIX™ et utilise un protocole pour échanger des données avec l'interface utilisateur Macao. Les applications reçoivent, de façon normalisée, la représentation d'un graphe et transmettent à l'interface graphique les résultats de leur calcul.
- Une interface utilisateur : Macao. Elle permet de saisir, sur des stations Macintosh, des modèles graphiques quelconques. Les réseaux de Petri constituent l'une des représentations possibles.

2.1 LA PLATE-FORME DE DIALOGUE

L'atelier logiciel AMI s'exécute dans un environnement distribué hétérogène. Les difficultés ont été nombreuses non seulement pour structurer un très gros logiciel (dont le développement atteint maintenant plusieurs dizaines de milliers de lignes de code), mais aussi pour intégrer toutes les contraintes associées aux environnements distribués hétérogènes.

Nous avons adopté une architecture distribuée pour l'atelier (par opposition à une station de travail qui regroupe l'ensemble des composantes).

Dans un souci de performance, nous avons distribué les exécutions sur un parc de machines possédant un système d'exploitation multi-tâches, multi-utilisateurs et multi-sites (système UNIX) muni d'un système de fichiers répartis. En effet, les outils d'analyse des modèles nécessitent une forte puissance de calcul et manipulent de grosses quantités d'informations.

Enfin, puisque nous avions basé cet atelier sur un ensemble de stations de travail hétérogènes connectées en réseau, une des contraintes a été de rendre transparente aux utilisateurs l'exécution des différents outils dans cet environnement distribué.

L'environnement système gère les problèmes de communication, concurrence et coopération entre les exécutions d'outils applicatifs et entre les outils eux-mêmes. Un langage commun de représentation de données facilite la communication des résultats entre des outils coopératifs et l'intégration des nouveaux outils.

La séparation fonctionnelle et physique de l'interface utilisateur offre la possibilité de développer les outils applicatifs en s'affranchissant des problèmes graphiques et d'interactivité.

La réalisation d'interfaces utilisateurs était généralement une tâche fastidieuse, nous avons conçu une interface utilisateur proposant une forme graphique homogène pour la saisie des données, la gestion des menus et des diagnostics, ainsi que pour la visualisation des résultats. Cette normalisation permet de réaliser, de manière efficace, des interfaces particularisables par le développeur. L'interface devient ainsi une donnée et non plus une partie du code de l'application. Le concepteur d'applications n'a alors plus à se préoccuper de la manipulation des données graphiques. Pour intégrer de nouveaux traitements et bénéficier de l'interface utilisateur, il suffit de spécifier les fonctionnalités de chaque application suivant la méthodologie établie.

L'absence de connexion de la station de travail supportant l'interface utilisateur (par choix de l'utilisateur ou résultant d'une impossibilité matérielle) n'empêche pas un mode d'utilisation autonome. L'utilisateur peut ainsi procéder à la manipulation graphique des différents modèles et les stocker localement. Il est alors nécessaire, lors de la reconnexion, d'assurer la cohérence des données.

L'atelier est décomposé en couches qui s'empilent depuis le niveau le plus physique (celui du système d'exploitation) jusqu'au niveau le plus abstrait : celui des applications (programme d'application et interaction utilisateur). Pour ces dernières, l'atelier apparaît bien sûr comme une seule entité, masquant la complexité physique de l'environnement distribué.

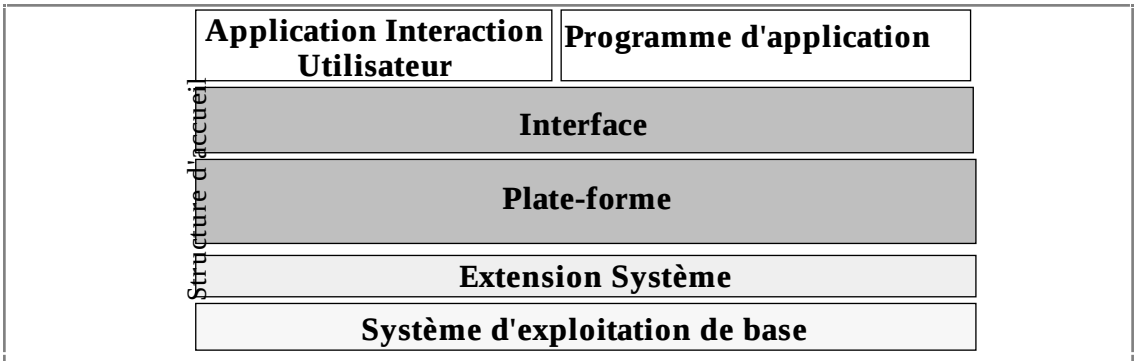


Figure VIII.1 : Architecture de l'atelier AMI.

Notre architecture repose sur un *système d'exploitation de base* (système de fichiers, système de communication). Le niveau *d'extension du système* ajoute des fonctionnalités graphiques et de répartition. Le *niveau plate-forme* contient un squelette d'application pour l'interface utilisateur et les gestionnaires de station de travail virtuel, des utilisateurs et des services. Au dessus du niveau plate-forme se situe le *niveau interface* entre les différentes applications et la plate-forme. Enfin le *niveau application* contient l'éditeur de graphes, des outils d'administration et les applications de calculs (Figure VIII.1).

2.2 L'INTERFACE UTILISATEUR MACAO

Nous avons localisé l'exécution de l'interface utilisateur sur un poste de travail mono-tâche de type Macintosh pour des raisons ergonomiques (interface utilisateur interactive et conviviale, facilité de transport et d'installation), techniques (primitives graphiques spécialisées intégrées, spécialisation du poste de travail dédié) et économiques (large diffusion, faible coût d'achat et de maintenance).

Cette interface est réalisée par le logiciel Macao, qui s'utilise de deux manières :

- en mode autonome : il est possible de créer et manipuler un fichier décrivant un modèle graphique,
- en mode connecté : il est possible d'accéder, via l'atelier, aux applications qui réalisent des services.

2.2.1 Le mode autonome

Lorsque l'utilisateur lance Macao, deux fenêtres apparaissent : une fenêtre d'historique et une fenêtre palette vide.

La *fenêtre d'historique* permet à Macao d'informer l'utilisateur d'un certain nombre d'événements (erreurs, étapes en cours,...). Elle est surtout utile en mode connecté.

La *palette* contient l'ensemble des outils de base de Macao. Son contenu est grisé quand aucune manipulation graphique n'est possible. Cela arrive lorsqu'aucun document n'est ouvert.

Lorsque l'on crée un nouveau document, une zone de dialogue s'affiche. Ce type de cadre apparaît lorsqu'il faut fournir un complément d'informations afin que la commande choisie puisse s'exécuter. En l'occurrence, il faut choisir le *formalisme* de ce nouveau fichier.

Définition VIII.1 : **Formalisme**

> Description d'une représentation formelle. Cette représentation est, dans le cas de Macao, graphique. Il s'agit d'indiquer et de décrire le type de graphes manipulé.

Une fois le formalisme choisi, et le document créé, une fenêtre vide de nom "Sans titre" apparaît; c'est dans celle-ci que l'utilisateur va dessiner son *énoncé*. La palette est mise à jour en fonction des objets du formalisme. Cette palette contient l'ensemble des nœuds et des connecteurs du formalisme ainsi que des opérateurs de base comme l'opérateur de sélection et de manipulation de texte.

L'utilisateur peut changer l'échelle du dessin. Il peut aussi travailler sur le même document à deux échelles différentes ce qui lui permet d'avoir à la fois une idée d'ensemble et un détail de son dessin. Le travail est ainsi possible à n'importe quelle échelle. L'utilisateur peut créer autant de fenêtres qu'il désire sur sa même feuille de travail.

Exemple VIII.1 : Il existe, dans Macao, plusieurs formalismes permettant de représenter, différents types de graphes. Par exemple :

- Le formalisme **réseaux de Petri Ordinaires** : permet de décrire un graphe orienté biparti composé de places (cercles) et de transitions (rectangles). A chaque place, sont associés les attributs *nom* (chaîne de caractères) et *marquage* (entier naturel). A chaque transition est associé un *nom* (chaîne de caractères). Des arcs relient ces nœuds, une *valuation* (entier naturel) leur est associée.
- Le formalisme **Graphes** : il n'existe qu'un seul type de nœud, les sommets auxquels sont associé un nom (chaîne de caractères) et une valeur (entier). Les sommets sont reliés par des arètes auxquelles une *valuation* est associées (chaîne de caractères).

La Figure VIII.2 montre un exemple de modèle dessiné à l'aide de Macao. Le graphe est un réseau de Petri. La palette (en haut à droite) décrit les objets du formalisme réseaux de Petri Ordinaires. Il existe deux classes de nœuds (*place* et *transition*) et une seule classe de connecteur (*arc*). La fenêtre de gauche contient *l'énoncé* : le modèle réseau de Petri. En bas, à droite, se trouve la fenêtre de dialogue.

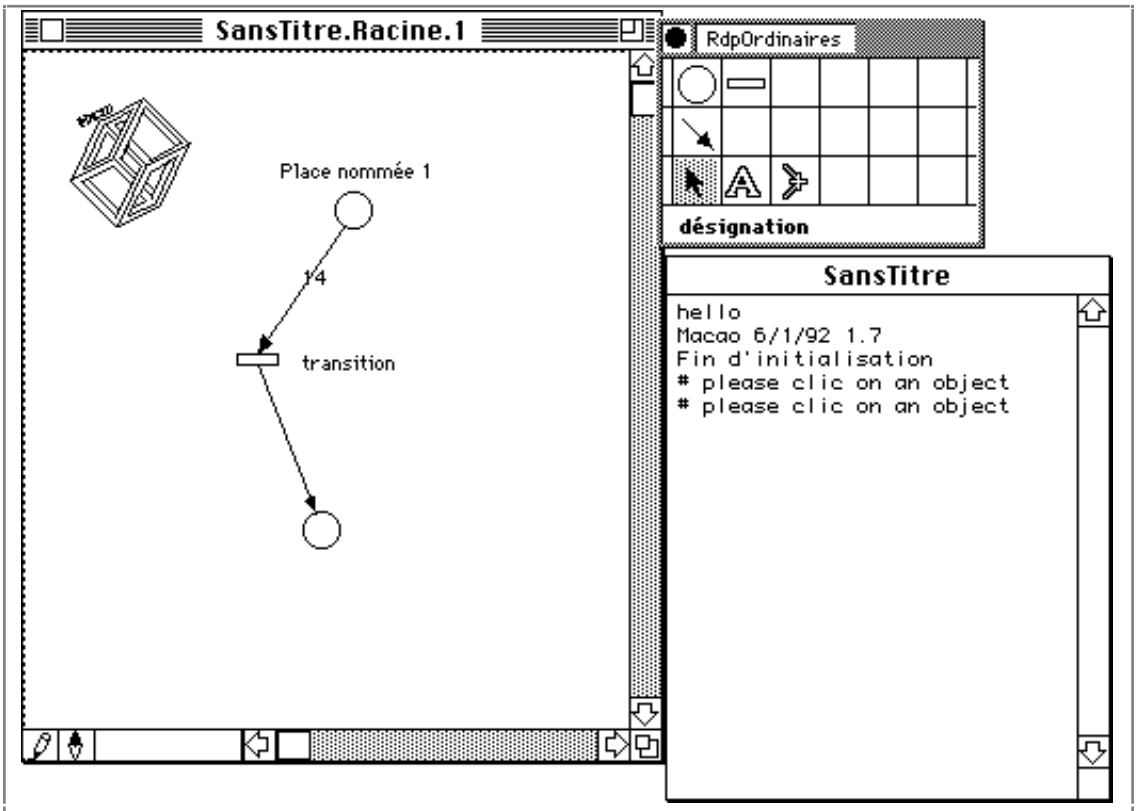


Figure VIII.2 : La palette sous Macao pour les réseaux de Petri ordinaires. La case grisée correspond à l'outil sélectionné.

2.2.2 Le mode connecté

Lorsque l'utilisateur demande une connexion, il commence un dialogue avec l'atelier. L'utilisateur doit être reconnu et identifié (login). A l'ouverture de cette session, AMI recherche la liste des services accessibles à l'utilisateur et l'envoie à Macao qui l'affiche. L'utilisateur peut alors faire son choix.

Le gérant des dialogues de l'atelier AMI transmet la liste des questions que l'on peut poser à l'application, puis il demande d'afficher cette liste. En même temps, il lance l'application correspondante qui s'initialise. Macao réceptionne la liste des questions que lui transmet AMI et affiche un nouveau menu

correspondant à cette application. Ce dernier apparaît grisé tant que l'application est en cours d'initialisation.

La phase d'initialisation de l'application consiste, en plus de phases d'initialisation classiques, à indiquer, pour chacune des questions possibles, si elles ont déjà été posées et si des résultats sont immédiatement disponibles. L'application peut aussi indiquer qu'une question est en cours d'exécution.

Macao réceptionne les mises à jour des questions et rend valide le menu de l'application. L'utilisateur peut alors poser des questions à l'application.

Lorsque l'utilisateur pose une question, la question et ses options sont transmises à la plate-forme AMI qui transmet l'énoncé du problème (le modèle) à l'application chargée de fournir un résultat.

Les résultats d'une question peuvent être de différentes formes:

- Résultats textuels : ils sont affichés dans une fenêtre d'historique.
- Mise en évidence d'objets dans l'énoncé : Macao permet de visualiser des ensembles d'objets comme s'ils avaient été sélectionnés.
- Modification des attributs de l'énoncé initial : des données, propres aux sommets des graphes, sont modifiées.
- Modification de l'énoncé initial : une application peut ainsi créer de nouveaux objets, ou supprimer des objets du modèle initial.
- Création d'un nouvel énoncé : dans ce cas, Macao ouvre une fenêtre de résultat et les objets créés sont mis dans la nouvelle fenêtre. Ces résultats peuvent être dans un autre formalisme.

En fin de question, la fenêtre d'état disparaît et l'utilisateur peut à nouveau modifier son énoncé.

2.3 LES OUTILS

A l'heure actuelle, un certain nombre d'outils sur des réseaux de Petri ont été intégrés à l'atelier AMI :

- ARP (Analyse de Réseaux de Petri) [Hein 88] : il permet l'analyse de réseaux de Petri Ordinaires (non colorés) et notamment le calcul des flots.
- GrapheCouverture [Finkel 90] : il calcule le graphe de couverture d'un réseau de Petri Ordinaire.
- ECS [Amarger 88] : il effectue le calcul du graphe d'états au plus tôt d'un réseau de Petri T-temporisé.
- MIAMI [Dagron 89] : c'est un Moteur d'Inférences utilisé dans l'atelier AMI. Il permet de générer des applications à partir de paquets de règles qui sont moulinés par le moteur d'inférences. A l'heure actuelle, il est possible, pour des réseaux de Petri Ordinaires :
  - de faire de la simulation,
  - de vérifier des propriétés structurelles (réseau pur, graphe d'événements, graphe simple,...),
  - de procéder à des réductions [Berthelot 86].

- COMBAG [Trèves 89] : il regroupe un certain nombre de méthodes de calcul de familles génératrices de semi-flots pour les réseaux de Petri Ordinaires et les réseaux de Petri colorés.
- GreatSPN [Chiola 87] : il permet, entre autres, de vérifier un grand nombre de propriétés structurelles (ensembles de conflits structurels, exclusion mutuelle, ...) sur les réseaux de Petri stochastiques ainsi que sur les réseaux de Petri temporisés.
- CHP [Couvreur 90a] : il intègre un nouvel algorithme de calcul des familles génératrices de semi-flots dans les réseaux à prédicats / prédicats unaires.
- Rdp / TAPIOCA [Bréant 90] : il s'agit d'une étude d'un générateur de code avec, comme langage cible, OCCAM.
- Rdp / TAGADA [Kordon 90] : c'est la version d'étude du générateur de code en langage Ada.

2.4 LE FORMALISME DE DESCRIPTION UTILISÉ

Pour une meilleure homogénéïsation des différents outils, nous les regroupons autour d'un formalisme plus général intégrant toutes les classes de réseaux de Petri traitées par l'atelier AMI : les AMI-nets [Bonnaire 92a]. L'objet de cette section est de décrire sommairement la structure d'un tel réseau de Petri.

Le modèle des réseaux AMI est dérivé du modèle théorique des Réseaux Bien Formés. Il s'agit d'un modèle de haut niveau qui comporte, outre la description graphique d'un réseau de Petri - places, transitions et arcs - des informations textuelles associées aux arcs et aux nœuds du modèle : noms des places et des transitions, valuation des arcs, marquages des places, domaines de couleurs des places et des transitions, garde des transitions...

La description d'un AMI-net se décompose en deux parties (Figure VIII.3).

La première contient la structure graphique du réseau de Petri (places, transitions et arcs).

La seconde concerne les attributs du modèle. Ces attributs sont associés aux différents objets du réseau de Petri (nom des objets, domaine de couleurs associés...). Ils respectent une syntaxe précise et référencent des objets définis dans une partie déclarative :

- les classes de couleurs;
- les domaines de couleurs, combinaisons de classes et / ou de domaines en vue de définir des marques composées;
- les variables utilisées pour la valuation des arcs du réseau; chaque variable possède un type (classe ou domaine, selon que la marque qu'elle représente est simple ou composée).

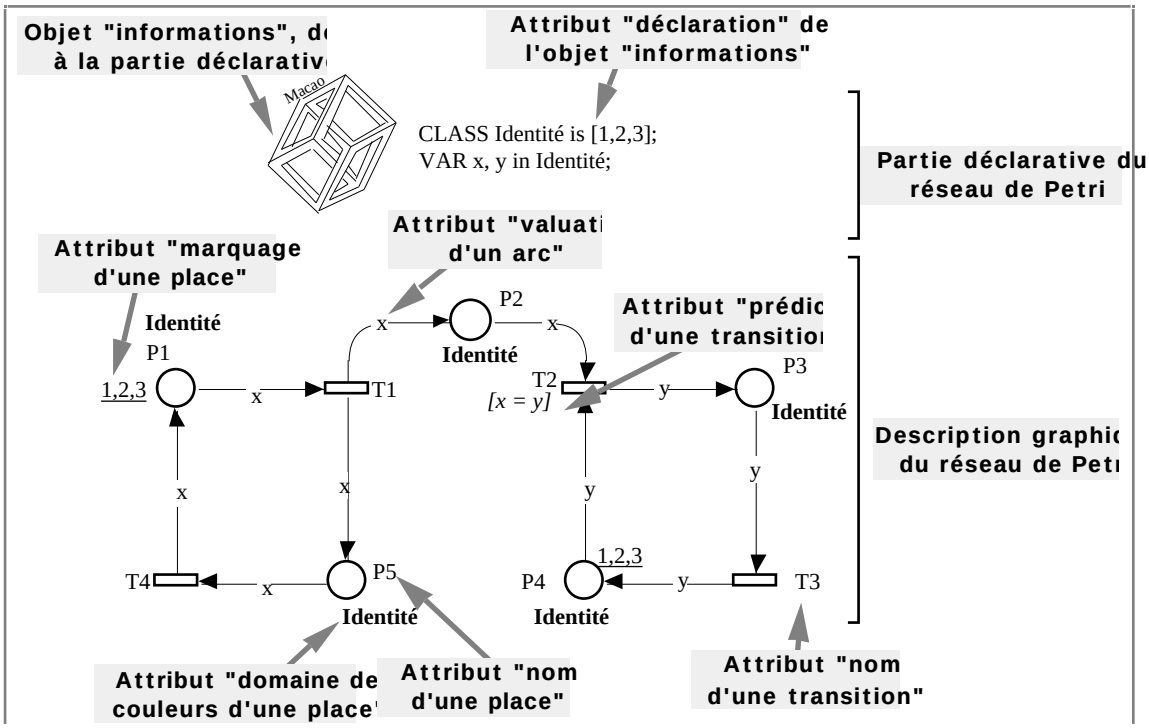


Figure VIII.3 : Exemple de réseau AMI.

Exemple VIII.2 : La Figure VIII.3 donne un exemple de producteur-consommateur énoncé dans le formalisme AMI-net, dans l'atelier de modélisation interactive. Il est dessiné à l'aide du logiciel Macao.

La déclaration des objets respecte une syntaxe précise [Bonnaire 92a]. Elle est fournie dans un attribut d'un objet supplémentaire dédié, l'objet "informations".

La description graphique contient des arcs et des nœuds. Les attributs associés aux objets fournissent des informations sémantiques (nom, marquage, domaine de couleurs, valuation, prédicat...).

Le texte contenu dans les attributs respecte une grammaire. Il est donc analysable par l'atelier. L'information sera transmise aux outils sous une forme pré-analysée [Bonnaire 92b].

Les *AMI-nets* permettent de décrire aussi bien des réseaux de Petri Ordinaires (certaines informations sont ignorées) que des réseaux de Petri de haut niveau (stochastiques, temporisés, colorés...). Naturellement, notre outil de prototypage ne prend en compte que les informations liées à la classe des réseaux de Petri Bien Formés définis dans [Chiola 90, Dutheillet 91].

3 Le générateur de code

Nous décrivons, dans cette section, le générateur de code que nous développons dans le cadre du projet TAGADA (Traduction, Analyse et Génération Automatique de code aDA).

A l'heure actuelle, seuls des prototypes Ada sont pris en compte par le générateur de code. Nous nous sommes cependant attachés à définir notre outil de manière à pouvoir intégrer facilement d'autres langages cibles. Ainsi, l'architecture reste générale; elle suit les principes donnés au Chapitre III.

Dans un premier temps, nous abordons l'architecture du générateur de code. Ensuite, nous détaillons les hypothèses que nous posons, dans un premier temps, sur les modèles que nous traitons. Ensuite nous nous intéressons aux différentes applications composant l'outil.

3.1 ARCHITECTURE DE L'OUTIL

Le générateur de code Ada consiste en quatre applications intégrées dans l'atelier AMI (Figure VIII.4).

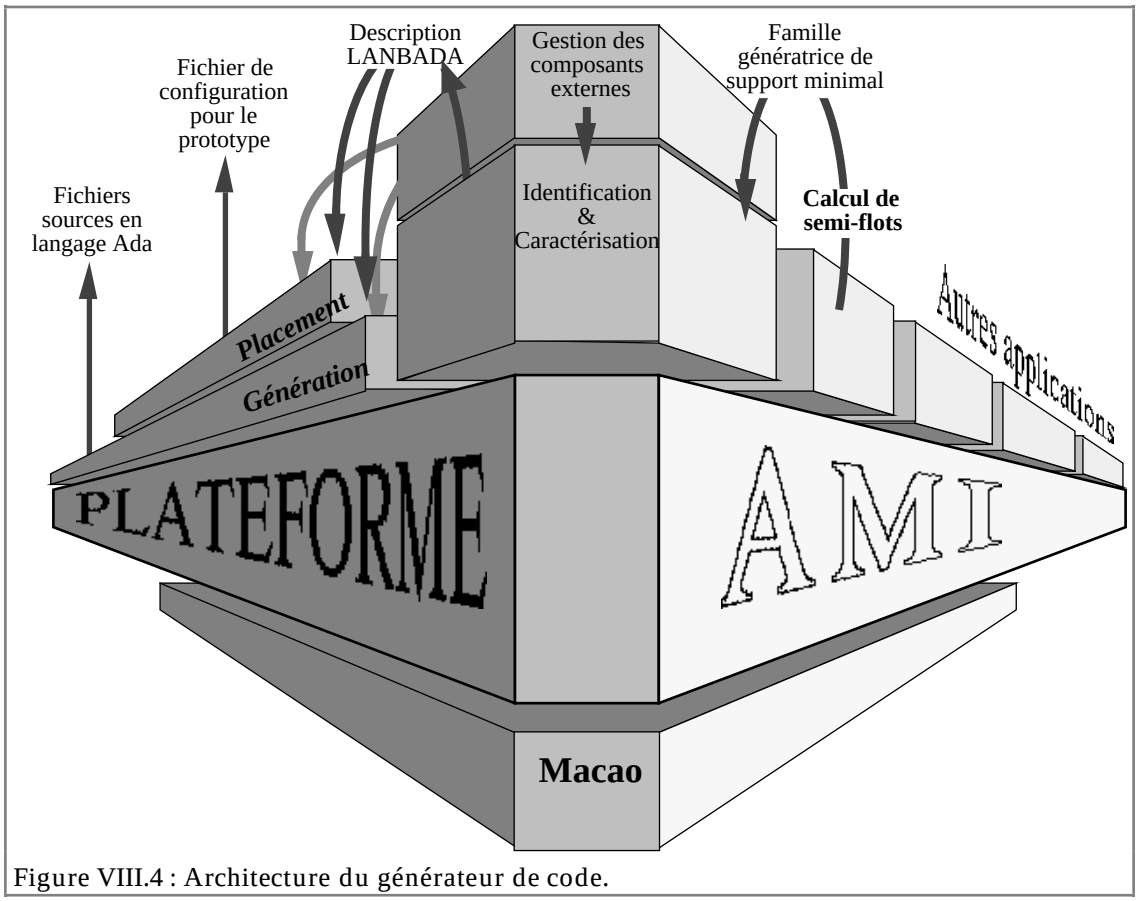


Figure VIII.4 : Architecture du générateur de code.

La première application couvre les deux premières étapes du prototypage (Identification et Caractérisation); elle produit une description intermédiaire exprimée en LANBADA (LANgage BAsic de Description d'un modèle

Analysé), décrivant le réseau de Petri en terme de G-objets. Le format de cette description est donné en Annexe C.

A partir de la description intermédiaire, une deuxième application produit un programme source Ada, correspondant au prototype centralisé ou décentralisé. C'est la phase de Génération décrite dans le chapitre III.

La troisième application, sur la base du modèle, de sa décomposition et de la description d'une architecture, propose un placement des objets composant le prototype réparti. C'est la phase de Placement définie dans le chapitre III. Cette application n'est pas encore réalisée.

Une quatrième application se charge de la gestion des bibliothèques de composants externes. Ces bibliothèques seront utilisées :

- durant la phase d'Identification,
- durant la phase de Placement,
- durant la phase de Génération.

L'application réalisant les deux premières étapes du prototypage se contente de lire la formalisation des composants externes. Les applications réalisant la phase de Placement ou de Génération utilisent la description interne : les contraintes de localisation pour l'une, les informations liées au langage pour l'autre.

L'application réalisant les phases d'Identification et de Caractérisation dialogue avec un outil effectuant le calcul d'une famille génératrice de support minimal. Pour ce calcul, le logiciel GreatSPN [Chiola 87] est utilisé.

3.2 HYPOTHÈSES

Nous introduisons dans cette section les hypothèses réalisées sur les modèles applicables au générateur de code :

- elles sont liées à des contraintes de notre méthode de prototypage;
- elles correspondent à des choix de réalisation du générateur de code;
- elles permettent de résoudre un problème d'interprétation du modèle.

Hypothèse liée aux contraintes de notre méthode de prototypage

***Hypothèse 1 :* précondition statique ou processus et Etat_Processus simple**

> En cas d'échec de l'évaluation d'une précondition processus ou statique (Définitions IV.11 et IV.34), il faut pouvoir effectuer un retour au point de choix précédent.

Hypothèses liées à des choix de réalisation du générateur de code

***Hypothèse 2 :* précondition contrainte et Ressources externes**

> Pour des raisons d'implémentation, nous interdisons, dans un premier temps, de consommer, dans une Ressource externe en sortie, une marque d'un profil donné.

***Hypothèse 3 :* fonction de diffusion**

Pour des raisons d'implémentation, nous ne gérons pas, dans un premier temps, la fonction de diffusion (sélection de l'ensemble des objets formant une classe).

***Hypothèse 4 :* manipulation de marques composées**

Pour des raisons d'implémentation, toute manipulation de marques composées, résultat d'une composition imbriquée de plusieurs domaines de couleurs est impossible.
Il existe deux façons de contourner cette hypothèse :

- "aplatir" les domaines de couleurs, de manière à n'avoir qu'un seul niveau de profondeur, cela peut être réalisé automatiquement au niveau d'une "première passe";
- effectuer l'extraction, ou la concaténation en plusieurs étapes.

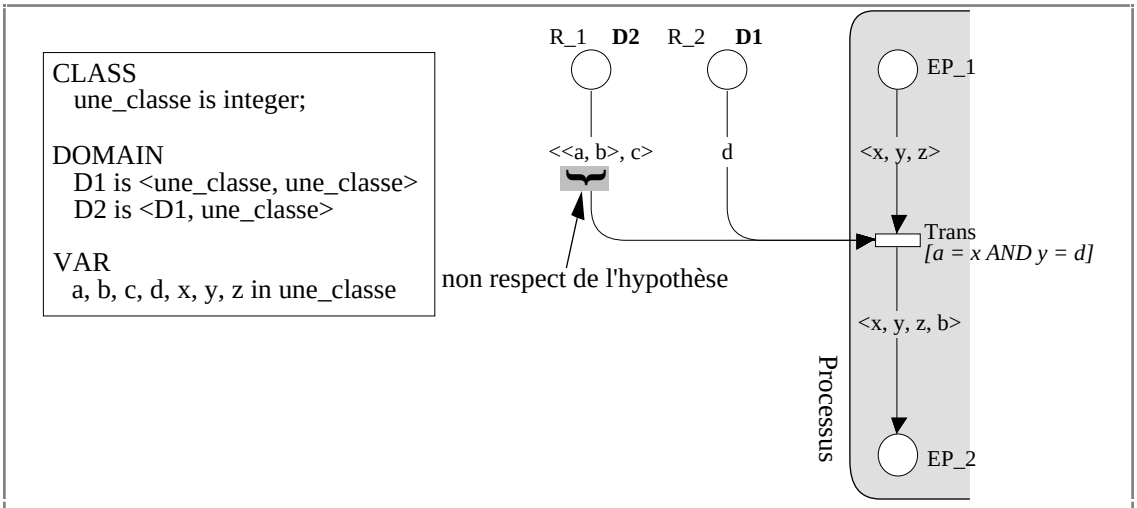


Figure VIII.5 : Exemple d'accès à une composante de profondeur 2. Le domaine de couleurs des Ressources sont donnés en caractères gras.

***Hypothèse 5 :* Actions synchronisées et Ressources**

Pour des raisons d'implémentation, nous avons décidé, dans un premier temps, de ne pas traiter les pré et post conditions ressource des Actions synchronisées.

***Hypothèse 6 :* contraintes sur les préconditions processus, statiques et synchrones**

Pour des raisons d'implémentation, nous avons décidé, dans un premier temps, de ne pas traiter :

- la comparaison de deux composantes d'un même mot d'état,
- l'opération entre deux résultats de requêtes.

***Hypothèse 7 :* l'opérateur relationnel ¬**

Pour des raisons de performances, nous avons décidé de ne pas réaliser l'opérateur ¬. Un tel opérateur correspond :

- soit à calculer une requête base de données, puis à prendre, dans les places Ressources, les marques qui n'ont pas été sélectionnées (opération fort coûteuse),

- soit à inverser les composantes argument du ¬ au niveau d'un prédicat. cela permet d'obtenir un code plus performant.

Il nous semble raisonnable d'appliquer la transformation liée à la seconde interprétation dans une "passe préliminaire" effectuée avant le prototypage du modèle.

Hypothèse 8 : **consommation des marques colorées**

Pour des raisons d'implémentation, il est impossible de consommer plus d'une marque colorée à la fois.
La consommation de plusieurs marques colorées est cependant possible.

Hypothèse 9 : **contraintes sur les mots d'état**

Les champs d'un mot d'état doivent tous être des domaines ou des classes définies préalablement dans les déclarations du modèle.
Une "première passe" effectuée avant l'application du prototypage doit permettre de supprimer aisément cette hypothèse.

Hypothèse 10 : **fonctions de couleurs**

Dans un premier temps, seules les fonctions de couleurs prédéfinies sur les classes de couleurs sont traitées, c'est-à-dire : successeur, prédécesseur et Identité.

Hypothèse 11 : **composants externes réversibles**

Pour des raisons d'implémentation, nous ne gérons dans un premier temps que des composants externes non-réversibles.

Hypothèses permettant de résoudre des problèmes d'interprétation

Hypothèse 12 : **position des modificateurs**

Il s'agit ici d'un choix dans la représentation d'un réseau de Petri. Les modificateurs (fonctions successeur et prédécesseur) ne peuvent être utilisés que :

- dans le prédicat associé à une transition : les relations entre marques sont alors soumises à ces opérations;
- dans la postcondition d'une transition : il y a alors modification de la valeur contenue dans la marque.

Hypothèse 13 : **contraintes sur les domaines de couleurs**

Les composantes d'un domaine doivent obligatoirement être :

- d'autres domaines déjà définis et nommés (définition d'un champ de marque composé),
- des classes déjà définies et nommées (définition d'un champ de marque simple).

Au cours d'une "première passe", un "aplatissement" des domaines permet de contourner cette hypothèse.

***Hypothèse 14* : contraintes sur les arcs entrants d'une Action**

Tous les identificateurs des variables référencées dans les arcs entrants d'une Action doivent être différents.
Si cette règle n'est pas respectée, il devient difficile d'interpréter, au niveau du générateur de code, la provenance des différentes composantes d'une précondition. De plus, l'utilisation de mêmes noms de variables, dans la définition d'égalités implicites entre marques (Figure VIII.6), pose parfois des problèmes d'ambiguïté.
Ainsi, l'hypothèse s'apparente à une règle de syntaxe applicable aux modèles qui sont transmis au générateur de code.

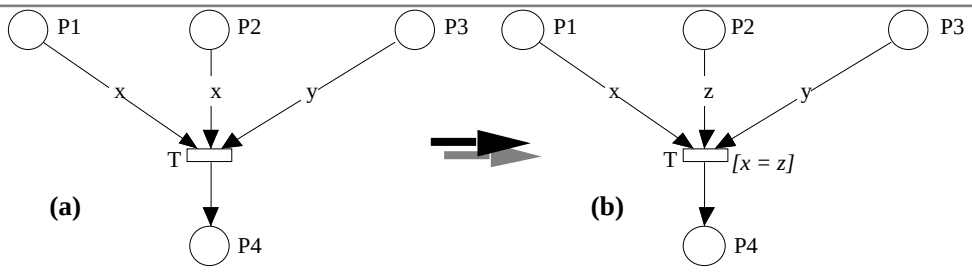


Figure VIII.6 : Exemple d'utilisation des noms de variables sur les arcs en entrée d'une Action pour spécifier une égalité des marques consommées (a). Le modèle (b) propose une écriture équivalente, moins ambiguë mais plus lourde. (b) vérifie l'hypothèse 1, (a) non.

3.3 LES PHASES D'IDENTIFICATION ET DE CARACTÉRISATION

Nous décrivons dans cette section l'application réalisant la décomposition du modèle.

3.3.1 Architecture

L'application réalisant les phases d'Identification et de Caractérisation se décompose en quatre parties (Figure VIII.7) :

- traduction du modèle en une structure de données interne,
- extraction des composants externes,
- décomposition du modèle en objets,
- génération de la description LANBADA.

La phase d'Identification s'effectue en deux étapes :

- *transcription de l'énoncé* : dans un premier temps, il faut lire, puis décoder la représentation transmise par AMI. Le résultat est stocké sous une forme intermédiaire qui est ensuite analysée en vue d'obtenir la structure de données finale. Enfin, il est procédé à la vérification de certaines contraintes (hypothèses 2, 3, 4, 7, 8, 10, 11 et 12).
- *trouver le modèle ouvert* : à partir des données contenues dans les bibliothèques de composants externes et du résultat de l'étape précédente, il faut déterminer les sous-ensembles de nœuds qui ne font pas partie du système modélisé. Cette étape fournit deux résultats :
  - une description du modèle ouvert,
  - la liste des Ressources externes du modèle; les points de connexion avec le modèle ouvert sont conservés.

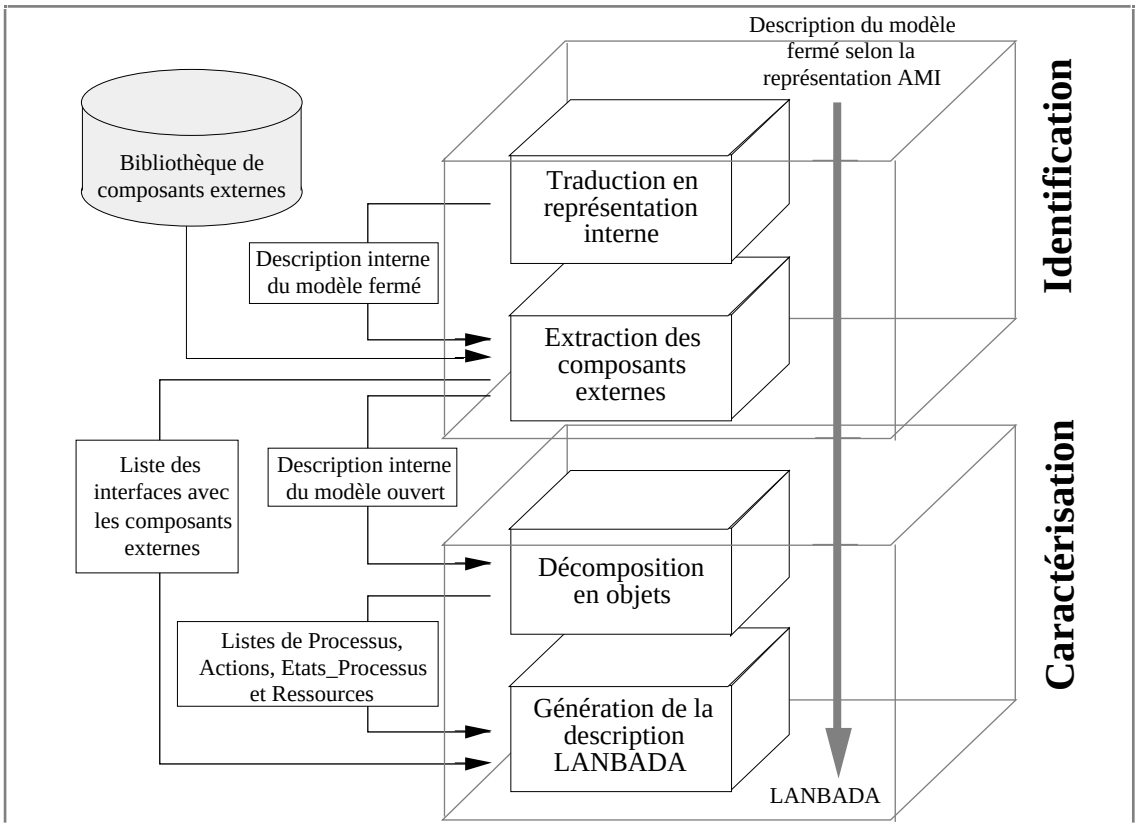


Figure VIII.7 : Architecture de l'application réalisant les phases d'Identification et de Caractérisation.

La phase de Caractérisation se décompose en deux étapes :

- *calcul des G-objets* : sur la base de la description du modèle ouvert, et de la famille génératrice de semi-flots, cette étape identifie les G-objets.
- *construction du résultat* : à partir de la liste des Ressources externes et de la décomposition G-objets du modèle, la description LANBADA est générée.

3.3.2 Identification des composants externes

Les données concernant les composants externes sont contenues dans des bibliothèques lues par la phase d'Identification. Tous les composants externes référencés dans le modèle ouvert doivent avoir été définis au préalable.

L'identification des composants externes s'effectue en plusieurs étapes (Figure VIII.8) : le repérage des nœuds correspondant aux composants externes, la vérification de leur utilisation et la création des résultats

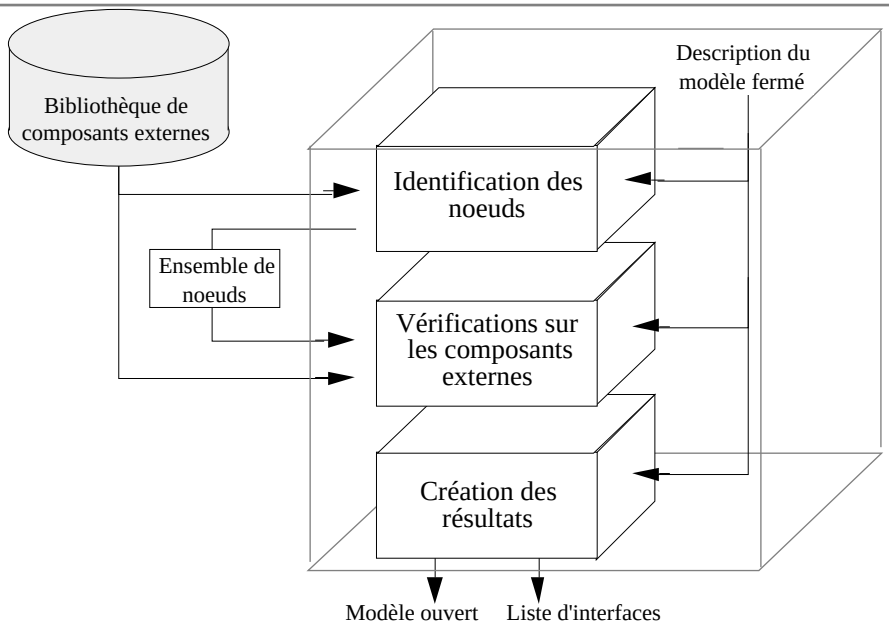


Figure VIII.8 : Etapes de la phase d'Identification.

Identification des composants externes

Le concepteur du modèle doit, pour cela, indiquer les places d'interfaces (Ressources externes) ainsi que le nom des composants externes qu'elles représentent. A partir des données contenues dans les bibliothèques, il est possible de déterminer si l'interface est en entrée ou en sortie.

Les Ressources externes étant les seuls liens entre la spécification du système et le composant, nous pouvons calculer les sous-ensembles de nœuds n'appartenant pas au modèle ouvert.

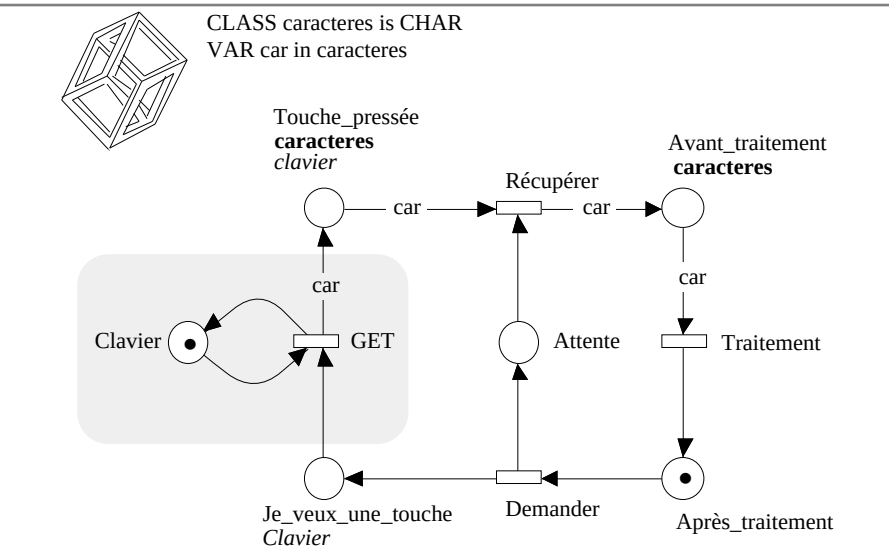


Figure VIII.9 : Exemple de modèle contenant un composant externe.

Exemple VIII.3 : Considérons le réseau de Petri de la Figure VIII.9. Le concepteur a indiqué que les places *Touche_pressée* et *Je_veux_une_touche* sont des interfaces avec le composant externe non-réversible *clavier*. Dans la bibliothèque, *Je_veux_une_touche* est classée en entrée tandis que *Touche_pressée* l'est en sortie.

Les Actions en entrée de *Touche_pressée* ou en sortie de *Je_veux_une_touche* appartiennent au composant. De là, nous déduisons l'ensemble des nœuds n'appartenant pas au modèle ouvert : *GET* et *Clavier*.

Si seules les interfaces avec les composants externes sont représentées, cette étape n'est pas effectuée. Il est alors possible de faire, pour des raisons ergonomiques (éviter des croisement d'arcs...), plusieurs références à une même interface. Au niveau de l'Identification, elles sont considérées comme un nœud unique.

Vérification de l'utilisation des composants externes

Nous effectuons deux types de vérification sur l'utilisation des composants externes :

- Les vérifications sémantiques : elles n'ont pas de sens si le modèle fermé a été validé au préalable. Ce n'est pas le cas lorsque seules les Ressources externes sont spécifiées.
- La vérification des contraintes d'utilisation (pour les composants externes non-réversibles).

Les vérifications sémantiques consistent en un test de non-blocage du système. Il existe des dépendances entre Ressources externes :

- des places d'interface en sortie peuvent correspondre à des résultats fournis par des services activés via des places d'interface en entrée (le clavier dans la Figure VIII.10);
- certains services peuvent être activés à l'aide de plusieurs places d'interface en entrée (Exemple VIII.4).

Il faut dans ce cas vérifier que l'utilisation du composant externe est cohérente. Si un système attend un résultat sans avoir activé le service correspondant, le blocage est certain.

Exemple VIII.4 : Considérons le composant externe non-réversible *connexion*, donné en Figure VIII.10 réalisant un service de communication synchrone. La communication ne peut avoir lieu que si deux clients de types complémentaires se connectent. Une fois la communication réalisée, les clients reprennent leur exécution.

Les clients signalent leur présence via les places d'interface en entrée : il y en a une par catégorie de client. Ils peuvent reprendre leur exécution lorsque les places d'interface en sortie sont marquées.

Les dépendances entre places d'interface sont les suivantes :

- *connexion_client_type1* et *connexion_client_type2* sont interdépendantes. Pour être vivant, un système doit produire dans l'une et dans l'autre, ou dans aucune des deux.
- *resultat_client_type1* dépend de *connexion_client_type1* et de *connexion_client_type2*. Un résultat ne peut être obtenu que lorsque deux clients se sont connectés. Il en est de même avec *resultat_client_type2*.

Si le système attend un résultat dans *resultat_client_type1* mais pas dans *resultat_client_type2*, un message d'alerte est transmis à l'utilisateur mais le prototypage n'est pas interrompu. Nous n'avons pas assez d'informations pour prendre une telle décision (a).

Par contre, un système attendant des résultats dans *resultat_client_type1* et *resultat_client_type2*, alors que seule l'une des deux entrées correspondantes a été connectée, est rejeté : le blocage est ici inévitable (b).

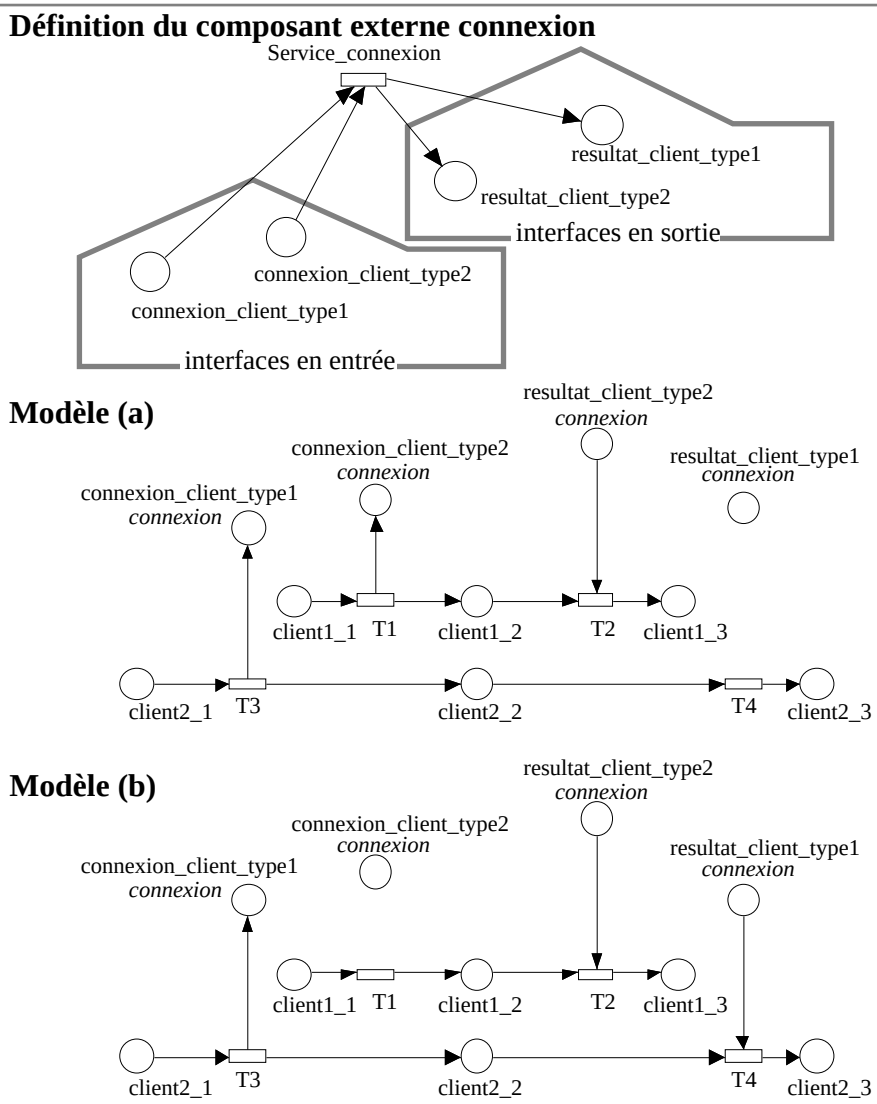


Figure VIII.10 : Exemple de dépendances d'interfaces en entrée dans un composant externe. Nous donnons deux mauvaises utilisations de ce composant externe.

Si le composant externe est intégré dans le modèle, il faut vérifier qu'il correspond bien à la formalisation qui en est donnée dans la bibliothèque. Si ce n'est pas le cas, le processus de prototypage est interrompu.

Enfin, il faut effectuer les vérifications liées aux contraintes d'utilisation des composants externes non-réversibles (Définition III.5).

Création des résultats

Enfin, si l'utilisation des composants externes est correcte, nous supprimons leurs nœuds dans le modèle fermé et ne conservons que leurs interfaces. La liste des interfaces et le modèle ouvert constituent deux résultats distincts.

3.3.3 Décomposition du modèle

La décomposition du modèle s'effectue en trois étapes (Figure VIII.11) :

- construction du modèle structurel équivalent au modèle ouvert;

- calcul de la famille génératrice de semi-flots de support minimal : il est réalisé par un outil indépendant, le dialogue étant assuré par l'environnement AMI;
- calcul de la décomposition, puis des G-objets selon l'algorithme décrit dans le chapitre II, section 3.1.

Un dialogue avec l'utilisateur est à prévoir dans certains cas :

- lorsque plusieurs décompositions sont possibles : l'outil doit lui permettre de visualiser les différents choix possibles puis identifier celle qui a été choisie.
- si aucune décomposition n'est possible : pour le moment, seul un diagnostic sommaire de la cause de l'échec est fourni.

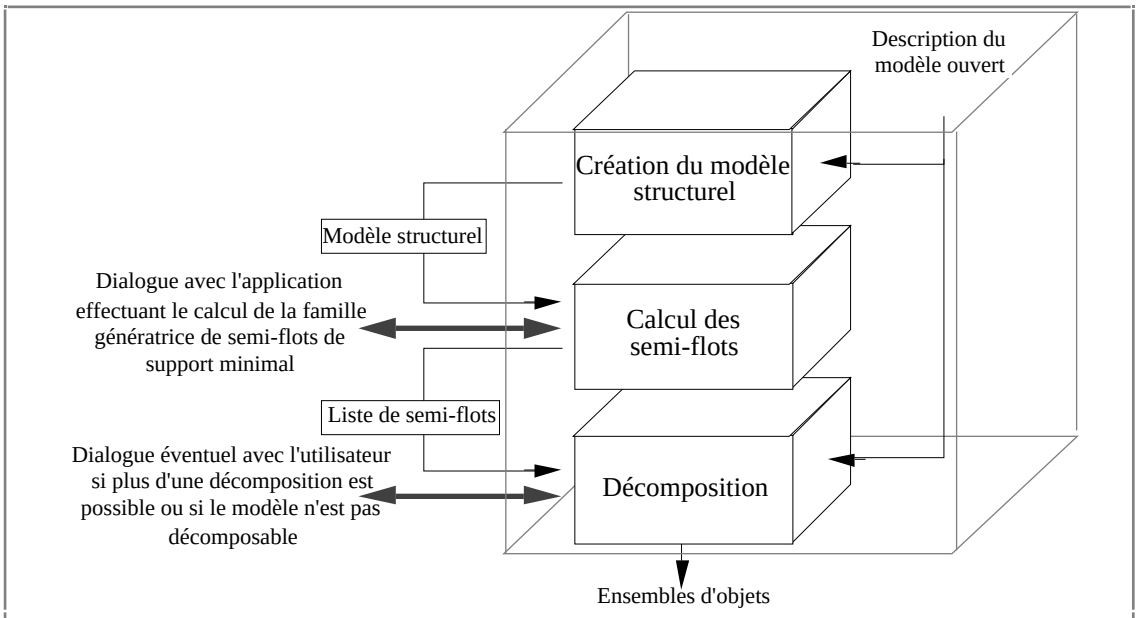


Figure VIII.11 : Traitements associés à la décomposition d'un modèle.

3.4 LA PHASE DE PLACEMENT

La phase de placement n'est encore supportée par aucun outil. De telles réalisations ont déja eu lieu [Lavarenne 89]. Nous envisageons de la réaliser sur la base des travaux présentés dans [Sinclair 87, Lo 88, Billionnet 89].

L'étude des heuristiques nous permettant, connaissant les particularités du prototype réparti, de proposer un placement adéquat permettra de réaliser cette application. Elle sera intégrée lorsque le générateur de code sera étendu en vue de générer également un prototype réparti.

3.5 LA PHASE DE GÉNÉRATION

La phase de génération est décomposée en trois étapes (Figure VIII.12). Dans un premier temps, il faut procéder à la traduction de la description LANBADA du modèle en une structure de données interne à l'application. Ensuite, nous vérifions les hypothèses 1, 5, 6, 9, 13 et 14.

Enfin, nous générons le squelette de l'application. Pour cela, il nous faut accéder à la bibliothèque de composants externes. A ce stade, cette bibliothèque doit contenir, pour les composants externes manipulés, les informations propres au langage Ada. Dans le cas contraire, la génération ne peut avoir lieu.

L'utilisateur peut influer sur le code généré :

- En demandant un prototype réparti ou non : pour le moment le générateur de code n'est pas capable de générer automatiquement un prototype réparti.
- En commutant le mode DEBUG, afin, par exemple, de tester l'intégration de ses bibliothèques dans le prototype.
- En donnant des directives concernant la politique de choix dans les alternatives. Actuellement, deux politiques sont disponibles : tirage aléatoire ou tourniquet [Krakowiak 85].

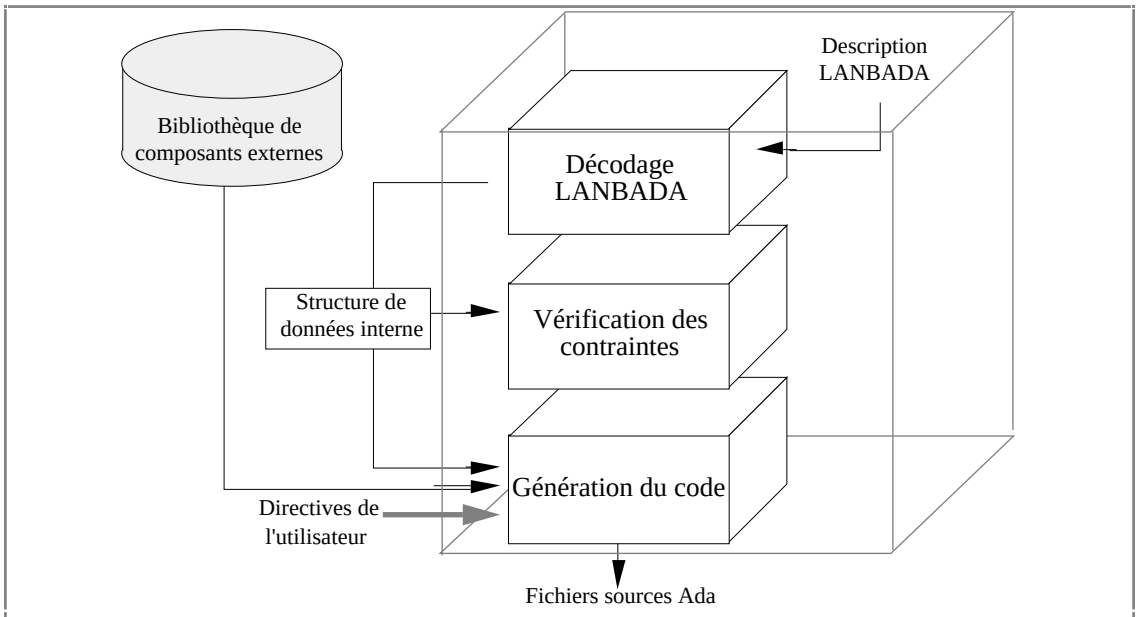


Figure VIII.12 : Etapes de la phase de Génération.

Les procédures associées aux Actions du modèle sont regroupées dans différents paquetages : un par processus plus un pour l'ensemble des Actions synchronisées. Un corps par défaut est fourni (instruction vide ou affichage d'une trace).

Le concepteur du modèle peut cependant associer ses propres programmes. Actuellement, il lui est cependant impossible de changer la spécification de telles procédures. Celles qu'il fournira devront respecter un format défini en fonction de l'analyse du modèle. Ce format lui permet, s'il le souhaite, d'accéder en *lecture seulement* (paramètres passés par valeur) aux marques manipulées par le processus concerné. Les données transmises sont :

- le mot d'état de l'instance du processus qui exécute l'Action;
- l'identité de l'instance (numéro interne plus type de processus). Cette information est destinée à des fins de certification du générateur de code.

Outre le prototype, nous générons le train de compilation, c'est à dire l'ordre de compilation des différentes unités. Nous intégrons les directives d'édition de liens que le concepteur du modèle souhaite éventuellement spécifier.

3.6 LA GESTION DES COMPOSANTS EXTERNES

La définition d'un composant externe est distincte du prototypage en lui-même; elle n'a aucune incidence sur le procédé de réalisation. Elle doit être effectuée avant le prototypage. Conformément à la description d'une abstraction de composant externe (Définition III.2), elle comporte deux volets :

- La *formalisation* qui permet de localiser le modèle ouvert et de déduire les dépendances entre interfaces (Ressources externes);
- La *description interne* qui contient deux types d'informations :
  - les contraintes de placement du composant utilisées dans le placement des objets dans un prototype réparti;
  - les noms et corps des primitives de manipulation utilisées pour générer le prototype.

  Il est possible de déduire, à partir du domaine d'une place d'interface, le profil de la primitive associée. Le concepteur d'un système doit cependant lui associer un traitement (corps de la primitive). Cela permet d'intégrer le composant externe au prototype.

Ces trois informations doivent être gérées différemment. La formalisation ne dépend ni du langage ni de l'architecture cible. Les contraintes de placement sont liées à l'architecture supportant l'application. Enfin, le nom et le corps des primitives de manipulation sont liés au langage cible.

Le stockage du composant, dans une bibliothèque, tient compte de ces différents aspects. Par ailleurs, il est possible d'enrichir un composant externe déjà défini, c'est-à-dire ajouter des contraintes liées à une architecture ou des corps de primitives pour un langage cible.

Si la formalisation d'un composant externe est incorrecte, il est rejeté. Enfin, il nous paraît nécessaire de prodiguer une gestion de bibliothèques de composants externes similaire à ce qu'offre un compilateur.

3.6.1 Organisation des différentes bibliothèques

Afin de faciliter l'utilisation de l'outil de prototypage, nous considérons trois niveaux de bibliothèques disjoints : le niveau *système*, propre à l'environnement système, le niveau *administrateur*, lié au domaine de l'application, et le niveau *concepteur*, lié à l'application elle-même

Le niveau système

Il s'agit d'un ensemble de bibliothèques contenant des composants externes prédéfinis, dont le comportement a été validé. Seul l'administrateur du système peut modifier le contenu de ces bibliothèques. A travers cette bibliothèque, les programmes utilisateurs manipulent des composants du système (périphériques, fichiers...). Les utilisateurs n'y accèdent qu'en lecture.

Le niveau administrateur

Il s'agit d'un ensemble de bibliothèques contenant des composants externes prédéfinis, eux aussi considérés comme sûrs. Seul l'administrateur du domaine d'application de l'outil peut modifier leur contenu. Les utilisateurs ne peuvent y accéder qu'en lecture.

Le niveau concepteur

Il s'agit d'un ensemble de bibliothèques contenant les composants externes d'un concepteur.

Chaque niveau comporte plusieurs bibliothèques. L'utilisateur peut fournir des règles de recherche dans la spécification de son système.

3.6.2 Organisation d'une bibliothèque

Chaque bibliothèque est gérée sous la forme d'une arborescence UNIX. C'est-à-dire d'un répertoire contenant des sous-répertoire; chacun d'eux décrit un composant.

Les différents fichiers décrivant un composant externe sont eux-même organisés en arborescence (Figure VIII.13). Les informations à conserver sont : le réseau de Petri, la listes des interfaces du modèle ainsi que les primitives d'accès et les contraintes de placement.

Chaque composant externe est vu comme la racine d'une arborescence. On y trouve cinq classes d'informations :

- *Description des interfaces* : chacune d'entre elles est décrite dans un fichier séparé. Les informations contenues sont : leur nom, leur mode (entrée ou sortie) et leur domaine de couleurs. Ces informations sont utilisées pour définir le profil type des primitives de manipulation associées. Elles sont lues lorsque le concepteur du composant externe décrit les primitives de manipulation dans un langage cible donné.
- *Informations générales* : il s'agit de la description du réseau de Petri formalisant le composant. Ces informations sont utilisées afin de déterminer les nœuds ne correspondant pas au modèle ouvert.
- *Informations propres au langage cible* : elles sont destinées à la phase de Génération et sont classées par langage. Lorsque le concepteur d'un composant externe crée des primitives de manipulation, un nouveau sous-répertoire est créé, contenant les informations liées à chacune des interfaces pour le langage correspondant. Les informations contenues sont :
  - le nom de la primitive,
  - le nombre de paramètres,
  - s'il y a lieu, le nom de chacun des paramètres,
  - une description du corps de la primitive,
  - des directives permettant, à l'édition de liens, de trouver les librairies contenant ces primitives.

- *Dépendances entre interfaces* : elles sont calculées à la définition de la formalisation du composant.
- *Contraintes architecturales* : il s'agit des contraintes de placement liées à une architecture. Ces informations sont destinées à la phase de Placement.

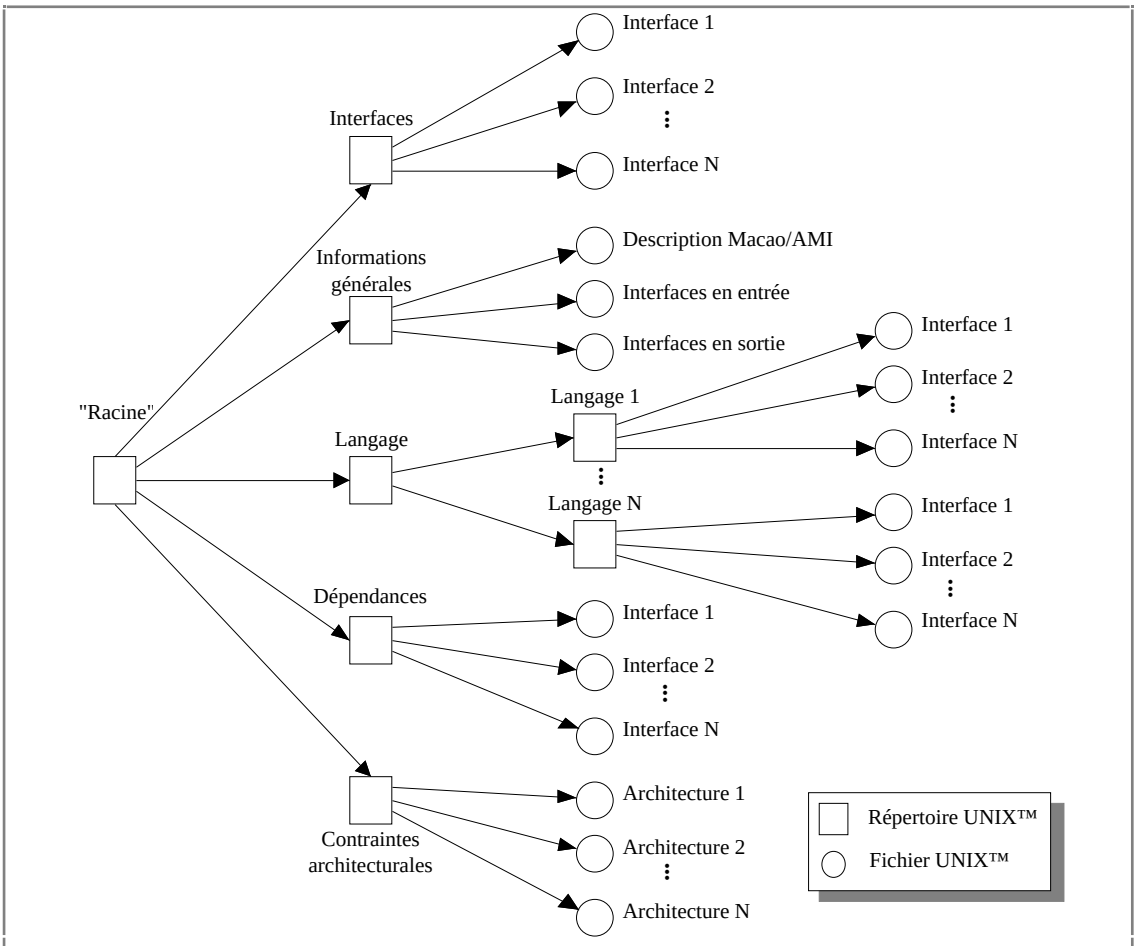


Figure VIII.13 : organisation de l'arborescence UNIX décrivant un composant.

3.6.3 Création d'un composant

Une application gère la création des composants externes. Elle réceptionne la formalisation sous la forme d'un réseau de Petri coloré (formalisme AMI-nets) et effectue des vérifications. Le composant externe sera ajouté à la bibliothèque s'il respecte les contraintes données dans la Définition III.3. Le type du composant externe (réversible ou non-réversible, doit être spécifié).

Une fois la formalisation du composant externe intégrée dans la bibliothèque, il est possible d'ajouter des données propres à un langage cible, ou à une architecture donnée.

La modification d'une information déjà contenue dans la bibliothèque (changement du corps d'une primitive, par exemple) est considérée comme un ajout simple, après suppression des anciennes données.

Ajout de la formalisation du composant

Cette opération s'effectue en trois étapes (Figure VIII.14). Dans un premier temps, nous conservons le réseau de Petri formalisant le composant.

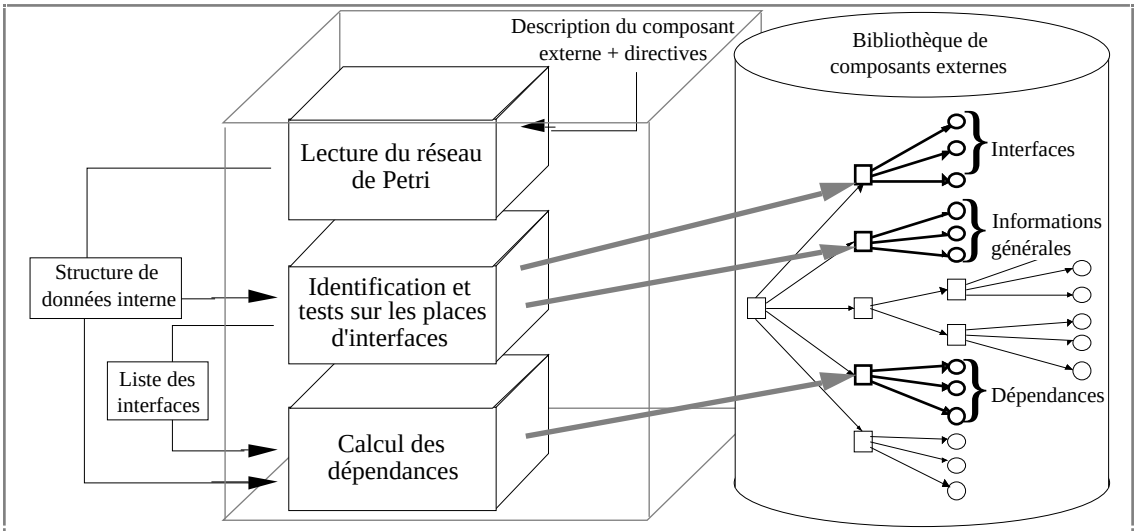


Figure VIII.14 : Traitements associés à l'ajout de la formalisation d'un nouveau composant externe dans la bibliothèque.

Ensuite, nous identifions les interfaces (Ressources externes) et effectuons des tests sur leurs relation avec la formalisation (partie privée) :

- au moins un arc sortant à partir de chaque place d'interface en entrée,
- pas d'arc entrant en direction d'une place d'interface en entrée,
- au moins un arc entrant en direction de chaque place d'interface en sortie,
- pas d'arc sortant à partir d'une place d'interface en sortie.

Si les contraintes sont respectées, le composant externe est valide. Une nouvelle entrée est créée dans la bibliothèque; les informations générales, puis celles concernant les interfaces, sont écrites.

Enfin, nous procédons au calcul des dépendances entre interfaces. Les dépendances circulaires sont signalées à l'utilisateur mais elles n'entraînent pas l'invalidation du composant.

La définition des primitives associées aux places d'interface

Connaissant le langage cible et le domaine de couleurs d'une Ressource externe, il est possible de déduire la spécification de la primitive associée. L'utilisateur ne peut influer que sur le nom des identificateurs concernés. Il doit également associer un corps à la primitive.

Ainsi, lorsqu'un concepteur souhaite définir, pour un langage donné, les primitives associées à un composant, l'application calcule automatiquement un profil et un corps par défaut. Le profil par défaut est composé d'identificateurs construits automatiquement; le corps par défaut est vide. L'ensemble lui est restitué, via l'atelier AMI, dans un formalisme textuel.

Si l'utilisateur veut changer les identificateurs, ou modifier le corps de la primitive, il réécrit le texte ainsi défini. Il peut également ajouter des directives d'édition de liens qui seront intégrées dans le fichier décrivant le train de compilation d'un prototype.

Aucune analyse, même syntaxique, ne peut être effectuée. Une fois les modifications validées, le texte est décodé et les informations sont intégrées à la bibliothèque.

4 Conclusion

Le projet CPN/TAGADA vise à l'application de nos méthodes de prototypage. Il est intégré à l'environnement AMI [Bernard 88, Bernard 89, Bernard 90] réalisé au laboratoire MASI.

Le générateur de code que nous avons développé a permis d'éprouver notre démarche. Plusieurs modèles de systèmes complexes (jusqu'à 250 processus instanciés) ont fait l'objet d'un prototypage.

La Figure VIII.15 donne une idée de l'ensemble du projet. Actuellement, seules certaines parties sont opérationnelles. Nous envisageons de poursuivre nos réalisations dans les domaines suivants :

- génération d'un prototype réparti;
- raffinement des algorithmes d'analyse;
- intégration des composants externes réversibles dans le prototype.

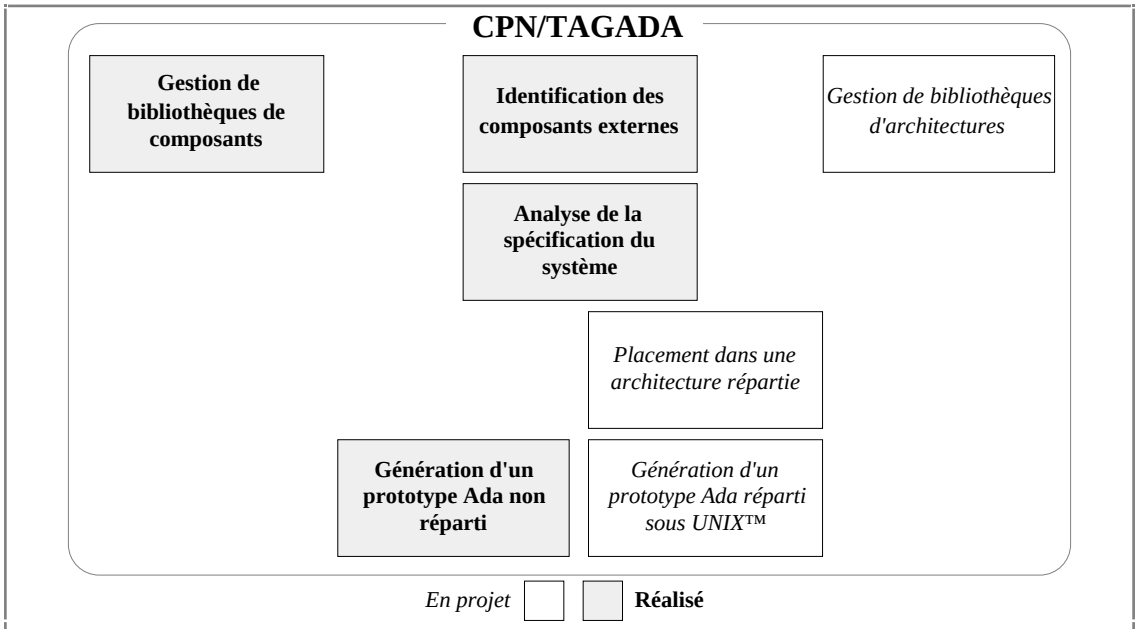


Figure VIII.15 : Etat de réalisation du générateur de code.

Par ailleurs, sur la base du résultat de la décomposition d'un modèle, la génération de prototypes dans d'autres langage seront étudiées (C, C++...).

En vue de rendre le prototype performant, un grand nombre d'optimisations sont réalisées, en vue de réduire le nombre de lignes de code inutiles, et d'augmenter les performances d'exécution.

De même, afin de rendre le code lisible, nous respectons les règles d'indentation préconisées dans [Ada 83] et générons des commentaires.

L'outil de prototypage ne constitue pas un développement à part, il est partie intégrante du projet MARS (Modélisation Analyse et Réalisation de Systèmes), dont l'objectif est d'offrir des outils de modélisation, d'analyse et de réalisation de systèmes parallèles.

			Introduction générale
	Chapitre I	:	*Prototypage de systèmes parallèles*
	Chapitre II	:	*Interprétation sémantique d'un réseau de Petri*
	Chapitre III	:	*Méthodologie de prototypage*
	Chapitre IV	:	*Structure du prototype généré*
	Chapitre V	:	*Caractéristiques du prototype Ada*
	Chapitre VI	:	*Le prototype centralisé*
	Chapitre VII	:	*Le prototype réparti*
	Chapitre VIII	:	*CPN/TAGADA : L'outil de prototypage*
+			*Conclusion générale*
			Bibliographie
			Annexes & Appendices

Conclusion générale

Résumé 327
Apport méthodologique 327
Apport Formel.....329
Apport Génie Logiciel.....329
Apport technique.....330
Perspectives 331

RÉSUMÉ

Le travail que nous avons présenté décrit une méthodologie de prototypage permettant, à partir d'une spécification formelle exprimée à l'aide de réseaux de Petri, de générer automatiquement un prototype opérationnel.

Notre domaine de recherche est assez large puisqu'il couvre un ensemble de problèmes liés à la modélisation, l'analyse et à la réalisation de systèmes distribués. Pour chacun de ces problèmes, il a été nécessaire de maîtriser et de développer les théories et les techniques associées.

Ainsi, nos recherches intègrent différents aspects :

- Méthodologiques : nous avons défini et synthétisé les étapes de l'approche globale permettant de passer de la modélisation d'un système à sa réalisation.
- Formels : nous avons intégré la théorie des réseaux de Petri et développé une interprétation particulière d'un modèle, sur la base du calcul d'un famille génératrice de semi-flots de support minimal.
- Génie logiciel : l'élaboration de notre outil de prototypage s'inscrit dans un cadre général que seule l'utilisation de techniques issues du génie logiciel ont contribué à résoudre.
- Techniques : la maîtrise des caractéristiques et des contraintes du langage Ada a été nécessaire pour réaliser le générateur de code. Par ailleurs, la réalisation des différents outils assurant la modélisation, l'analyse et le prototypage de systèmes utilise largement les aspects les plus délicats de l'informatique dont certains sont déjà définis (gestion du parallélisme, répartition, compilation…).

APPORT MÉTHODOLOGIQUE

Nous avons défini une méthodologie de génération de code permettant de repousser la dépendance par rapport au langage cible et d'intégrer progressivement différentes classes d'informations (interfaçage avec l'environnement, contraintes du langage de réalisation...) permettant de définir un prototype sans ambiguïté. L'intégration d'une représentation formelle garantit un niveau élevé de fiabilité des propriétés du système.

Dans notre approche, la réalisation d'une application a été décomposée en trois étapes (Figure e) :

- Modélisation formelle : à partir du formalisme choisi et du cahier des charges, le concepteur construit un modèle du système à réaliser. Une validation de ce modèle garantit l'absence d'erreurs structurelles de spécification. Certaines aberrations comportementales peuvent également être évitées.

- Prototypage : il s'agit d'automatiser le processus de réalisation d'une application. Cette étape produit un programme source.
- Intégration du prototype : il s'agit d'une édition de liens, permettant d'assembler le squelette de contrôle du prototype et les programmes associés aux différentes actions définies par le concepteur (traitements, composants externes) dans des bibliothèques.

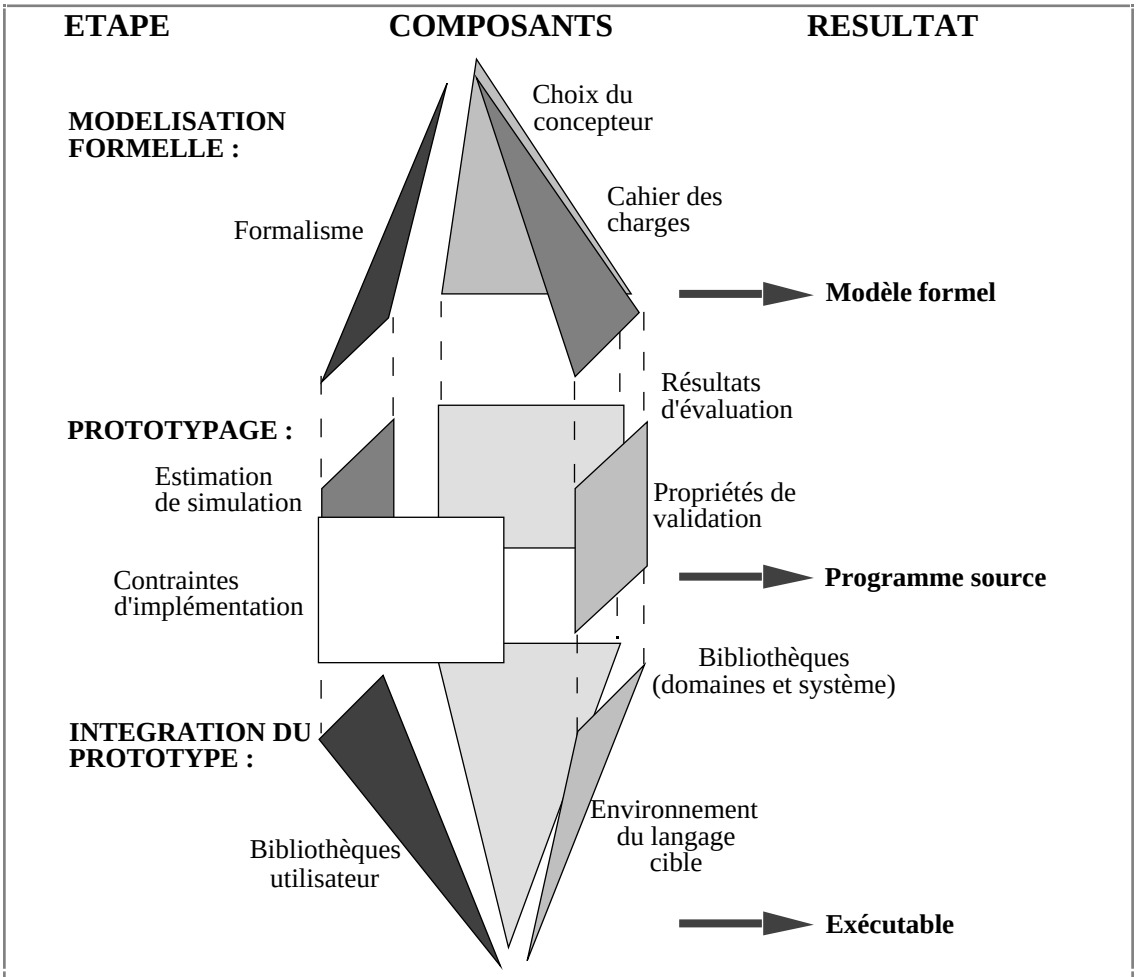


Figure e : Composants d'information dont l'intégration permet d'obtenir une spécification formelle, un programme source puis un programme exécutable.

L'étape de prototypage a été divisée en quatre phases, intégrant chacune une catégorie d'informations supplémentaire :

- la phase d'*Identification*, chargée d'identifier les composants externes afin d'extraire le modèle ouvert;
- la phase de *Caractérisation*, chargée de décomposer le modèle ouvert en G-objets;
- la phase de *Placement*, chargée de proposer un placement adéquat des composantes du prototype sur une architecture donnée;
- la phase de *Génération*, chargée de générer le squelette du prototype, à partir de la décomposition en G-objets et des relations du modèle avec les composants externes.

Les deux premières étapes travaillent sur la spécification du système. Elles sont donc indépendantes du langage cible dans lequel est produit le

prototype. Elles ne sont à appliquer qu'une seule fois, quel que soit le nombre de prototypes différents à produire. Les deux autres étapes sont dépendantes de l'architecture cible et du langage cible utilisé. Elles sont à effectuer pour obtenir chaque prototype.

L'utilisation des mécanismes de traduction progressive liée à la méthodologie permet de limiter la dérive des spécifications.

APPORT FORMEL

L'utilisation, comme formalisme pivot, d'un modèle formel - les *réseaux de Petri bien formés* - nous assure une assise théorique.

Pour générer automatiquement un prototype, nous avons interprété les spécifications formelles qui nous sont transmises en entrée du processus. Nous avons, à cette fin, défini des algorithmes permettant de décomposer un réseau de Petri en objets fonctionnels (G-objets), préfigurant les fonctionnalités du prototype.

Ces algorithmes reposent sur l'étude des invariants de la spécification du système. Cependant, tous les modèles ne sont pas interprétables. Ils doivent être "bien constitués". Cette notion est définie à l'aide de propriétés étroitement liées aux techniques de décomposition.

A partir de ces propriétés, nous divisons les réseaux de Petri en différentes classes :

- les réseaux "prototypables" qui sont "bien constitués" et, par conséquent, décomposables en G-objets;
- les réseaux "transformables" qui peuvent être automatiquement transformés en modèles "prototypables";
- les réseaux "non-prototypables", pour lesquels nos techniques de décomposition échouent.

Seul ce travail théorique a permis d'automatiser l'interprétation du modèle en termes d'informations directement assimilables à la production de code.

La décomposition de la spécification s'effectue sur le modèle ouvert, c'est-àdire la spécification du système sans son environnement.

APPORT GÉNIE LOGICIEL

L'étude de la production d'un logiciel souligne les deux problèmes suivants :

- l'étude du comportement d'un système doit être effectuée en fonction de l'environnement dans lequel il évolue;
- l'utilisation de réseaux de Petri dans la spécification de systèmes complexes peut s'avérer lourde; les modèles devenant très importants, une sorte de "modélisation séparée", par analogie à la compilation séparée, doit être envisagée.

Notre solution repose sur la notion de composant externe décrite dans le Chapitre III. Un composant externe constitue une abstraction permettant de décrire une portion d'environnement. Cette description est utilisable pour la validation comportementale du modèle comme pour le prototypage. L'application générée automatiquement s'intègre dans son environnement conformément à la spécification réseau de Petri. La notion d'environnement reste suffisamment large pour que les composants externes soient utilisés dans des buts différents :

- elle peut correspondre à des dispositifs externes pour lesquels aucune supposition, concernant leur réalisation, ne peut être effectuée;
- elle peut correspondre à une autre partie du système dont elle décrit les interfaces.

Pour le prototypage, on distingue en général deux classes de méthodes : le maquettage et la génération de programmes.

Dans le premier cas, le prototype n'est qu'une maquette astreinte à un environnement donné. Il est possible d'extraire toutes sortes d'informations mais l'implémentation reste à effectuer avec les moyens traditionnels. Les risques de mauvaise interprétation du modèle ne sont donc pas complètement éliminés.

Dans l'autre approche, le prototype est utilisable en dehors de l'environnement qui l'a produit. La conception d'un système se fait donc uniquement par l'intermédiaire de sa spécification. Les risques dus à une mauvaise interprétation des spécifications sont quasiment inexistants.

Nous nous sommes résolument placés dans l'esprit de la génération de programmes. Elle permet la conception par raffinements successifs d'un système sur la base des informations suivantes :

- Une validation qualitative, possible si la spécification est exprimée de façon formelle. De tels résultats sont importants car ils permettent de caractériser avec certitude les propriétés du système.
- Une validation quantitative, possible grâce à l'étude de l'exécution des versions successives du prototype. Le faible coût de réalisation d'un prototype rend ces informations accessibles.

APPORT TECHNIQUE

Nous avons spécifié de façon précise les différents modules fonctionnels (associés aux G-objets) composant un prototype en ne tenant compte que de contraintes très générales, propres à un grand nombre de langages de programmation. Ainsi, notre démarche est applicable pour tout langage procédural permettant la gestion du parallélisme.

Nous nous sommes ensuite attachés à concrétiser le résultat de nos travaux par des réalisations, dans le but d'éprouver notre méthodologie. Cela nous a amené :

- à appliquer la réalisation des différents modules fonctionnels du prototype dans un langage donné : Ada;

- à concevoir et réaliser un générateur automatique de programmes, selon la méthodologie que nous avons définie.

L'application des principes d'organisation d'un prototype en Ada nous a amené à développer des mécanismes propres à ce langage. Nous avons également envisagé deux types d'exécution : mono-processeur (prototype centralisé) et multi-processeurs (prototype réparti).

Une telle démarche nous a permis de démontrer la cohérence de l'organisation fonctionnelle d'un prototype et la pertinence des G-objets que nous caractérisons. En effet, alors que les problèmes posés par la réalisation des deux prototypes entraînent des choix de réalisation différents, l'organisation des modules fonctionnels n'est pas affectée.

PERSPECTIVES

La version du générateur de code permettant de produire un prototype centralisé est quasiment opérationnelle. Elle fera l'objet d'une démonstration, dans le cadre de la treizième Conférence Internationale sur les Applications et la Théorie des Réseaux de Petri (Sheffield, U.K., 22-26 Juin 1992).

A partir de cette version, nous envisageons d'entamer une campagne de mesures afin d'évaluer les performances du prototype généré. Parallèlement, plusieurs modifications, tant d'un point de vue d'optimisation que d'extension, seront effectuées. Elles concerneront en particulier les points suivants :

- L'actuelle décomposition en G-objets repose sur des hypothèses encore restrictives. La caractérisation formelle de certaines caractéristiques de G-objets (Actions fork, join, locales, différenciées, hybrides) déjà introduite dans le Chapitre II, permettra d'étendre la classe des réseaux "prototypables". Pour cela, il faudra également intégrer de manière formelle ces nouvelles fonctionnalités au prototype.
- Dans la perspective de produire automatiquement des prototypes répartis, nous envisageons d'étendre le générateur actuel. Un placement des composantes du prototype sur les différents sites d'une architecture est également envisagé, sur la base d'une analyse plus complète du modèle.
- La méthodologie n'intégrant les spécificités du langage cible que dans les étapes ultimes, nous envisageons de l'appliquer à la génération d'autres langages cibles (C++...).

Afin d'éprouver les limites de notre générateur, en particulier sur les aspects liés aux composants externes et à l'expression des comportements synchrones, nous envisageons de traiter une application réelle. Il s'agit d'une application de robotique [Le Roch 90, Le Roch 91] mettant en œuvre plusieurs robots coopérants, selon un modèle producteur-consommateur avec conflit potentiel sur des aires de production. Ces travaux s'inscrivent dans le cadre du projet MARSALA regroupant différentes équipes des laboratoires confédérés de l'Institut Blaise Pascal (LADL, LITP, LAFORIA et MASI).

		Introduction générale
Chapitre I	:	*Prototypage de systèmes parallèles*
Chapitre II	:	*Interprétation sémantique d'un réseau de Petri*
Chapitre III	:	*Méthodologie de prototypage*
Chapitre IV	:	*Structure du prototype généré*
Chapitre V	:	*Caractéristiques du prototype Ada*
Chapitre VI	:	*Le prototype centralisé*
Chapitre VII	:	*Le prototype réparti*
Chapitre VIII	:	*CPN/TAGADA : L'outil de prototypage*
		Conclusion générale
+		*Bibliographie*
		Annexes & Appendices

Bibliographie

[Ada 83] Ada Join Program Office, *"Reference Manual for the Ada Programming Language, ANSI/MIL-STD-1815A"*, U.S. Department of Defense, The Pentagon, Washington, January 1983.

[Ada 91a] Ada Join Program Office, *"Ada 9x Mapping Document : volume I, Mapping Rationale"*, U.S. Department of Defense, The Pentagon, Washington, August 1991.

[Ada 91b] Ada Join Program Office, *"Ada 9x Mapping Document : volume II, Mapping Specification"*, U.S. Department of Defense, The Pentagon, Washington, August 1991.

[Amarger 88] S. Amarger, *"Calcul du graphe d'Etat au plus tôt d'un réseau de Petri T-temporisé : intégration à l'atelier AMI"*, rapport technique (DEA), Laboratoire MASI, Juillet 1988.

[Arbaoui 91] S.Arbaoui & F. Oquendo, "Où en est la modélisation du processus de production du logiciel?", *Software engineering & its applications*, Toulouse, 9-13 Décembre 1991.

[Arnold 92] A. Arnold, J. Beauquier, B. Bérard & B. Rozoy, *"Programmes parallèles : modèles et validation"*, Armand Colin, Paris, 1992.

[Arthaud 89] R. Arthaud & A. Roan, "AGE: Maîtriser la construction des systèmes temps réels", *Proceedings of the Second International Workshop on software engineering and its applications*, Toulouse, France, Décembre 1989.

[Atkinson 88] C. Atkinson & S.J. Goldsak, *"Communication between Ada Programs in DIADEM"*, Ada Letters, vol VIII, N°7, 1988.

[Avizienis 87] A. Avizienis, M. Koptz & J.C. Darie (editors), *"The evolution of fault-tolerant computing"*, Springer-Verlag, 1987.

[Ayache 85] J.M.Ayache, J.P.Courtiat, M.Diaz & G.Juanole, *"Utilisation des réseaux de Petri pour la modélisation et la validation de protocole"*, TSI vol 4 N°1, Janvier 1985.

[Bachatene 92] H. Bachatene & P. Estraillier, *"Specification of large distributed systems integrating oriented objects concepts"*, 12th World Computer Congress IFIP 92 "From research to Practice", Madrid, Spain, September 1992.

[Bazalgette 91] G. Bazalgette, D. Békélé, C. Dernon, M. Filali, J.M. Rigaud & A. Sayah, "STRADA : un système de transformation et de répartition pour le langage Ada", *Software engineering & its applications*, Toulouse, 9-13 Décembre 1991.

[Bellahsene 82] Z. Bellahsene, *"Etude de méthodes d'ordonnancement des opérateurs relationnels dans les systèmes multi-processeurs de bases de données centralisés et répartis"*, Thèse de 3ème cycle, Université Pierre & Marie Curie, Paris, 1982.

[Bernard 88] J.M. Bernard, J.L. Mounier, N. Beldiceanu & S. Haddad, "AMI, an Extensible Petri Net Interactive Workshop", *9th European Workshop on Application and Theory of Petri Nets*, Venice, Italy, June 1988.

[Bernard 89] J.M. Bernard & J.L. Mounier, "L'Atelier Logiciel AMI : Un environnement Multi-Utilisateurs, Multi-Sessions pour une architecture Distribuée", *Séminaire franco-brésilien sur les Systèmes Informatiques Répartis*, Septembre 1989.

[Bernard 90] J.M. Bernard & J.L. Mounier, *"Conception et Mise en Oeuvre d'un environnement système pour la modélisation, l'analyse et la réalisation de*

systèmes informatiques", Thèse de doctorat de l'Université Pierre & Marie Curie, 4 place Jussieu, 75252 Paris Cedex 05, Décembre 90.

[Bernard 91] G. Bernard, D. Steve & M. Simatic, *"Placement et migration de processus dans les systemes repartis faiblement couplés"*, TSI, Mai 1991.

[Bernstein 78] P.A. Bernstein, J.B. Rothnie, H. Goodman & C.A. Papadimitriou, *"The concurrency mechanism of SDD-1 : A system for distributed database (the fully redundant case)"*, IEEE transaction on software engineering, vol SE-4, N°3, May 1978.

[Bernstein 80] P.A. Bernstein & N. Goodman, "Timestamp-Based Algorithm for Concurrency Control in Distributed Database Systems", *6th Very Large Data Bases*, Montréal, October 1980.

[Bernstein 81] P.A. Bernstein & N. Goodman, *"Concurrency Control in Distributed Database Systems"*, ACM Computing Surveys, vol 13, Nº2, June 1981.

[Berthelot 86] G. Berthelot, *"Transformation and decomposition of Nets"*, Advances in Petri Nets, Part I, Springer-Verlag, September 1986.

[Billionnet 89] A. Billionnet, M.C. Costa & A. Sutter, *"Les problèmes de placement dans les systèmes distribués"*, Techniques et Science Informatique, vol 8, N°4, 1989.

[Birell 84] A.D. Birell & B.J. Nelson, *"Implementing remote procedure Call"*, Transaction in Computer Systems, vol 2 N° 1, February 1984.

[Birman 85] K.P. Birman, "Replication and availability in the ISI system", *proceedings of the tenth acm symposium on operating systems principles*, Washington, December 85.

[Birmann 89] K.P. Birmann & T.A. Joseph, *"Exploiting replication in distributed systems"*, chapter 15 in "Distributed Systems", edited by S. Mullender, ACM press, 1989.

[Blanc 90] P. Blanc, *"Détection de propriétés stables dans les systèmes distribués non-fifo : application aux problèmes de terminaison"*, Thèse de l'Université P. & M. Curie, 4 Place Jussieu, 75252 Paris Cedex 05, Mars 1990.

[Boehm 76] B.W. Boehm, *"Software Engineering"*, TRW Software Series, TRW Defense and Space System Group, System Engineering and Integration Division, One Space Park, Redondo Beach, CA, October 1976.

[Boehm 81] B.W. Boehm, *"Software Engineering Economics"*, Prentice-Hall, Englewood Cliffs, New Jersey, U.S.A. 1981.

[Bonnaire 92a] X. Bonnaire, C. Dutheillet & S. Haddad, *"SANDRINE : Système d'ANalyse de Déclaration de Réseau pour INtégration d'Extension"*, Rapport IBP/MASI N°92/20, Institut Blaise Pascal, 4 place Jussieu, 75252 Paris Cedex 05, Mai 1992.

[Bonnaire 92b] X. Bonnaire & F. Kordon, *"Description d'un AMI-net après son analyse syntaxique dans l'environnement AMI"*, Rapport IBP/MASI N°92/21, Institut Blaise Pascal, 4 place Jussieu, 75252 Paris Cedex 05, Mai 1992.

[Booch 86] G. Booch, *"Object Oriented Development"*, IEEE Transactions on Software Engineering, February 1986.

[Booch 87] G. Booch, *"Software Engineering using Ada"*, Benjamin Cummings, 1987.

[Boussand 92] C. Boussand, E. Cavillon, H. Milone & F. Kordon, *"TEPACAP : Transformation Et réPartition d'une Application Codée automatiquement en Ada Pour unix"*, Rapport IBP/MASI N°92/22, Institut Blaise Pascal, 4 place Jussieu, 75252 Paris Cedex 05, Mai 1992.

[Brams 82] G.W. BRAMS, *"Réseaux de Petri : Théorie et Pratique" Tomes I et II*, MASSON, Paris, 1982.

[Bréant 89] F. Bréant, P. Estraillier & M.F. Leroch, "Génération Automatique de code OCCAM à partir d'un réseau de Petri", *Journées AFCET*, Chamonix, Juin 1989.

[Bréant 90] F. Bréant, "Tapioca : OCCAM Rapid Prototyping from Petri Nets", *Proceedings of the Vth Jerusalem Conference on Information Technology*, Jerusalem, Israel, October 1990.

[Brinkkemper 90] S. Brinkkemper & A.H. ter Hosftede, *"The Conceptual Task Model : a Specification Technique between Requirements Engineering and Program Development"*, Advances Information Systems Engineering, Second Nordic Conference CAiSE'90, LNCS N° 436, Springer Verlag, Stockolm, Sweden, May 1990.

[Brunet 91] J. Brunet, "Modelling the World with Semantic Objects", *Proceeding of Working Conf. on The Object Oriented Approach in Information Systems*, Quebec, October 1991.

[Bruno 86] G. Bruno & G. Marchetto, *"Process translatable Petri-Nets for the rapid prototyping of control-systems"*, IEEE Trans on Software Engineering, vol SE-12 Nº2, February 1986.

[Budde 84] R. Budde, K. Kuhlenkamp, L. Mathiassen & H. Züllighoven, *"Approaches to prototyping"*, Springer Verlag, Berlin, 1984.

[Burns 81] J.E. Burns, "Symetry in Systems of Asynchronous Processes", *proceedings of the 22nd Annual symposium on Foundations of Computer Science*, October 1981.

[Burns 90] C. Burns, "PROTO - A software Requirements Specification, Analysis and validation tool", *1st International Workshop on Rapid System Prototyping*, Research Triangle Park, June 1990.

[Burns 91] C. Burns, "Parallel PROTO - A prototyping tool for analyzing and validating sequential and parallel processing software requirements", *2nd International Workshop on Rapid System Prototyping*, Research Triangle Park, June 1991.

[Case 86] A.F. Case, *"Information Systems Development : Principle of Computer-Aided Software Engineering"*, Prentice Hall, Englewood, 1986.

[Cassigneul 91] V. Cassigneul, *"S.A.O. presentation"*, Internal report Nº 463.097/91, Aérospatiale, Toulouse, March 1991.

[Chehaibar 90] G. Chehaibar, "Validation of phase-executed protocols modelled with coloured Petri nets", *Proceedings of the 11th International Conference on Application and Theory of Petri Nets*, Paris, June 1990.

[Chehaibar 91] G. Chehaibar, *"Méthodes d'analyse hiérarchique des réseaux de Petri"*, Thèse de l'Ecole Nationale des Ponts et Chaussées, Paris, Septembre 1991.

[Cheng 90] J. Cheng, *"Lancement d'une application répartie"*, Stage de DEA, Laboratoire MASI, Université P. & M. Curie, Paris, Septembre 1990.

[Chiola 87] G. Chiola, "A graphical PN/tool for performance Analysis", *International Workshop on modelling techniques and performance evaluation,AFCET*, Paris, March 1987.

[Chiola 90] G. Chiola, C. Dutheillet, G. Francheschinis & S. Haddad, "On Well Formed Colored Nets and their Symbolic Reachability Graph", *11th International Conference On Application and Theory of Petri Nets*, Paris, June 1990.

[Chiola 91] G. Chiola, "GreatSPN 1.5 : software architecture", *5th conference on modelling techniques and tools for computer performance evaluation*, Torino, Italy, February 1991.

[Chvatal 83] V. Chvatal, *"Linear Programming : théorie et pratique"*, Masson, Paris, 1983.

[Coad 92] P. Coad & E Yourdon, *"Analyse Orientée Objets"*, Masson & Prentice Hall, Paris, 1992.

[Coffman 71] E.G. Coffman, M.J. Elphick & A. Shoshani, *"System Deadlocks"*, ACM Computing Surveys, vol 3, N°2, June 1971.

[Colom 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, pp207*, 1986.

[Cousin 88a] B. Cousin & P. Estraillier, *"Traduction et Analyse d'un modèle en vue d'une implémentation optimale"*, Rapport MASI N°207, Paris, Janvier 1988.

[Cousin 88b] B. Cousin, P. Estraillier & F. Kordon, "Generation of Ada code from a Petri Net model : an application", *3rd International symposium on Computer & Information sciences*, Izmir, October 1988.

[Couvreur 90a] J.M. Couvreur, S. Haddad & J.F. Peyre, *"Résolution paramétrée d'une famille de systèmes d'équations linéaires à solutions positives"*, Rapport I.B.P, Université P.& M. Curie et C.N.R.S. MASI, 4 place Jussieu, 75252 Paris cedex 05, 1990.

[Couvreur 90b] J.M. Couvreur, *"Extension des méthodes de calcul de flots pour les réseaux de Petri de haut-niveau. Application à la validation de systèmes distribués"*, Thèse de l'Université P. & M. Curie, 4 Place Jussieu, 75252 Paris cedex 05, Décembre 1990.

[Dagron 89] N. Dagron, F. Derrough & A. El Fallah, "Ingénierie à l'aide du système expert MIAMI", *1ère conférence internationale Maghrébine d'intelligence artificielle et de Génie Logiciel*, Constantine, Algérie, Septembre 1989.

[De Marco 78] T. De Marco, *"Structured Analysis and System Specification"*, Prentice Hall, Englewood, 1978.

[Di Giovanni 90] R. Di Giovanni, "Petri Nets and Software Engineering : HOOD Nets", *11th International Conference on Application and Theory of Petri Nets*, Paris, June 1990.

[Dollas 90] A. Dollas & V. Chi, "Rapid System Prototyping in Academic Laboratories of the 1990's", *1st International Workshop on rapid system Prototyping*, Research Triangle Park, North Carolina, June 1990.

[Dutheillet 91] C. Dutheillet, *"Symétries dans les réseaux colorés : Définition, Analyse et Application à l'Evaluation de Performances"*, Thèse de doctorat de l'Université Pierre & Marie Curie, 4 place Jussieu, 75252 Paris Cedex 05, Paris, Janvier 1991.

[El Fallah 89] A. El Fallah, "A knowledge basis for the analysis and the validation of Petri Nets", *5th International Congress on Robotic and Artificial Intelligence, North Holland*, Bratislava, November 1989.

[Estraillier 91] P. Estraillier, *"Modélisation, Analyse et Réalisation de Systèmes Distribués"*, Rapport d'habilitation à diriger des recherches, Université P. & M. Curie, Décembre 1991.

[Estraillier 92] P. Estraillier & J.L. Mounier, "Marsupilami Modélisation, Analyse et Réalisation de Systèmes : Un Projet intégrant l'Atelier de Modélidation Interactive", *proceedings des Journées Firtech-AFCET : "Bases de données et Génie Logiciel"*, Fevrier 1992.

[Eswaran 76] K.P. Eswaran, J.N Gray, R. Lorie & L.L. Traiger, *"The Notion of Consistency and Predicate Locks in a Database System"*, Communication of the A.C.M. vol 19, N°11, November 1976.

[Finkel 90] A. Finkel, "Effective analysis of large Petri Nets with the minimal coverability graph", *Journées du FIRTECH systèmes et télématique, CNET Issy-les-Moulineaux*, Janvier 1990.

[Floyd 84] C. Floyd, *"A systematic look at prototyping"*, in Approaches to prototyping, Springer-Verlag, Berlin, 1984.

[Folliot 89] B. Folliot & M. Ruffin, "GATOS : un gérant de tâches distribué", *Convention UNIX, AFUU*, Paris, Février 1989.

[GAO 79] U.S. Government Accounting Office, *"Contracting for computer software development serious problems require management attention to avoid wasting additionnal millions"*, Rapport FGMS-80-4, November 1979.

[Garcia-Molina 85] H. Garcia-Molina & D. Barbara, *"How to assign votes in a distributed system"*, Journal of the ACM, vol 32, N°4, 1985.

[Garcia-Molina 89] H. Garcia-Molina & A.M. Spauster, *"Message ordering in a multicast environment"*, proceeding of the 9th International IEEE Conference on DCS, June 1989.

[Gardarin 86] G. Gardarin, *"Bases de données : les systèmes et leurs langages"*, Eyrolles, Paris, 1986.

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

[Girault 87] C. Girault, C. Chatelain & S. Haddad, *"Specification and properties of a cache coherence protocol model"*, Advances in Petri Nets, Lecture notes in Computer Science, vol 266, Springer-Verlag 1987.

[Goscinski 91] A. Goscinski, *"Distributed Operating Systems : the logical design", chapter 8*, Addison-Wesley, 1991.

[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 88] S. Haddad, *"Les réseaux de Petri"*, Cours de D.E.A. de l'Université Pierre et Marie Curie, Paris, 1988.

[Haddad 91] S. Haddad, *"Méthodes de vérification des systèmes parallèles"*, Rapport d'habilitation à diriger des recherches, Université P. & M. Curie, 4 place Jussieu, 75252 Paris Cedex 05, Décembre 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.

[Hein 88] J.C. Hein, *"Intégration du logiciel ARP dans l'atelier AMI"*, rapport technique (DEA), Laboratoire MASI, Juillet 1988.

[Heitz 91] M. Heitz, "Développement d'applications réparties avec Ada et HOOD", *Journées AFCET Ada-France*, Paris, Décembre 1991.

[Hoare 74] C.A.R. Hoare, *"Monitors : an operating system structuring concept"*, Communication of the A.C.M. vol 17, N°10, October 1974.

[HOOD 89] European Space Agency and technology center, *"HOOD reference manual, issue 3.0"*, Noordwijk, the Netherlands, September 1989.

[Hubert 90] P. Hubert, K. Jensen & R.M. Shapiro, *"Hierarchies in coloured Petri Nets"*, Advances in Petri Nets, Lecture Notes in Computer Science, vol 483, Springer Verlag, Berlin Heidelberg, New York, 1990.

[Ichbiah 86] J.D. Ichbiah, J.G.P. Barnes, R.J. Firth & M. Woodger, *"Rationale for the design of the Ada programming language"*, Department of Defense, 1986.

[ISO 7498] *"ISO : Information Processing System, Open Systems interconnection, Basic reference model"*, ISO, 1984.

[Jaulent 90] P. Jaulent, *"Génie Logiciel : les méthodes"*, Ed Colin, 1990.

[Jensen 78] K. Jensen & N. Wirth, *"Pascal : user manual and report"*, Springer-Verlag, Berlin, Heidelberg, New York, 1978.

[Jensen 81] K. Jensen, *"Coloured Petri nets and the invariant method"*, T.S.C. 14, pp 317-336, 1981.

[Kordon 89] F. Kordon, P. Estraillier & R. Card, "Génération automatique de code Ada pour la communication entre Processus", *Congrès AFCET Ada-France*, Paris, Décembre 1989.

[Kordon 90] F. Kordon, P. Estraillier & R. Card, "Rapid Ada prototyping : principles and example of a complex application", *9th International Phoenix Conference on Computers and Communications*, Phoenix, April 1990.

[Kordon 91a] F. Kordon & P. Estraillier, "Complex Systems rapid Prototyping and Environment Abstraction", *2nd International Workshop on Rapid System Prototyping*, Triangle Park Institute, June 1991.

[Kordon 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 91c] F. Kordon & P. Sens, "Répartir des programmes Ada sur un ensemble homogène de machines Unix, une expérience de réalisation", *Conférence Internationale Francophone Ada-France : "Ada, premier bilan d'utilisation"*, Novembre 1991.

[Krakowiak 85] S. Krakowiak, *"Principes des systèmes d'exploitation des ordinateurs"*, Dunod-informatique, Paris, 1985.

[Kurtz 90] B.D. Kurtz, D. Ho & T.A. Wall, *"An Object-Oriented Methodology for System Analysis and Specifications"*, HP Journal, April 1990.

[Lamport 78] L. Lamport, *"Time, clock and the ordering of Events in a Distributed System"*, Communication of the ACM, vol 21, N° 7, July 1978.

[Lanord 92] C. Lanord, *"Les normes non conventionnelles en droit international humanitaire"*, thèse de l'université Rennes I, Octobre 1992.

[Lavarenne 89] C. Lavarene & Y. Sorel, *"SYNDEX : un environnement de programmation pour multi-processeurs de traitement du signal"*, Rapport INRIA N°113, Domaine de Voluceau, Rocquencourt BP105, Novembre 1989.

[Le Lann 77] G. Le Lann, "Distributed Systems. Toward a formal approach", *proceedings of the IFIP Congress*, Toronto, August 1977.

[Le Roch 90] M-F. Le Roch & G. Novak, "Moteurs! Acteurs", *proceedings du Deuxième Congrès Francophone de Robotique Pédagogique*, Montreal, Canada, Août 1990.

[Le Roch 91] M-F. Le Roch & G. Novak, "Robotic Communicating Actors", *proceedings of the second France Israel Symposium on Robotics*, Saclay, April 1991.

[Lientz 78] B. Lientz, E. Swanson & G. Tompkins, *"Characteritics of application software maintenance"*, Communication of the ACM, Nº21, June 1978.

[Lintulampi 90] R. Lintulampi & P. Pulli, "Graphics Based Prototyping of real-Time Systems", *1st International Workshop on rapid system Prototyping*, Research Triangle Park, North Carolina, June 1990.

[Loomis 87] M. Loomis, A. Shah & J. Rumbaugh, "An Object Modelling Technique for Conceptual Design", *European Conference on OOP*, June 1987.

[Lo 88] V.M. Lo, *"Euristic Algorithm for task assignment in distributed systems"*, IEEE transaction, Software engineering, vol 37, N° 11, 1988.

[Marzullo 91] K.Marzullo, R.Cooper, M.Wood & K.Birman, *"Tools for distributed application management"*, IEEE computer, August 91.

[Memmi 83] G. Memmi, *"Méthodes d'analyse de réseaux de Petri réseaux à file, application au temps réel"*, Thèse d'Etat de l'Université Pierre et Marie Curie, Laboratoire MASI, 4 place Jussieu, 75252 Paris cedex 05, Juin 1983.

[Meyer 88] B. Meyer, *"Object Oriented Software Construction"*, Prentice Hall, 1988.

[Millard 91] B.R. Millard, D.S. Miller & C. Wu, "Support for Ada Intertask Communication in a Message-Based Distributed Operating System", *proceedings of the 10 th Annual International Phoenix conference on computer communication (IEEE)*, Scottsdale, Arizona, March 1991.

[Moitessier 91] F. Moitessier, *"MISTRAL : Méthodologie Interactive de conception et réalisation des Systèmes de commande Temps réel répartis en Automatique : application à un processsus éLectromécanique"*, Thèse de doctorat de l'Institut National Polytechnique de Lorraine, Nancy, Juin 1991.

[Mullender 89] S. Mullender, *"Introduction"*, chapter 1 in "Distributed Systems", edited by S. Mullender, ACM press, New York, 1989.

[Murata 86] T. Murata, N. Komoda & K. Matsumoto, *"A Petri Nets based Factory Automation controller for flexible and maintainable control specification"*, IEEE Trans on Industrial Electronics, vol IE-33, Nº1, February 1986.

[Murata 90] T. Murata, *"Petri nets : properties, analysis and applications"*, proceedings of the IEEE, vol 6, nº1, pp 39-50, January 1990.

[Murphy 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 81] B.J. Nelson, *"Remote Procedure Call"*, CSL-81-9, Xerox Palo Alto Research Center, 1981.

[Nelson 83] R.A. Nelson, L. M. Haibt & P. B. Sheridan, *"Casting Petri Nets into Programs"*, IEE transaction on software engineering, vol SE-9, September 1983.

[Nelson 90] V.P. Nelson, *"Fault-Tolerant computing : Fundamental concepts"*, Computer, July 1990.

[Paludetto 90] M. Paludetto, R. Valette & M. Courvoisier, "Génération de code Ada à partir d'une approche orientée objets HOOD/réseaux de Petri", *Software engineering and its applications*, Toulouse, France, December 1990.

[Paludetto 91] M. Paludetto, *"Sur la commande de procédés industriels : une méthodologie basée objets et réseaux de Petri"*, Thèse de l'Université Paul Sabatier, Toulouse, Décembre 1991.

[Petersen 90] T. A. Petersen, D. A. Thomae & D. E. Van den Bout, "The Anyboard: a Rapid-Prototyping System for use in teaching Digital Circuit Design", *1st International Workshop on rapid system Prototyping*, Research Triangle Park, North Carolina, June 1990.

[Peterson 81] J.L. Peterson, *"Petri net Theory and the modelling of system"*, Prentice Hall,, 1981.

[Pettrucci 91] L. Pettrucci, *"Technique d'analyse de réseaux de Petri algébriques"*, Thèse de doctorat de l'Université Pierre & Marie Curie, 4 place Jussieu, 75252 Paris cedex 05, Janvier 1991.

[Pomello 90] L.Pomello, *"Refinement of Concurrent Systems based on Local State Transformations"*, Stepwise Refinement of Distributed Systems, REX Workshop, Mook, The Netherlands, June 1989, LNCS N° 430, Springer Verlag, 1990.

[Pujolle 85] G. Pujolle, E. Horlait, D. Dromard & D. Seret, *"Réseaux et télématique"*, *tomes I et II*, Eyrolles, 1985.

[Raynal 84] M. Raynal, *"Algorithmique du parallélisme : le problème de l'exclusion mutuelle"*, Dunod, 1984.

[Raynal 87] M. Raynal, *"Systèmes répartis et réseaux : concepts, outils et algorithmes"*, Eyrolles, 1987.

[Raynal 88] M. Raynal & J.M. Helary, *"Synchronisation et contrôle des systèmes et programmes répartis"*, Eyrolles, 1988.

[Raynal 91] M. Raynal, *"La communication et le temps dans les réseaux et les systèmes répartis"*, Eyrolles, 1991.

[Richard Foy 91] M. Richard-Foy, *"L'optimisation des tâches passives en Ada"*, La lettre Ada, N° 48-49, Novembre 1991.

[Rosenkrantz 77] D.J.Rosenkrantz, R.E.Stearns & P.W.Lewis, *"System level concurrency control for distributed database systems"*, ACM Transactions on Database Systems, vol 2 n°3, September 1977.

[Royal 90] M. Royal, T. Markas, T. Yang & N. Konapoulos, "Creating the IC palette", *1st International Workshop on rapid system Prototyping*, Research Triangle Park, North Carolina, June 1990.

[Sens 90] P. Sens, *"Gestion de communications dans une application de génération de code Ada réparti"*, Stage de DEA (Université P. & M. Curie), Paris, Septembre 1990.

[Shapiro 86] R.M. Shapiro, *"Structures and encapsulation in distributed systems: the Proxy Principle"*, IEEE, Cambridge Mass. (USA), May 1986.

[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.

[Silva 79] M. Silva & R. David, *"Synthèse programmée des automates logiques décrits par réseaux de Petri : une méthode de mise en oeuvre sur microcalculateurs"*, Rairo-Automatique, vol 13, N°4, Novembre 1979.

[Silva 80] M. Silva & S. Velilla, "Sistemas especializado en la simulaciòn de redes de Petri sanas", *Simp. IFAC sobre Modelado y Simulaciòn*, Sevilla, Mayo 1980.

[Silva 82] M. Silva & S. Velilla, "Programmable logic controllers and petri nets", *International symposium IFAC-IFIP on Software Computer Control*, Madrid, September 1982.

[Sinclair 87] J.B. Sinclair, *"Efficient Computation of optimal Assignments for distributed tasks"*, Journal of Parallel et distributed computing, vol 4, 1987.

[Souissi 89] Y. Souissi, "Compositions of nets via a communication medium", *Proceedings of the 10th International Conference on Application and Theory of Petri Nets*, Bonn, Germany, 1989.

[Souissi 90] Y. Souissi, "On liveness preservation by composition of nets via a set of places", *Proceedings of the 11th International Conference on Application and Theory of Petri Nets*, Paris, France, June 1990.

[Stevens 90] W.R. Stevens, *"UNIX Network programming"*, Prentice Hall, 1990.

[Stroustrup 87] B. Stroustrup, "What is Object Oriented Programming", *European Conference on O-O-Programming*, Paris, France, June 1987.

[Taubner 87] D. Taubner, "On the implementation of Petri Nets", *8th European Workshop on Application and Theory of Petri Nets*, Saragosa, Spain, June 1987.

[Theimer 88] M. Theimer & K. Lantz, "Finding Idle Machines in a Workstation-based Distributed System", *8th International Conference on Distributed System*, San Jose, June 13 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.

[Trèves 89] N. Trèves, *"Comparative study of different techniques for semiflow computation in P/T Nets"*, Lecture Notes in Computer Science, Advances in Petri nets N°424, 1989.

[Tu 90] S. Tu, S.M. Shatz & T. Murata, "Applying Petri Net reductions to support Ada-tasking deadlock detection", *proceedings of the IEEE 10th International Conference on Distributed Computing Systems, pp 96-103*, Paris, May-June 1990.

[Valette 82] R. Valette, M. Courvoisier & J.M. Bigou, "Les réseaux d'automates : Analyse de la coopération", *Journées d'études SEE : La pénétration des automates dans les systèmes automatisés*, Gif sur Yvette, 1982.

[Valette 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.

[Van Glabbeck 90] R. Van Glabbeck & U.Goltz, *"Refinement of Actions in Causality Based Models"*, Stepwise Refinement of Distributed Systems, REX Workshop, Mook, The Netherlands, June 1989, LNCS N° 430, Springer Verlag, 1990.

[Vautherin 84] J. Vautherin & G. Memmi, *"Computation of flows for unary Predicate/Transition Nets"*, Advances in Petri Nets, G. Rosenberg ed, Lecture Notes in Computer Science Nª188, Springer Verlag, 1984.

[Vinci 90] P.O. Vinci & R.M. Shapiro, "Development and Implementation of a strategy for electronic funds transfert by mean of hierarchical colored Petri Nets", *Proceedings of the 11th International Conference on Application and Theory of Petri Nets*, Paris, June 1990.

[Vonk 92] R. Vonk, *"Prototypage : l'utilisation efficace de la technologie CASE"*, MASSON & Prentice Hall, Paris, 1992.

[Weihl 89] W.E. Weihl, *"Remote Procedure Call"*, chapter 4 in "Distributed systems", edited by S. Mullender, ACM press, 1989.

[Wirth 83] N. Wirth, *"Programming in Modula-2"*, Springer-Verlag, Berlin, Heidelberg, New York, 1983.

ANNEXES & APPENDICES

			Introduction générale
	Chapitre I	:	*Prototypage de systèmes parallèles*
	Chapitre II	:	*Interprétation sémantique d'un réseau de Petri*
	Chapitre III	:	*Méthodologie de prototypage*
	Chapitre IV	:	*Structure du prototype généré*
	Chapitre V	:	*Caractéristiques du prototype Ada*
	Chapitre VI	:	*Le prototype centralisé*
	Chapitre VII	:	*Le prototype réparti*
	Chapitre VIII	:	*CPN/TAGADA : L'outil de prototypage*
			Conclusion générale
			Bibliographie
+			*Annexes & Appendices*

Annexe A :

Les algorithmes types du prototype

Modèle de tâche déduit d'un Processus....349
Etat_Processus alternatif....349
Action simple....350
Action simple gardée....350
Action synchronisée....351
Structure d'un serveur (Ressources ou synchronisation)....352
Algorithme d'un gardien....353
Mise à jour d'une estampille locale à un serveur de Ressources....354
Production d'un marquage postcondition....355
Annulation d'une demande d'évaluation....356
Réveil des préconditions en attente d'un événement positif....357
Réception d'une demande de réactivation....358
Tenter l'envoi d'un jeton....359
Réception d'une demande de jeton....359
Réception d'un jeton R....359
Réception d'un jeton T....360
Evaluateur d'une précondition....360
Evaluation d'une précondition....361

Nous donnons, dans cet appendice, un certain nombre d'algorithmes du prototype. Pour chacun d'eux, nous donnons une brève fiche décrivant leurs conditions d'application.

MODÈLE DE TÂCHE DÉDUIT D'UN PROCESSUS

Politique : *Pas de choix*	Module concerné
Condition d'utilisation : *pr Pr*	*Processus*..... 7 *Gestion des Ressources*..... *Gestion des Actions synchronisée*.... *Contrôle*.....

```
DEBUT
  ATTENDRE l'initialisation
    Récupérer les paramètres d'initialisation
  TANT QUE continuer FAIRE
    CAS Etat_Courant FAIRE
      Cas Actif 1 : traitements associés à l'état actif 1
      ...
      Cas Actif N : traitements associés à l'état actif N
    FCAS
  FTQ
  Signaler sa terminaison au module de contrôle de l'application
FIN
```

ETAT_PROCESSUS ALTERNATIF

Politique : *Evaluation simultanée des Actions postcondition*	Module concerné
Condition d'utilisation : $p \quad \varepsilon_A (pr)$	*Processus* .....7 *Gestion des Ressources*..... *Gestion des Actions synchronisées*.. *Contrôle* .....

```
DEBUT
   Lancer l'évaluation de l'ensemble des Actions possibles depuis
   l'Etat_Processus
   ATTENDRE la première Action réalisable
   Supprimer l'évaluation des Actions non élues
   Etat_Courant <- Action réalisable
FIN
```

Politique : *Choix a priori d'un Action postcondition*	Module concerné
Condition d'utilisation : $p \quad \varepsilon_A (pr)$	*Processus* .....7 *Gestion des Ressources*..... *Gestion des Actions synchronisées*.. *Contrôle* .....

```
DEBUT
   Etat_Courant <- Fonction_de_tirage (liste d'Actions possibles);

FIN
```

ACTION SIMPLE

Politique : *Pas de choix*	Module concerné
Condition d'utilisation : $p \quad \alpha_P (pr)$ *et* $\Pi (a) = \Pi_p (a)$ *dégénérée*	*Processus* .....7 *Gestion des Ressources*..... *Gestion des Actions synchronisées*.. *Contrôle* .....

```
DEBUT
   Appel de la procédure associée à l'Action
   Produire la postcondition ressource
   Mettre à jour le mot d'état du Processus
   Passer à l'Etat Actif suivant

FIN
```

ACTION SIMPLE GARDÉE

Politique : *Pas de choix*	Module concerné
Condition d'utilisation : - $a \quad \alpha_P (pr)$ *et* $\Pi_p (a) = \{\chi_p (a) , \{p\}\}$ *avec* $p \quad \varepsilon_S (pr)$ *(postcondition d'un Etat_Processus simple.)*	*Processus* .....7 *Gestion des Ressources*..... *Gestion des Actions synchronisées*.. *Contrôle* .....

```
DEBUT
   Demander l'évaluation de la précondition ressource
   ATTENDRE la réalisation de la précondition ressource
   Appel de la procédure associée à l'Action
   Produire la postcondition ressource
   Mettre à jour le mot d'état du Processus
   Passer à l'Etat Actif suivant

FIN
```

Politique : *Choix a priori d'une Action postcondition d'un Etat_Processus alternatif*	Module concerné
Condition d'utilisation : - *a* $\alpha_P(pr)$ *et* $\Pi_p(a) = \{\chi_p(a), \{p\}\}$ *avec* $p \in \varepsilon_A(pr)$ *(postcondition d'un Etat_Processus alternatif)*	*Processus* ..... 7 *Gestion des Ressources*..... *Gestion des Actions synchronisées*... *Contrôle* .....

```
DEBUT
   Evaluer la précondition processus
   SI la précondition processus est vérifiée ALORS
      Demander l'évaluation de la précondition ressource
      ATTENDRE un certain temps sa réalisation
      SI la précondition ressource est vérifiée ALORS
         EFFECTUER_ACTION <- VRAI
      SINON
         EFFECTUER_ACTION <- FAUX
      FSI
   SINON
      EFFECTUER_ACTION <- FAUX
   FSI
   SI EFFECTUER_ACTION ALORS
      Appel de la procédure associée à l'Action
      Produire la postcondition ressource
      Mettre à jour le mot d'état du Processus
      Passer à l'Etat Actif suivant
   SINON
      Revenir à l'Etat Actif prédécesseur (Etat_Processus alternatif)
   FSI

FIN
```

Politique : *Evaluation simultanée des Actions postcondition de l'Etat_Processus alternatif*	Module concerné
Condition d'utilisation : - *a* $\alpha_P(pr)$ *et* $\Pi_p(a) = \{\chi_p(a), \{p\}\}$ *avec* $p \in \varepsilon_A(pr)$ *(postcondition d'un Etat_Processus alternatif)*	*Processus* ..... 7 *Gestion des Ressources*..... *Gestion des Actions synchronisées*... *Contrôle* .....

```
DEBUT
   Appel de la procédure associée à l'Action
   Produire la postcondition ressource
   Mettre à jour le mot d'état du Processus
   Passer à l'Etat Actif suivant

FIN
```

ACTION SYNCHRONISÉE

Politique : *Pas de choix*	Module concerné
Condition d'utilisation : - *a* $\alpha_S(pr)$ *et* $\Pi_t(a, pr) = \{\chi_t(a, pr), \{p\}\}$ *avec* $p \varepsilon_S(pr)$ *(postcondition d'un Etat_Processus simple)*	*Processus* .....7 *Gestion des Ressources*..... *Gestion des Actions synchronisées*.. *Contrôle* .....

```
DEBUT
   Signaler sa présence au module de gestion des Actions synchronisées
   ATTENDRE la réalisation
   Mettre à jour le mot d'état du processus
   Passer à l'Etat Actif suivant

FIN
```

Politique : *Choix a priori d'une Action postcondition de l'Etat_Processus*	Module concerné
Condition d'utilisation : - *a* $\alpha_S(pr)$ *et* $\Pi_t(a, pr) = \{\chi_t(a, pr), \{p\}\}$ *avec* $p \varepsilon_S(pr)$ *(postcondition d'un Etat_Processus alternatif)*	*Processus*.....7 *Gestion des Ressources* ..... *Gestion des Actions synchronisées*... *Contrôle*.....

```
DEBUT
   Signaler sa présence au gestionnaire des Actions synchronisées
   ATTENDRE un certain temps la réalisation de la synchronisation
   SI la synchronisation a eu lieu
   ALORS
      SYNCHRO_REALISEE <- VRAI
   SINON
      SYNCHRO_REALISEE <- FAUX
   FSI
   SI SYNCHRO_REALISEE
   ALORS
      Mettre à jour le mot d'état du processus
      Passer à l'Etat Actif suivant
   SINON
      Revenir à l'Etat Actif précédent
   FSI

FIN
```

Politique : *Evaluation simultanée des Actions postcondition de l'Etat_Processus alternatif*	Module concerné
Condition d'utilisation : - *a* α_S *(pr) et* Π_t *(a, pr) =* $\{\chi_t$ *(a, pr) , {p}} avec p* ε_S *(pr) (postcondition d'un Etat_Processus alternatif)*	*Processus* .....7 *Gestion des Ressources*..... *Gestion des Actions synchronisées*.. *Contrôle* .....

```
DEBUT
   rien

FIN
```

STRUCTURE D'UN SERVEUR (RESSOURCES OU SYNCHRONISATION)

Politique : *Pas de choix*	Module concerné
Condition d'utilisation : *systématique*	*Processus* ..... *Gestion des Ressources*..... 7 *Gestion des Actions synchronisées* 7 *Contrôle* .....

```
DEBUT
  Attendre INITIALISATION
    PHASE_DE_FONCTIONNEMENT <- VRAI
    Effectuer les initialisations dont on a la charge
  TANT QUE PHASE_DE_FONCTIONNEMENT FAIRE
    ACCEPTER
      Service 1
        Traiter le service 1
    OU
      ...
    OU
      Service N
        Traiter le service N
    OU
      Service TERMINER
        PHASE_DE_FONCTIONNEMENT <- FAUX
    FACCEPTER
  FTQ

FIN
```

ALGORITHME D'UN GARDIEN

Politique : *Pas de choix*	Module concerné
Condition d'utilisation : *Prototype centralisé*	*Processus* ..... *Gestion des Ressources*..... *Gestion des Actions synchronisées*.. *Contrôle* 7

```
DEBUT
  Effectuer les initialisations liées au langage cible
  POUR tous les processus du modele FAIRE
    Creer les instances du processus
    POUR toutes les instances d'un processus FAIRE
      Initialiser l'instance
    FPOUR
  FPOUR
  Initialiser la gestion des Ressources
  Initialiser la gestion des Actions synchronisées
  TANT QUE il reste des instances de processus actives OU
           pas (ERREUR) FAIRE
    ATTENDRE que l'on signale la fin d'une exécution
    SI l'instance qui se termine le fait sur une erreur ALORS
      ERREUR <- VRAI
    FSI
  FTQ
  SI erreur ALORS
    Détruire brutalement toutes les tâches du prototype
  SINON
    Terminer la gestion des Ressources
    Terminer la gestion des Actions synchronisées

  FSI

  Effectuer les terminaisons liées au langage cible

FIN
```

Politique : *Pas de choix*	Module concerné
Condition d'utilisation : *Prototype réparti*	*Processus* ..... *Gestion des Ressources*..... *Gestion des Actions synchronisées*.. *Contrôle* 7

```
DEBUT
  ATTENDRE son numéro d'exécutable : N et le nombre d'exécutables : E
  POUR I variant de 1 à N -1 FAIRE
    ATTENDRE une connexion d'identification d'un exécutable de numéro
    inférieur et mettre à jour la table de localisation des
    exécutables
  FPOUR
  POUR I variant de N + 1 à E FAIRE
    Emettre vers l'exécutable I son identification et attendre sa
    réponse et mettre à jour la table de localisation des exécutables
  FPOUR
  Lire les paramètres de placement des tâches et mettre à jour la
  table de localisation des TTR
  Effectuer les initialisations liées au langage cible
  POUR tous les processus du modèle FAIRE
    Creer les instances du processus
    POUR toutes les instances d'un processus FAIRE
      Initialiser l'instance en mode Actif ou Ecran
    FPOUR
  FPOUR
  Initialiser la gestion de Ressources
  POUR toutes les Actions synchronisées FAIRE
    SI l'Action est active dans l'exécutable ALORS
      Initialiser la gestion de l'Action en mode Actif
    SINON
      Initialiser la gestion de l'Action en mode Ecran
    FSI
  FPOUR
  TANT QUE il existe une TTR active dans l'exécutable ou pas (ERREUR)
          FAIRE
    ACCEPTER
      ATTENDRE que l'on signale la fin d'une exécution
      SI l'instance qui se termine le fait sur une erreur ALORS
        ERREUR <- VRAI
      FSI
    OU
      ATTENDRE message d'un autre gardien
      SI c'est un message de terminaison ALORS
        ERREUR <- VRAI
      SINON
        Mettre à jour la table de localisation des TTR ou la
        table de localisation des exécutables
        ERREUR <- FAUX
      FSI
    FACCEPTER
  FTQ
  SI erreur ALORS
    Détruire brutalement toutes les tâches du prototype
    Diffuser un message de terminaison aux autres serveurs
  SINON
    Terminer la gestion des Ressources
    Terminer la gestion des Actions synchronisées
    Signaler aux autres gardiens que l'on se termine
  FS

  Effectuer les terminaisons liées au langage cible

FIN
```

MISE À JOUR D'UNE ESTAMPILLE LOCALE À UN SERVEUR DE RESSOURCES

Politique : *Pas de choix*	Module concerné
Condition d'utilisation : *Prototype réparti*	*Processus* ..... *Gestion des Ressources*..... 7 *Gestion des Actions synchronisées*.. *Contrôle* .....

```
DEBUT
  SI Estampille du message > Compteur_local ALORS
    Compteur_local <- Estampille du message + Estimation
  FSI
FIN
```

PRODUCTION D'UN MARQUAGE POSTCONDITION

Politique : *Pas de choix*	Module concerné
Condition d'utilisation : *Prototype centralisé.*	*Processus* ..... *Gestion des Ressources*..... 7 *Gestion des Actions synchronisées*.. *Contrôle* .....

```
DEBUT
  TANT QUE il reste des composantes à traiter FAIRE
    Produire le marquage dans la Ressource considérée
    Réveiller les préconditions en attente d'un événement positif
    Tenter d'envoyer le jeton au serveur concerné
    Passer à la composante suivante
  FTQ
FIN
```

Politique : *Pas de choix*	Module concerné
Condition d'utilisation : *Prototype réparti, demande émanant d'un module-client.*	*Processus* ..... *Gestion des Ressources*..... 7 *Gestion des Actions synchronisées*.. *Contrôle* .....

```
DEBUT
  TANT QUE il reste des composantes à traiter FAIRE
    SI le jeton R correspondant à la ressource est sur le site ALORS
      Produire le marquage dans la Ressource considérée
      Réveiller les préconditions en attente d'un événement positif
      Tenter d'envoyer le jeton au serveur concerné
    SINON
      SI le jeton T correspondant à la ressource est sur le site
      ALORS
        Ajouter la demande dans le jeton T
      SINON
        Ajouter un élément à une postcondition à diffuser
      FSI
    FSI
    Passer à la composante suivante
  FTQ
  SI il existe des Ressources pour lesquelles ni le jeton R, ni
     le jeton T ne sont sur le site ALORS
    Diffuser la postcondition construite
  FSI
FIN
```

Politique : *Pas de choix*	Module concerné
Condition d'utilisation : *Prototype réparti, demande émanant d'un autre serveur de Ressources.*	*Processus*..... *Gestion des Ressources* ..... 7 *Gestion des Actions synchronisées*... *Contrôle*.....

```
DEBUT
   Décoder le message
   Mettre à jour le compteur d'estampille
   TANT QUE on n'a pas évalué l'ensemble des composantes FAIRE
     SI le serveur possède le jeton R de la composante ALORS
       Produire le marquage
       Réveiller les préconditions en attente d'un événement positif
       SI le jeton T n'est pas présent ALORS
          Mettre l'estampille de la production dans la liste des
          demandes traitées entre l'arrivée du Jeton R et celle du
          jeton T
       FSI
     SINONSI le serveur possède le jeton T ALORS
       Construire une postcondition à une seule composante
       Ajouter cette postcondition à la liste des productions
       à faire suivre
     FSI
     Passer à la composante suivante
   FTQ
FIN
```

ANNULATION D'UNE DEMANDE D'ÉVALUATION

Politique : *Pas de choix*	Module concerné
Condition d'utilisation : *Prototype centralisé.*	*Processus* ..... *Gestion des Ressources*..... 7 *Gestion des Actions synchronisées*.. *Contrôle* .....

```
DEBUT
   SI l'annulation est acceptable ALORS
     POUR toutes les Ressources concernées par la demande FAIRE
       Supprimer l'estampille dans les listes de demandes du jeton R
       (demandes en attente d'évaluation, demandes en attente
       d'un événement positif)
     FPOUR
     Accepter l'annulation
   SINON
     Refuser l'annulation
   FSI
FIN
```

Politique : *Pas de choix*	Module concerné
Condition d'utilisation : *Prototype réparti, demande émanant d'un module-client.*	*Processus* ..... *Gestion des Ressources*..... 7 *Gestion des Actions synchronisées*.. *Contrôle* .....

```
DEBUT
  SI l'annulation est acceptable ALORS
    POUR toutes les Ressources concernées par la demande FAIRE
      SI le jeton R est présent ALORS
        Supprimer l'estampille dans les listes de demandes du jeton
        R (demandes en attente d'évaluation, demandes en attente
        d'un événement positif)
      SINONSI le jeton T est présent ALORS
        Mettre la demande d'annulation dans le jeton T
      SINON
        Ajouter une composante au message de diffusion de
        l'annulation
      FSI
    FPOUR
    SI le message de diffusion de l'annulation n'est pas vide ALORS
      Le diffuser
    FSI
    Accepter l'annulation
  SINON
    Refuser l'annulation
  FSI
FIN
```

Politique : *Pas de choix*	Module concerné
Condition d'utilisation : *Prototype réparti, demande émanant d'un autre serveur de ressources.*	*Processus*..... *Gestion des Ressources* ..... 7 *Gestion des Actions synchronisées*... *Contrôle*.....

```
DEBUT
  Décoder le message
  Mettre à jour le compteur d'estampille
  POUR toutes les Ressources concernées par la demande FAIRE
    SI le jeton R est présent ALORS
      Supprimer l'estampille dans les listes de demandes du jeton
      R (demandes en attente d'évaluation, demandes en attente
      d'un événement positif)
      SI le jeton T n'est pas présent ALORS
        Mettre l'estampille de l'annulation dans la liste des
        demandes traitées entre l'arrivée du jeton R et celle du
        jeton T
      FSI
    SINONSI le jeton T est présent ALORS
      Mettre la demande d'annulation dans le jeton T
    FSI
  FPOUR
FIN
```

RÉVEIL DES PRÉCONDITIONS EN ATTENTE D'UN ÉVÉNEMENT POSITIF

Politique : *Pas de choix*	Module concerné

Condition d'utilisation : *Prototype centralisé*	*Processus* ..... *Gestion des Ressources*..... 7 *Gestion des Actions synchronisées* 7 *Contrôle* .....

```
DEBUT
  TANT QUE il reste des demandes associées à la Ressource en attente
           d'un événement positif FAIRE
    Mettre l'estampille de la demande dans la liste des demandes à
    traiter
    Evaluer la précondition
  FTQ
FIN
```

Politique : *Pas de choix*	Module concerné
Condition d'utilisation : *Prototype réparti*	*Processus* ..... *Gestion des Ressources*..... 7 *Gestion des Actions synchronisées* 7 *Contrôle* .....

```
DEBUT
  POUR toutes les demandes du jeton R associé à la Ressource
       en attente d'un événement positif FAIRE
    POUR tous les jetons R présents dans l'exécutable FAIRE
      SI la demande est référencée ALORS
        La réactiver
        SI elle est référencée dans la liste des demandes
           en attente d'un événement positif ALORS
          La retirer de la liste
        FSI
      FSI
    FPOUR
    Mettre la demande dans la liste des demandes de jeton à
    traiter
    POUR tous les jetons T présents dans l'exécutable FAIRE
      SI le jeton R correspondant n'est pas sur le site ALORS
        Mettre à jour la liste des réactivations redirigées
      FSI
    FPOUR
    Mettre à jour la liste des demandes à réactivation qui sera
    diffusée aux autres serveurs
    TANT QUE la demande la plus prioritaire émane du serveur ET
             tous les jeton R de cette demande sont présents ALORS
      Tenter une évaluation de la précondition
    FSI
  FPOUR
  Tenter d'envoyer les jetons modifiés
FIN
```

RÉCEPTION D'UNE DEMANDE DE RÉACTIVATION

Politique : *Pas de choix*	Module concerné
Condition d'utilisation : *Prototype réparti, message émanant d'un autre serveur de Ressources.*	*Processus* ..... *Gestion des Ressources*..... 7 *Gestion des Actions synchronisées*.. *Contrôle* .....

```
DEBUT
  Décoder le message
  Mettre à jour le compteur d'estampille
  POUR chaque demande contenue dans le message FAIRE
    POUR tous les jeton R présents dans l'exécutable FAIRE
      SI la demande est référencée dans le jeton ALORS
        La réactiver
      FSI
    FPOUR
    POUR tous les jeton T présents dans l'exécutable FAIRE
      SI le jeton R correspondant n'est pas présent ALORS
        Ajouter la demande à la liste des réactivations à rediriger
      FSI
    FPOUR
  Tenter d'envoyer les jetons modifiés
FIN
```

TENTER L'ENVOI D'UN JETON

Politique : *Pas de choix*	Module concerné
Condition d'utilisation : *Prototype réparti*	*Processus* ..... *Gestion des Ressources*..... 7 *Gestion des Actions synchronisées*.. *Contrôle* .....

```
DEBUT
  SI le jeton T correspondant au jeton R est présent dans
     l'exécutable ALORS
    SI la demande non inhibée la plus prioritaire émane d'un autre
       serveur ALORS
      Transmettre le jeton R en direction du serveur concerné
    FSI
  FSI
FIN
```

RÉCEPTION D'UNE DEMANDE DE JETON

Politique : *Pas de choix*	Module concerné
Condition d'utilisation : *Prototype réparti*	*Processus*..... *Gestion des Ressources* ..... 7 *Gestion des Actions synchronisées*... *Contrôle*.....

```
DEBUT
  Décoder le message
  Mettre à jour le compteur d'estampille
  SI le jeton R est dans l'exécutable ALORS
    Ajouter la demande dans le jeton
    Tenter d'envoyer le jeton R
  SINONSI  seul le jeton T est dans l'exécutable ALORS
    Ajouter la demande dans la liste des demandes à rediriger
  FSI
FIN
```

RÉCEPTION D'UN JETON R

Politique : *Pas de choix*	Module concerné

Condition d’utilisation : *systématique*	*Processus* ..... *Gestion des Ressources*..... 7 *Gestion des Actions synchronisées*.. *Contrôle* .....

```
DEBUT
  Décoder le message
  Mettre à jour le compteur d’estampille
  Envoyer un accusé de réception du jeton au serveur émetteur
  SI toutes les Ressources référencées dans la précondition la
     plus prioritaire sont présentes ALORS
    Tenter une évaluation
  FSI
    Tenter d’envoyer le jeton
FIN
```

RÉCEPTION D’UN JETON T

Politique : *Pas de choix*	Module concerné
Condition d’utilisation : *Prototype réparti*	*Processus* ..... *Gestion des Ressources*..... 7 *Gestion des Actions synchronisées*.. *Contrôle* .....

```
DEBUT
  Décoder le message
  Mettre à jour le compteur d'estampille
  POUR toutes les demandes de jeton redirigées dans le jeton T FAIRE
    SI la demande ne figure pas dans le jeton R associé ALORS
      L'ajouter dans la liste des demandes du jeton R
    FSI
  FPOUR
  POUR toutes les demandes d'annulation redirigées dans le jeton T
       FAIRE
    SI la demande figure dans la liste des demandes du jeton R
       associé ALORS
      La supprimer de la liste des demandes du jeton R
    FSI
  FPOUR
  POUR toutes les demandes de production redirigées FAIRE
    SI la demande n'est pas référencée dans la liste des productions
       effectuées depuis l'arrivée du jeton R ALORS
      Produire la postcondition
    FSI
  FPOUR
  Vider le jeton T
  Vider la liste des productions effectuées depuis l'arrivée du
  jeton R dans le jeton R associé
FIN
```

EVALUATEUR D'UNE PRÉCONDITION

Politique : *Pas de choix*	Module concerné

Condition d'utilisation : *Systématique*	*Processus* ..... *Gestion des Ressources*..... 7 *Gestion des Actions synchronisées*.. *Contrôle* .....

```
DEBUT
  CAS le nœud décrivant la précondition
    un nœud comparaison :
      SI on compare le contenu de deux Ressources ALORS
        Effectuer une jointure entre les deux Ressources
      SINONSI on compare une Ressource à une valeur ALORS
        Effectuer une sélection dans la Ressource
      FSI
    un nœud ET :
      RESULTAT_COURANT <- évaluation du premier argument
      TANT QUE il reste des arguments non évalués ET
               le RESULTAT_COURANT n'est pas vide FAIRE
        RESULTAT_COURANT <- RESULTAT_COURANT ∩ évaluation de
        l'argument
        Passer à l'argument suivant
      FTQ
    un nœud OU :
      RESULTAT_COURANT <- vide
      TANT QUE il reste des arguments non évalués FAIRE
        RESULTAT_COURANT <- RESULTAT_COURANT   évaluation de
        l'argument
        Passer à l'argument suivant
      FTQ
  FCAS
FIN
```

EVALUATION D'UNE PRÉCONDITION

Politique : *Pas de choix*	Module concerné
Condition d'utilisation : *Prototype centralisé*	*Processus*..... *Gestion des Ressources* ..... 7 *Gestion des Actions synchronisées*... *Contrôle*.....

```
DEBUT
  Estampiller la demande d'évaluation
  SI evaluation possible ALORS
    SI une Ressource externe est référencée ET
       la primitive associée rend un résultat positif ALORS
      Sélectionner un ensemble de marques cohérent
      Effectuer la production
    SINON
      Signaler que la demande est en attente d'un événement positif
      sur le composant externe associé
    FSI
  SINON
    SI l'échec est dû à une Ressource ALORS
      Signaler que la demande est en attente d'un événement positif
      sur la Ressource
    SINON
      Signaler que la demande est en attente d'un événement positif
      sur l'ensemble des Ressources référencées
    FSI
  FSI
FIN
```

Politique : *Pas de choix*	Module concerné
Condition d'utilisation : *Prototype réparti*	*Processus*..... *Gestion des Ressources* ..... 7 *Gestion des Actions synchronisées*... *Contrôle*.....

```
DEBUT
  Estampiller la demande d'évaluation
  SI les jetons R des Ressources internes référencées sont tous`
     présents ALORS
    SI evaluation possible ALORS
      SI une Ressource externe est référencée ALORS
        SI elle est gérée par le serveur ET
            la primitive associée rend un résultat positif ALORS
          Sélectionner un ensemble de marques cohérent
          Effectuer la production
        SINON
          Diffuser une requête sur la Ressource externe
          SI le responsable de la Ressource externe rend un
             résultat positif ALORS
          Sélectionner un ensemble de marques cohérent
          Effectuer la production
        FSI
      SINON
        Signaler que la demande est en attente d'un événement
        positif sur le composant externe associé
        Réactiver la demande sur les jetons R des Ressources
        référencées
      FSI
    SINON
      SI l'échec est dû à une Ressource ALORS
        Mettre l'estampille dans la liste des demandes en attente
        d'un événement positif sur le jeton R
      SINON
        Mettre l'estampille dans la liste des demandes en attente
        d'un événement positif sur les jetons R de toutes les
```

```
          Ressources référencées
        FSI
        Réactiver la demande sur les jetons R des Ressources
        référencées
      FSI
    SINON
      POUR toutes les Ressources référencées FAIRE
        SI le jeton R est présent ALORS
          Mettre l'estampille de la précondition dans la liste des
          demandes en attente du jeton R
        SINON
          SI le jeton T est présent ALORS
            Mettre l'estampille de la précondition dans la liste des
            demandes de jeton redirigées
          SINON
            Ajouter la ressource dans la demande à diffuser
          FSI
        FSI
      FPOUR
      Diffuser la demande de jeton aux autres serveurs
    FSI
FIN
```

		Introduction générale
Chapitre I	:	*Prototypage de systèmes parallèles*
Chapitre II	:	*Interprétation sémantique d'un réseau de Petri*
Chapitre III	:	*Méthodologie de prototypage*
Chapitre IV	:	*Structure du prototype généré*
Chapitre V	:	*Caractéristiques du prototype Ada*
Chapitre VI	:	*Le prototype centralisé*
Chapitre VII	:	*Le prototype réparti*
Chapitre VIII	:	*CPN/TAGADA : L'outil de prototypage*
		Conclusion générale
		Bibliographie
+		*Annexes & Appendices*

Annexe B :

Extraits du code généré pour un exemple

1. Le modèle....365
2. Exécution du prototype centralisé associé....367
3. Extraits du code généré....368

Code 1 : spécification de l'unité de gestion des traces....368
Code 2 : spécification de l'unité de gestion de la pile des appels de sous-programmes....369
Code 3 : exemple de fonction avec et sans mode DEBUG....369
Code 4 : spécification du paquetage de gestion des communications client-serveur....370
Code 5 : spécification de l'unité de description des données générales du prototype....371
Code 6 : spécification du paquetage de description des classes de couleurs....372
Code 7 : description des domaines de couleurs....373
Code 8 : spécification de l'unité décrivant le marquage du modèle....374
Code 9 : spécification de l'unité décrivant une condition....376
Code 10 : spécification de l'unité de gestion du résultat d'une consommation....380
Code 11 : description des ressources du modèle....382
Code 12 : spécification du gestionnaire des Ressources....383
Code 13 : spécification de l'unité chargée de l'évaluation des requêtes sur les Ressources....384
Code 14 : spécification de l'unité décrivant les opérations élémentaires sur les Ressources....385
Code 15 : spécification de l'unité permettant la manipulation du mot d'état de P1....385
Code 16 : spécification du paquetage décrivant le comportement de P1....386
Code 17 : modèle de tâche associé à P1....386
Code 18 : spécification de l'unité décrivant les Etats_Processus alternatifs de P1....388
Code 19 : spécification de l'unité décrivant les Actions de P1....389
Code 20 : code associé à l'Etat_Processus P1a....389
Code 21 : code associé à la transition Tcons1....390
Code 22 : spécification du paquetage réalisant l'Action synchronisée Tsync....392
Code 23 : spécification du module de contrôle....392
Code 24 : le programme principal....393

1. Le modèle

Nous donnons en Figure B.1 un réseau de Petri que nous utiliserons pour illustrer la génération de code en langage Ada. Le modèle que nous présentons a été dessiné à, l'aide du logiciel MACAO [Bernard 90]. Il s'agit d'un réseau de Petri Bien Formé [Chiola 90, Dutheillet 91] décrit sous AMI à l'aide du formalisme AMI-nets [Bonnaire 92a].

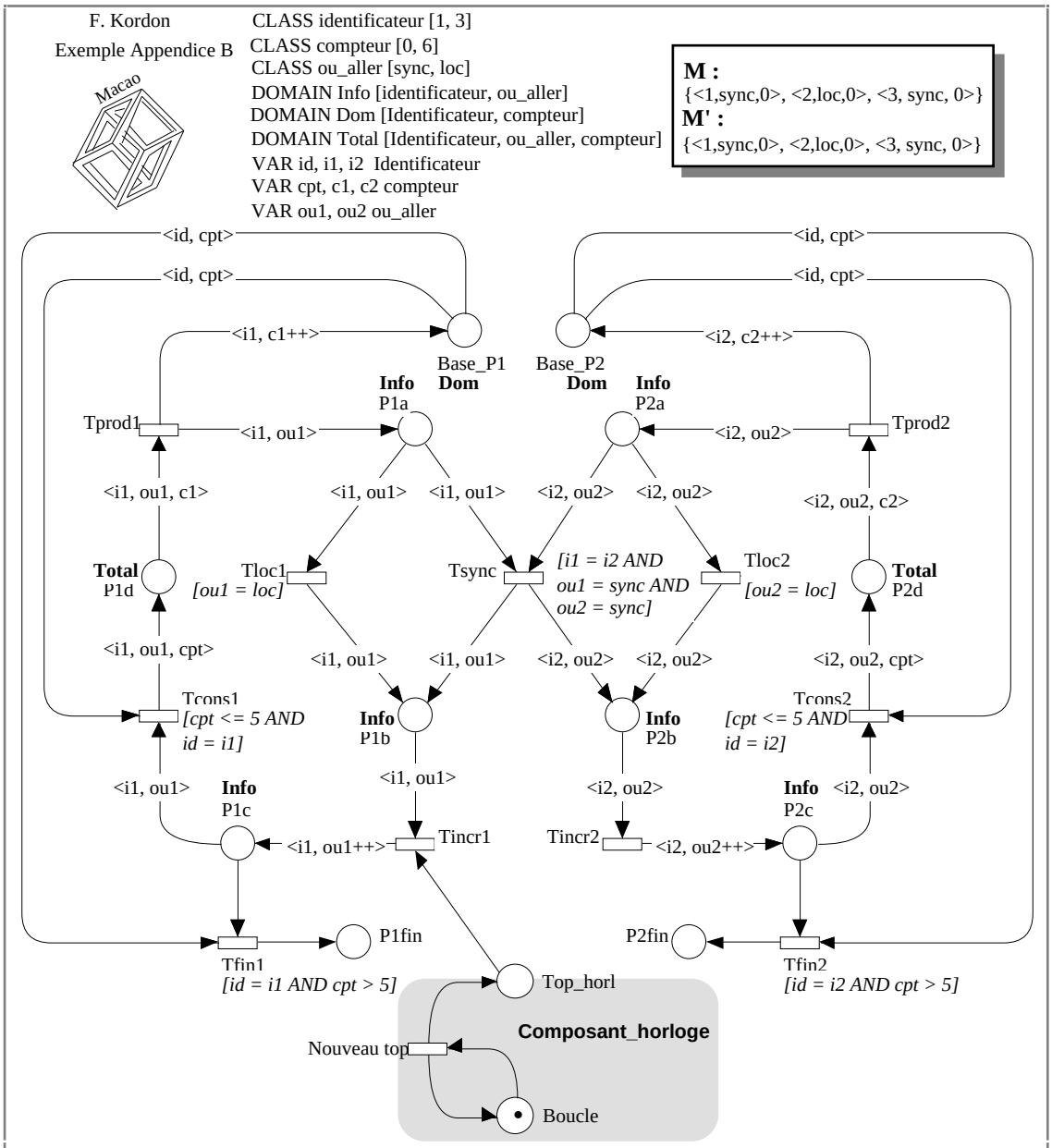


Figure B.1 : Exemple de modèle.

L'algorithme de décomposition, appliqué au modèle de la Figure B.1, caractérise deux Processus pouvant se synchroniser via l'Action *Tsync* . Les G-objets sont récapitulés dans le tableau ci-après.

Objets	Etats_Processus			Actions			Ressources	Processus
	simp.	alt.	de term.	simp.	sync	gard.		
Base_P1							7	
Base_P2							7	
P1a		7						P1
P1b	7							P1
P1c		7						P1
P1d	7							P1
P1fin			7					P1
P2a		7						P2
P2b	7							P2
P2c		7						P2
P2d	7							P2
P2fin			7					P2
Tsync					7	7		P1, P2
Tprod1				7				P1
Tloc1				7		7		P1
Tincr1				7				P1
Tcon1				7		7		P1
Tfin1				7		7		P1
Tprod2				7				P2
Tloc2				7		7		P2
Tincr2				7				P2
Tcons2				7		7		P2
Tfin2				7		7		P2

Le mot d'état de P1 possède trois composantes : *i1*, de type *identificateur*, *ou1*, de type *ou_aller* et *c1*, de type *compteur*. Trois processus instanciés sont définis sur ce modèle. Ils démarrent tous à l'Etat_Processus *P1d*, avec, respectivement, les mots d'états initiaux : <1,sync,0>, <2,loc,0> et <3,sync,0>.

Le mot d'état de P2 possède trois composantes : *i2*, de type *identificateur*, *ou2*, de type *ou_aller* et *c2*, de type *compteur*. Trois processus instanciés sont définis sur ce modèle. Ils démarrent tous à l'Etat_Processus *P2d*, avec, respectivement, les mots d'états initiaux : <1,sync,0>, <2,loc,0> et <3,sync,0>.

Le Processus P1 est relié à un composant externe ne contenant qu'une place d'interface en sortie. Le composant externe est une horloge donnant un signal de temps en temps. Chaque processus instancié de P1 doit attendre la réalisation de ce top pour pouvoir exécuter *Tincr1*.

Les Ressources *Base_P1* et *Base_P2* contiennent le nombre de tours effectué par les instances des Processus P1 et P2. Au sixième tour, chacun des Processus se termine.

Les domaines de couleurs des places sont donnés en caractères gras. Les prédicats associés aux Actions sont donnés en caractères italiques.

Le code généré pour ce modèle, en mode DEBUG, comporte environ 10300 lignes de source Ada (6850 lorsque le mode DEBUG n'est pas positionné).

2. EXÉCUTION DU PROTOTYPE CENTRALISÉ ASSOCIÉ

Si nous considérons que les procédures associées aux Actions affichent "Actions de <*nom de l'action*> pour <mot d'état du processus associé>", une exécution du prototype donne :

```
Actions de Tprod1 pour < 1, OU_ALLER_SYNC, 0>
Actions de Tprod1 pour < 2, OU_ALLER_LOC, 0>
Actions de Tprod1 pour < 3, OU_ALLER_SYNC, 0>
Actions de Tprod2 pour < 1, OU_ALLER_SYNC, 0>
Actions de Tprod2 pour < 2, OU_ALLER_LOC, 0>
Actions de Tprod2 pour < 3, OU_ALLER_SYNC, 0>
Actions de Tloc1 pour < 2, OU_ALLER_LOC, 0>
Actions de Tsync pour P1 < 3, OU_ALLER_SYNC, 0> et P2 < 3, OU_ALLER_SYNC, 0>
Actions de Tincr1 pour < 2, OU_ALLER_LOC, 0>
Actions de Tincr1 pour < 3, OU_ALLER_SYNC, 0>
Actions de Tincr2 pour < 3, OU_ALLER_SYNC, 0>
Actions de Tcons1 pour < 2, OU_ALLER_SYNC, 1>
Actions de Tcons2 pour < 3, OU_ALLER_LOC, 1>
Actions de Tprod1 pour < 2, OU_ALLER_SYNC, 1>
Actions de Tprod2 pour < 3, OU_ALLER_LOC, 1>
Actions de Tsync pour P1 < 1, OU_ALLER_SYNC, 0> et P2 < 1, OU_ALLER_SYNC, 0>
Actions de Tincr1 pour < 1, OU_ALLER_SYNC, 0>
Actions de Tincr2 pour < 1, OU_ALLER_SYNC, 0>
Actions de Tloc2 pour < 3, OU_ALLER_LOC, 1>
Actions de Tincr2 pour < 3, OU_ALLER_LOC, 1>
Actions de Tcons1 pour < 1, OU_ALLER_LOC, 1>
Actions de Tprod1 pour < 1, OU_ALLER_LOC, 1>
Actions de Tcons2 pour < 3, OU_ALLER_SYNC, 2>
Actions de Tprod2 pour < 3, OU_ALLER_SYNC, 2>
Actions de Tloc1 pour < 1, OU_ALLER_LOC, 1>
Actions de Tincr1 pour < 1, OU_ALLER_LOC, 1>
Actions de Tcons1 pour < 1, OU_ALLER_SYNC, 2>
Actions de Tprod1 pour < 1, OU_ALLER_SYNC, 2>
Actions de Tloc2 pour < 2, OU_ALLER_LOC, 0>
Actions de Tincr2 pour < 2, OU_ALLER_LOC, 0>
Actions de Tcons2 pour < 1, OU_ALLER_LOC, 1>
Actions de Tprod2 pour < 1, OU_ALLER_LOC, 1>
Actions de Tcons1 pour < 3, OU_ALLER_LOC, 1>
Actions de Tprod1 pour < 3, OU_ALLER_LOC, 1>
Actions de Tloc2 pour < 1, OU_ALLER_LOC, 1>
Actions de Tloc1 pour < 3, OU_ALLER_LOC, 1>
Actions de Tincr2 pour < 1, OU_ALLER_LOC, 1>
Actions de Tincr1 pour < 3, OU_ALLER_LOC, 1>
Actions de Tcons2 pour < 1, OU_ALLER_SYNC, 2>
Actions de Tprod2 pour < 1, OU_ALLER_SYNC, 2>
Actions de Tsync pour P1 < 1, OU_ALLER_SYNC, 2> et P2 < 1, OU_ALLER_SYNC, 2>
Actions de Tincr1 pour < 1, OU_ALLER_SYNC, 2>
Actions de Tincr2 pour < 1, OU_ALLER_SYNC, 2>
Actions de Tcons1 pour < 1, OU_ALLER_LOC, 3>
Actions de Tprod1 pour < 1, OU_ALLER_LOC, 3>
Actions de Tloc1 pour < 1, OU_ALLER_LOC, 3>
Actions de Tincr1 pour < 1, OU_ALLER_LOC, 3>
Actions de Tcons2 pour < 1, OU_ALLER_LOC, 3>
Actions de Tprod2 pour < 1, OU_ALLER_LOC, 3>
Actions de Tloc2 pour < 1, OU_ALLER_LOC, 3>
Actions de Tincr2 pour < 1, OU_ALLER_LOC, 3>
Actions de Tcons1 pour < 3, OU_ALLER_SYNC, 2>
Actions de Tprod1 pour < 3, OU_ALLER_SYNC, 2>
Actions de Tsync pour P1 < 3, OU_ALLER_SYNC, 2> et P2 < 3, OU_ALLER_SYNC, 2>
Actions de Tincr1 pour < 3, OU_ALLER_SYNC, 2>
Actions de Tincr2 pour < 3, OU_ALLER_SYNC, 2>
Actions de Tcons2 pour < 3, OU_ALLER_LOC, 3>
Actions de Tprod2 pour < 3, OU_ALLER_LOC, 3>
Actions de Tloc2 pour < 3, OU_ALLER_LOC, 3>
Actions de Tincr2 pour < 3, OU_ALLER_LOC, 3>
Actions de Tcons1 pour < 1, OU_ALLER_SYNC, 4>
Actions de Tprod1 pour < 1, OU_ALLER_SYNC, 4>
Actions de Tcons2 pour < 2, OU_ALLER_SYNC, 1>
Actions de Tprod2 pour < 2, OU_ALLER_SYNC, 1>
Actions de Tsync pour P1 < 2, OU_ALLER_SYNC, 1> et P2 < 2, OU_ALLER_SYNC, 1>
Actions de Tincr1 pour < 2, OU_ALLER_SYNC, 1>
Actions de Tincr2 pour < 2, OU_ALLER_SYNC, 1>
Actions de Tcons1 pour < 3, OU_ALLER_LOC, 3>
Actions de Tprod1 pour < 3, OU_ALLER_LOC, 3>
Actions de Tloc1 pour < 3, OU_ALLER_LOC, 3>
Actions de Tincr1 pour < 3, OU_ALLER_LOC, 3>
Actions de Tcons2 pour < 1, OU_ALLER_SYNC, 4>
Actions de Tcons1 pour < 3, OU_ALLER_SYNC, 4>
Actions de Tprod2 pour < 1, OU_ALLER_SYNC, 4>
Actions de Tprod1 pour < 3, OU_ALLER_SYNC, 4>
Actions de Tsync pour P1 < 1, OU_ALLER_SYNC, 4> et P2 < 1, OU_ALLER_SYNC, 4>
Actions de Tincr1 pour < 1, OU_ALLER_SYNC, 4>
Actions de Tincr2 pour < 1, OU_ALLER_SYNC, 4>
Actions de Tcons2 pour < 2, OU_ALLER_LOC, 2>
Actions de Tprod2 pour < 2, OU_ALLER_LOC, 2>
Actions de Tloc2 pour < 2, OU_ALLER_LOC, 2>
Actions de Tincr2 pour < 2, OU_ALLER_LOC, 2>
Actions de Tcons2 pour < 2, OU_ALLER_SYNC, 3>
Actions de Tcons1 pour < 1, OU_ALLER_LOC, 5>
```

```
Actions de Tprod2 pour < 2, OU_ALLER_SYNC, 3>
Actions de Tprod1 pour < 1, OU_ALLER_LOC, 5>
Actions de Tloc1 pour < 1, OU_ALLER_LOC, 5>
Actions de Tincr1 pour < 1, OU_ALLER_LOC, 5>
Actions de Tfin1 pour < 1, OU_ALLER_SYNC, 5>
Terminaison de P1. 1
Actions de Tcons1 pour < 2, OU_ALLER_LOC, 2>
Actions de Tprod1 pour < 2, OU_ALLER_LOC, 2>
Actions de Tloc1 pour < 2, OU_ALLER_LOC, 2>
Actions de Tincr1 pour < 2, OU_ALLER_LOC, 2>
Actions de Tcons2 pour < 3, OU_ALLER_SYNC, 4>
Actions de Tprod2 pour < 3, OU_ALLER_SYNC, 4>
Actions de Tsync pour P1 < 3, OU_ALLER_SYNC, 4> et P2 < 3, OU_ALLER_SYNC, 4>
Actions de Tincr1 pour < 3, OU_ALLER_SYNC, 4>
Actions de Tincr2 pour < 3, OU_ALLER_SYNC, 4>
Actions de Tcons1 pour < 3, OU_ALLER_LOC, 5>
Actions de Tprod1 pour < 3, OU_ALLER_LOC, 5>
Actions de Tloc1 pour < 3, OU_ALLER_LOC, 5>
Actions de Tincr1 pour < 3, OU_ALLER_LOC, 5>
Actions de Tfin1 pour < 3, OU_ALLER_SYNC, 5>
Terminaison de P1. 3
Actions de Tcons1 pour < 2, OU_ALLER_SYNC, 3>
Actions de Tprod1 pour < 2, OU_ALLER_SYNC, 3>
Actions de Tsync pour P1 < 2, OU_ALLER_SYNC, 3> et P2 < 2, OU_ALLER_SYNC, 3>
Actions de Tincr1 pour < 2, OU_ALLER_SYNC, 3>
Actions de Tincr2 pour < 2, OU_ALLER_SYNC, 3>
Actions de Tcons2 pour < 2, OU_ALLER_LOC, 4>
Actions de Tprod2 pour < 2, OU_ALLER_LOC, 4>
Actions de Tloc2 pour < 2, OU_ALLER_LOC, 4>
Actions de Tincr2 pour < 2, OU_ALLER_LOC, 4>
Actions de Tcons2 pour < 1, OU_ALLER_LOC, 5>
Actions de Tprod2 pour < 1, OU_ALLER_LOC, 5>
Actions de Tcons1 pour < 2, OU_ALLER_LOC, 4>
Actions de Tprod1 pour < 2, OU_ALLER_LOC, 4>
Actions de Tloc1 pour < 2, OU_ALLER_LOC, 4>
Actions de Tincr1 pour < 2, OU_ALLER_LOC, 4>
Actions de Tcons1 pour < 2, OU_ALLER_SYNC, 5>
Actions de Tprod1 pour < 2, OU_ALLER_SYNC, 5>
Actions de Tcons2 pour < 2, OU_ALLER_SYNC, 5>
Actions de Tprod2 pour < 2, OU_ALLER_SYNC, 5>
Actions de Tsync pour P1 < 2, OU_ALLER_SYNC, 5> et P2 < 2, OU_ALLER_SYNC, 5>
Actions de Tincr1 pour < 2, OU_ALLER_SYNC, 5>
Actions de Tincr2 pour < 2, OU_ALLER_SYNC, 5>
Actions de Tfin1 pour < 2, OU_ALLER_LOC, 5>
Actions de Tfin2 pour < 2, OU_ALLER_LOC, 5>
Terminaison de P1. 2
Terminaison de P2. 2
Actions de Tcons2 pour < 3, OU_ALLER_LOC, 5>
Actions de Tprod2 pour < 3, OU_ALLER_LOC, 5>
Actions de Tloc2 pour < 3, OU_ALLER_LOC, 5>
Actions de Tincr2 pour < 3, OU_ALLER_LOC, 5>
Actions de Tfin2 pour < 3, OU_ALLER_SYNC, 5>
Terminaison de P2. 3
Actions de Tloc2 pour < 1, OU_ALLER_LOC, 5>
Actions de Tincr2 pour < 1, OU_ALLER_LOC, 5>
Actions de Tfin2 pour < 1, OU_ALLER_SYNC, 5>
Terminaison de P2. 1
```

3. EXTRAITS DU CODE GÉNÉRÉ

Les extraits présentés dans cet appendice sont issus du prototype *généré automatiquement*, à l'aide de l'outil CPN/TAGADA, du modèle de la Figure B.1.

Code 1 : spécification de l'unité de gestion des traces

```
package UTILITAIRES_DE_TRACE is

   procedure TRACE_AVEC_RC (TEXTE : in STRING);

   procedure TRACE_SANS_RC (TEXTE : in STRING);

   procedure INITIALISE_TRACES;

   procedure TERMINE_TRACES;

end UTILITAIRES_DE_TRACE;
```

Code 2 : spécification de l'unité de gestion de la pile des appels de sous-programmes

```
package MECANISME_DE_DIAGNOSTIC is

   procedure POSITIONNER_DIAGNOSTIC (NOM_ROUTINE : in STRING;
                                     EXCEPTION_LEVEE : in STRING;
                                     MESSAGE_COMPL : in STRING := "");

   procedure RESTITUER_DIAGNOSTIC;

end MECANISME_DE_DIAGNOSTIC;
```

Code 3 : exemple de fonction avec et sans mode DEBUG

```
   function L_EXEMPLE_APPLIQUE_SUCCESSEUR (MARQUE : in MARQUAGE_L_EXEMPLE;
                                           RG_SUCC : in POSITIVE := 1) return
                                                        MARQUAGE_L_EXEMPLE is
   VAL_RET : MARQUAGE_L_EXEMPLE := MARQUE;

   begin

      -- Trace puisque DEBUG a ete selectionne en mode DEVELOPPEUR
      if TRACE then
         TRACE_AVEC_RC ("--> L_EXEMPLE_APPLIQUE_SUCCESSEUR (MARQUE," &
                        INTEGER'IMAGE (RG_SUCC) & ")");
      end if;

      -- Apres la trace, les instructions

      case MARQUE.DOMA is
         when IDENT_L_EXEMPLE_OU_ALLER =>
            VAL_RET.C_L_EXEMPLE_TT_VAL_CLASSES_ENUM :=
               L_EXEMPLE_SUCC_OU_ALLER (VAL_RET.C_L_EXEMPLE_TT_VAL_CLASSES_ENUM,
                                        RG_SUCC);
         when IDENT_L_EXEMPLE_IDENTIFICATEUR =>
            VAL_RET.C_L_EXEMPLE_INTEGER :=
               L_EXEMPLE_SUCC_IDENTIFICATEUR (VAL_RET.C_L_EXEMPLE_INTEGER,
                                              RG_SUCC);
         when IDENT_L_EXEMPLE_COMPTEUR =>
            VAL_RET.C_L_EXEMPLE_INTEGER :=
               L_EXEMPLE_SUCC_COMPTEUR (VAL_RET.C_L_EXEMPLE_INTEGER, RG_SUCC);
         when others =>
            -- Il y a un probleme, cela n'a pas de sens pour un domaine
            -- ou une marque non coloree
            POSITIONNER_DIAGNOSTIC ("L_EXEMPLE_APPLIQUE_SUCCESSEUR",
                                    "L_EXEMPLE_PANIQUE_MARQUAGE",
                                    "Fonction inapplicable a une marque qui " &
                                    "n'est pas de type classe");
            raise L_EXEMPLE_PANIQUE_MARQUAGE;
      end case;

      -- Trace puisque DEBUG a ete selectionne en mode DEVELOPPEUR
      if TRACE then
         TRACE_AVEC_RC ("<-- L_EXEMPLE_APPLIQUE_SUCCESSEUR");
      end if;

      return VAL_RET;

   -- Generer la pile des appels de procedure en cas d'erreur
   exception
      when others =>
         POSITIONNER_DIAGNOSTIC ("L_EXEMPLE_APPLIQUE_SUCCESSEUR", "");
         raise;
   end L_EXEMPLE_APPLIQUE_SUCCESSEUR;
```

Code 4 : spécification du paquetage de gestion des communications client-serveur

```
with DESCRIPTION_MOT_D_ETAT_DE_L_EXEMPLE_P1,
     DESCRIPTION_MOT_D_ETAT_DE_L_EXEMPLE_P2,
     L_EXEMPLE_DESCRIPTION_RESULTAT_CONSOMMATION,
     L_EXEMPLE_DEFINITIONS_POUR_LES_PROCESSUS;

use  DESCRIPTION_MOT_D_ETAT_DE_L_EXEMPLE_P1,
     DESCRIPTION_MOT_D_ETAT_DE_L_EXEMPLE_P2,
     L_EXEMPLE_DESCRIPTION_RESULTAT_CONSOMMATION,
     L_EXEMPLE_DEFINITIONS_POUR_LES_PROCESSUS;

package L_EXEMPLE_SYSTEME_D_INTERFACES is

   -- Les differentes categories d'attente d'un evenement

   type L_EXEMPLE_LES_ATTENTES is (
      DEFINITIVEMENT,
      MOMENTANEMENT);

   -- Initialisation du mecanisme de communication pour l'ensemble des instances
   -- d'un processus du modele

   procedure L_EXEMPLE_INIT_SYSTEME_D_INTERFACE
                 (NOMBRE_DE_PROCESSUS : in POSITIVE);

   -- Initialisation du mecanisme de communication pour l'ensemble des processus
   -- du modele

   procedure L_EXEMPLE_INIT_NB_INSTANCES (LE_PROCESSUS : in PROCESSUS_DE_L_EXEMPLE;
                                          NB_D_INSTANCES : in POSITIVE);

   -- Creer un lien de communication entre une instance et les serveurs

   procedure L_EXEMPLE_INIT_INTERFACES_POUR_UNE_INSTANCE
                (IDENTITE_DE_L_INSTANCE : in IDENTITE_L_EXEMPLE);

   -- Detruire brutalement tous les liens de communications

   procedure L_EXEMPLE_TUE_TOUS_LES_LIENS;

   -- Primitive par laquelle une instance d'un processus se bloque en attente du
   -- resultat de l'évaluation d'une precondition

   procedure L_EXEMPLE_J_ATTEND_PRECONDITION
                    (IDENTITE_DE_L_INSTANCE : in IDENTITE_L_EXEMPLE;
                     RESULTAT : out L_EXEMPLE_RESULTAT_CONSOMMATION;
                     CATEGORIE_D_ATTENTE : in L_EXEMPLE_LES_ATTENTES);

   -- Primitive de reveil d'un client sur fin d'evaluation d'une precondition

   procedure L_EXEMPLE_EVENEMENT_RESSOURCE
                   (CLIENT_A_REVEILLER : in IDENTITE_L_EXEMPLE;
                    RESULTAT_A_TRANSMETTRE : in L_EXEMPLE_RESULTAT_CONSOMMATION);

   -- Primitive de reveil d'un client sur fin de synchronisation pour P1

   procedure L_EXEMPLE_EVENEMENT_SYNCHRO_POUR_P1
                   (CLIENT_A_REVEILLER : in IDENTITE_L_EXEMPLE;
                    MOT_D_ETAT_A_TRANSMETTRE : in MOT_D_ETAT_L_EXEMPLE_P1);

   -- Attendre le resultat d'une synchronisation pour une instance de P1

   procedure L_EXEMPLE_J_ATTEND_SYNCHRO_POUR_P1
                   (IDENTITE_DE_L_INSTANCE : in IDENTITE_L_EXEMPLE;
                    MOT_D_ETAT : out MOT_D_ETAT_L_EXEMPLE_P1;
                    CATEGORIE_D_ATTENTE : in L_EXEMPLE_LES_ATTENTES;
                    REALISATION : out BOOLEAN);

   -- Primitive de reveil d'un client sur fin de synchronisation pour P2

   procedure L_EXEMPLE_EVENEMENT_SYNCHRO_POUR_P2
```

```
                    (CLIENT_A_REVEILLER : in IDENTITE_L_EXEMPLE;
                     MOT_D_ETAT_A_TRANSMETTRE : in MOT_D_ETAT_L_EXEMPLE_P2);

   -- Attendre le resultat d'une synchronisation pour une instance P2

   procedure L_EXEMPLE_J_ATTEND_SYNCHRO_POUR_P2
                    (IDENTITE_DE_L_INSTANCE : in IDENTITE_L_EXEMPLE;
                     MOT_D_ETAT : out MOT_D_ETAT_L_EXEMPLE_P2;
                     CATEGORIE_D_ATTENTE : in L_EXEMPLE_LES_ATTENTES;
                     REALISATION : out BOOLEAN);

end L_EXEMPLE_SYSTEME_D_INTERFACES;
```

Code 5 :spécification de l'unité de description des données générales du prototype

```
package L_EXEMPLE_DEFINITIONS_POUR_LES_PROCESSUS is

   -- Le type permettant d'estampiller des demandes

   subtype ESTAMPILLE_L_EXEMPLE is NATURAL;

   -- Valeur par defaut d'une estampille (fait office de "pas d'estampille")

   PAS_D_ESTAMPILLE_L_EXEMPLE : constant ESTAMPILLE_L_EXEMPLE := 0;

   -- Le type permettant d'identifier les classes de processus

   type PROCESSUS_DE_L_EXEMPLE is (
      L_EXEMPLE_ID_PROC_P1,
      L_EXEMPLE_ID_PROC_P2);

   -- Le type "Etat Actif universel"

   type ETAT_ACTIFS_DE_L_EXEMPLE is (

         -- Les Etats Actifs de P1

      P1_E_ACTIF_P1A,
      P1_E_ACTIF_P1C,
      P1_E_ACTIF_TLOC1,
      P1_E_ACTIF_TINCR1,
      P1_E_ACTIF_TFIN1,
      P1_E_ACTIF_TCONS1,
      P1_E_ACTIF_TPROD1,
      P1_E_ACTIF_TSYNC,

         -- Les Etats Actifs de P2

      P2_E_ACTIF_P2A,
      P2_E_ACTIF_P2C,
      P2_E_ACTIF_TLOC2,
      P2_E_ACTIF_TINCR2,
      P2_E_ACTIF_TFIN2,
      P2_E_ACTIF_TCONS2,
      P2_E_ACTIF_TPROD2,
      P2_E_ACTIF_TSYNC);

   -- Declaration du type permettant d'identifier un client demandant un service

   type IDENTITE_L_EXEMPLE is
      record
         CLASSE_DE_PROCESSUS : PROCESSUS_DE_L_EXEMPLE;
         NUMERO_D_INSTANCE : POSITIVE;
      end record;

   -- Transformer une estampille en chaine de caracteres (DEBUG)

   function L_EXEMPLE_ESTAMPILLE_VERS_CHAINE
          (ESTAMPILLE : in ESTAMPILLE_L_EXEMPLE) return STRING;

   -- Transformer une identite en chaine de caracteres (DEBUG)

   function L_EXEMPLE_IDENTITE_VERS_CHAINE
          (IDENTITE : in IDENTITE_L_EXEMPLE) return STRING;

   -- Creation d'un identificateur d'instance d'une classe de Processus

   function L_EXEMPLE_CREER_IDENTITE
          (CATEGORIE_PROCESS : in PROCESSUS_DE_L_EXEMPLE;
           NUMERO_INSTANCE : in POSITIVE) return IDENTITE_L_EXEMPLE;

   -- Recuperer la classe de processus dans un identificateur

   function L_EXEMPLE_RECUPERER_TYPE_PROCESSUS
          (INDENTIFICATEUR : in IDENTITE_L_EXEMPLE) return PROCESSUS_DE_L_EXEMPLE;
```

```
   -- Recuperer le numero de l'instance du processus dans un identificateur

   function L_EXEMPLE_RECUPERER_INSTANCE
          (IDENTIFICATEUR : in IDENTITE_L_EXEMPLE) return POSITIVE;

   -- Provoquer une election de tache Ada (probleme lie a la preemption dans
   -- certains compilateurs)

   procedure L_EXEMPLE_PROVOQUE_ELECTION;

end L_EXEMPLE_DEFINITIONS_POUR_LES_PROCESSUS;
```

Code 6 : spécification du paquetage de description des classes de couleurs

```
package L_EXEMPLE_DESCRIPTION_CLASSES is

   -- Description des Classes enumerees du modele

   -- Declaration du type decrivant toutes les valeurs des classes enumerees

   type L_EXEMPLE_TT_VAL_CLASSES_ENUM is (
      L_EXEMPLE_SANS_VALEUR_OU_ALLER, -- Convention SANS_VALEUR pour OU_ALLER
      OU_ALLER_SYNC,
      OU_ALLER_LOC,
      L_EXEMPLE_ETOILE_CLASSE_ENUM);

   -- Declaration des classes enumerees

   subtype L_EXEMPLE_OU_ALLER is L_EXEMPLE_TT_VAL_CLASSES_ENUM
                              range L_EXEMPLE_SANS_VALEUR_OU_ALLER .. OU_ALLER_LOC;

   -- Declaration des classes de type entier
   -- Attention, la premiere valeur de l'intervalle est inutilisable
   -- La borne inferieure a ete modifiee pour autoriser la creation
   -- de "sans_valeur"

   subtype L_EXEMPLE_IDENTIFICATEUR is INTEGER range INTEGER'PRED (1) .. 3;
   L_EXEMPLE_SANS_VALEUR_IDENTIFICATEUR : constant L_EXEMPLE_IDENTIFICATEUR
                                        := INTEGER'PRED (1);

   subtype L_EXEMPLE_COMPTEUR is INTEGER range INTEGER'PRED (0) .. 6;
   L_EXEMPLE_SANS_VALEUR_COMPTEUR : constant L_EXEMPLE_COMPTEUR
                                    := INTEGER'PRED (0);

end L_EXEMPLE_DESCRIPTION_CLASSES;
```

Code 7 : description des domaines de couleurs

```
with L_EXEMPLE_DESCRIPTION_CLASSES;

use  L_EXEMPLE_DESCRIPTION_CLASSES;

package L_EXEMPLE_DESCRIPTION_DOMAINES is

   -- Declaration de types incomplets en cas de references croisees

   type L_EXEMPLE_INFO;
   type L_EXEMPLE_DOM;
   type L_EXEMPLE_TOTAL;

   -- Declararation du domaine Info

   type L_EXEMPLE_INFO is
      record
         INFO_CHAMP_1 : L_EXEMPLE_IDENTIFICATEUR;
         INFO_CHAMP_2 : L_EXEMPLE_OU_ALLER;
      end record;

   -- Convention SANS_VALEUR pour Info

   L_EXEMPLE_SANS_VALEUR_INFO : constant L_EXEMPLE_INFO := (
      INFO_CHAMP_1 => L_EXEMPLE_SANS_VALEUR_IDENTIFICATEUR,
      INFO_CHAMP_2 => L_EXEMPLE_SANS_VALEUR_OU_ALLER);

   -- Declararation du domaine Dom

   type L_EXEMPLE_DOM is
      record
         DOM_CHAMP_1 : L_EXEMPLE_IDENTIFICATEUR;
         DOM_CHAMP_2 : L_EXEMPLE_COMPTEUR;
      end record;

   -- Convention SANS_VALEUR pour Dom

   L_EXEMPLE_SANS_VALEUR_DOM : constant L_EXEMPLE_DOM := (
      DOM_CHAMP_1 => L_EXEMPLE_SANS_VALEUR_IDENTIFICATEUR,
      DOM_CHAMP_2 => L_EXEMPLE_SANS_VALEUR_COMPTEUR);

   -- Declararation du domaine Total

   type L_EXEMPLE_TOTAL is
      record
         TOTAL_CHAMP_1 : L_EXEMPLE_IDENTIFICATEUR;
         TOTAL_CHAMP_2 : L_EXEMPLE_OU_ALLER;
         TOTAL_CHAMP_3 : L_EXEMPLE_COMPTEUR;
      end record;

   -- Convention SANS_VALEUR pour Total

   L_EXEMPLE_SANS_VALEUR_TOTAL : constant L_EXEMPLE_TOTAL := (
      TOTAL_CHAMP_1 => L_EXEMPLE_SANS_VALEUR_IDENTIFICATEUR,
      TOTAL_CHAMP_2 => L_EXEMPLE_SANS_VALEUR_OU_ALLER,
      TOTAL_CHAMP_3 => L_EXEMPLE_SANS_VALEUR_COMPTEUR);

end L_EXEMPLE_DESCRIPTION_DOMAINES;
```

Code 8 : spécification de l'unité décrivant le marquage du modèle

```
with L_EXEMPLE_DESCRIPTION_CLASSES,
     L_EXEMPLE_DESCRIPTION_DOMAINES;

use  L_EXEMPLE_DESCRIPTION_CLASSES,
     L_EXEMPLE_DESCRIPTION_DOMAINES;

package L_EXEMPLE_DESCRIPTION_MARQUAGE is

   -- Definition de l'exception L_EXEMPLE_PANIQUE_MARQUAGE
```

```
   L_EXEMPLE_PANIQUE_MARQUAGE : exception;

   -- Les differents types de comparaisons possibles

   type LES_COMPARAISONS is (
      EGALITE,
      SUPERIEUR,
      INFERIEUR,
      SUPERIEUR_OU_EGAL,
      INFERIEUR_OU_EGAL,
      DIFFERENT);

   -- Type enumeratif permettant d'identifier un domaine ou une classe

   type LISTE_DE_DOMAINES_L_EXEMPLE is (
      IDENT_L_EXEMPLE_INFO,
      IDENT_L_EXEMPLE_DOM,
      IDENT_L_EXEMPLE_TOTAL,
      IDENT_L_EXEMPLE_IDENTIFICATEUR,
      IDENT_L_EXEMPLE_COMPTEUR,
      IDENT_L_EXEMPLE_OU_ALLER,
      IDENT_L_EXEMPLE_MONOCHROME);

   -- Types decrivant une marque et un marquage

   type MARQUE_L_EXEMPLE (DOMA : LISTE_DE_DOMAINES_L_EXEMPLE);

   type MARQUAGE_L_EXEMPLE is access MARQUE_L_EXEMPLE;

   type MARQUE_L_EXEMPLE (DOMA : LISTE_DE_DOMAINES_L_EXEMPLE) is
      record
         IDENT_MARQUE_SUIVANTE : MARQUAGE_L_EXEMPLE;
         IDENT_COMPTEUR_MARQUE : NATURAL := 1;
         case DOMA is
            when IDENT_L_EXEMPLE_INFO =>
               C_L_EXEMPLE_INFO : L_EXEMPLE_INFO;
            when IDENT_L_EXEMPLE_DOM =>
               C_L_EXEMPLE_DOM : L_EXEMPLE_DOM;
            when IDENT_L_EXEMPLE_TOTAL =>
               C_L_EXEMPLE_TOTAL : L_EXEMPLE_TOTAL;
            when IDENT_L_EXEMPLE_OU_ALLER =>
               C_L_EXEMPLE_TT_VAL_CLASSES_ENUM : L_EXEMPLE_TT_VAL_CLASSES_ENUM;
            when IDENT_L_EXEMPLE_IDENTIFICATEUR |
                 IDENT_L_EXEMPLE_COMPTEUR =>
               C_L_EXEMPLE_INTEGER : INTEGER;
            when IDENT_L_EXEMPLE_MONOCHROME =>
               null;
         end case;
      end record;

   -- constante “marquage vide”

   MARQUAGE_NULL_L_EXEMPLE : constant MARQUAGE_L_EXEMPLE := null;

   -- Primitives de manipulation de marques

   -- Creation d'une marque

   function CREER_MARQUE_L_EXEMPLE
         (TP_MRQ : in LISTE_DE_DOMAINES_L_EXEMPLE := IDENT_L_EXEMPLE_MONOCHROME;
          CREER_MARQUE_P_INFO : in L_EXEMPLE_INFO := L_EXEMPLE_SANS_VALEUR_INFO;
          CREER_MARQUE_P_DOM : in L_EXEMPLE_DOM := L_EXEMPLE_SANS_VALEUR_DOM;
          CREER_MARQUE_P_TOTAL : in L_EXEMPLE_TOTAL := L_EXEMPLE_SANS_VALEUR_TOTAL;
          CREER_MARQUE_P_TT_VAL_CLASSES_ENUM : in L_EXEMPLE_TT_VAL_CLASSES_ENUM
                                               := L_EXEMPLE_ETOILE_CLASSE_ENUM;
          CREER_MARQUE_P_INTEGER : in INTEGER := INTEGER'FIRST;
          NBR_EX : in POSITIVE := 1) return MARQUAGE_L_EXEMPLE;

   -- Destruction d'une marque
```

```
   procedure DETRUIRE_MARQUE_L_EXEMPLE (P_M : in out MARQUAGE_L_EXEMPLE);

   -- Destruction d'une liste de marques

   procedure DETRUIRE_LST_MARQUE_L_EXEMPLE (L_M : in out MARQUAGE_L_EXEMPLE);

   -- Ajouter une marque dans une liste

   function ACCROCHER_MARQUE_L_EXEMPLE (LST_M : in MARQUAGE_L_EXEMPLE;
                                        NEW_M : in MARQUAGE_L_EXEMPLE)
                                                  return MARQUAGE_L_EXEMPLE;

   -- Recuperer le profil suivant dans une liste

   function MARQUE_SUIVANTE_L_EXEMPLE
                   (P_M : in MARQUAGE_L_EXEMPLE) return MARQUAGE_L_EXEMPLE;

   -- Creation d'un masque ETOILE pour un domaine donne au format d'une marque

   function L_EXEMPLE_MASQUE_ETOILE (T_DOMAINE : in LISTE_DE_DOMAINES_L_EXEMPLE;
                                     NBRE : in POSITIVE := 1)
                                                  return MARQUAGE_L_EXEMPLE;

   -- Recuperer le nombre d'exemplaires d'une marque pour un format donne

   function L_EXEMPLE_NB_EXEMPLAIRES
                   (FORMAT_DE_MARQUE : in MARQUAGE_L_EXEMPLE) return POSITIVE;

   -- Dupliquer une marque (avec, par defaut, remise a 1 du compteur d'exemplaires)

   function L_EXEMPLE_DUPLIQUE_UNE_MARQUE
          (MARQUE_A_DUPLIQUER : in MARQUAGE_L_EXEMPLE;
           PROFIL_SEULEMENT : in BOOLEAN := TRUE) return MARQUAGE_L_EXEMPLE;

   -- Appliquer la fonction SUCCESSEUR a une marque

   -- ATTENTION : Cette fonction ne peut s'appliquer qu'a une marque de "type"
   --             classe. Dans le cas contraire, on levera
   --             L_EXEMPLE_PANIQUE_MARQUAGE

   function L_EXEMPLE_APPLIQUE_SUCCESSEUR
                   (MARQUE : in MARQUAGE_L_EXEMPLE;
                    RG_SUCC : in POSITIVE := 1) return MARQUAGE_L_EXEMPLE;

   -- Appliquer la fonction PREDECESSEUR a une marque

   -- ATTENTION : Cette fonction ne peut s'appliquer qu'a une marque de "type"
   --             classe. Dans le cas contraire, on levera
   --             L_EXEMPLE_PANIQUE_MARQUAGE

   function L_EXEMPLE_APPLIQUE_PREDECESSEUR
                   (MARQUE : in MARQUAGE_L_EXEMPLE;
                    RG_PREDE : in POSITIVE := 1) return MARQUAGE_L_EXEMPLE;

   -- Visualiser une marque sous la forme d'une chaine de caracteres (mode DEBUG)

   function L_EXEMPLE_MARQUE_VERS_CHAINE
                   (MARQUE : in MARQUAGE_L_EXEMPLE) return STRING;

   -- Affichage d'une liste de marques

   procedure L_EXEMPLE_AFFICHE_LST_MARQUES (LST_MARQUES : in MARQUAGE_L_EXEMPLE);

   -- Recuperer un champ donne d'une marque composee (0 = toute la marque)

   function L_EXEMPLE_GET_CHAMP (MARQUE : in MARQUAGE_L_EXEMPLE;
                                 NUM_CHP : in NATURAL) return MARQUAGE_L_EXEMPLE;

   -- Comparaison d'un masque a une marque

   function L_EXEMPLE_MASQUE_CORRESPOND
```

```
                    (MARQUE_1 : in MARQUAGE_L_EXEMPLE;
                     MARQUE_2 : in MARQUAGE_L_EXEMPLE;
                     COMPARAISON : in LES_COMPARAISONS := EGALITE) return BOOLEAN;

end L_EXEMPLE_DESCRIPTION_MARQUAGE;
```

Code 9 : spécification de l'unité décrivant une condition

```
with L_EXEMPLE_DESCRIPTION_MARQUAGE,
     L_EXEMPLE_DESCRIPTION_RESSOURCE;

use  L_EXEMPLE_DESCRIPTION_MARQUAGE,
     L_EXEMPLE_DESCRIPTION_RESSOURCE;

package L_EXEMPLE_DESCRIPTION_CONDITION is

   -- Definition de l'exception L_EXEMPLE_PANIQUE_DESCRIPTION_CONDITION

   L_EXEMPLE_PANIQUE_DESCRIPTION_CONDITION : exception;

   -- Le type permettant de decrire les modificateurs dans une precondition

   type LES_MODIFICATEURS is (
      PAS_DE_MODIFICATEUR,
      MODIF_PREDECESSEUR,
      MODIF_SUCCESSEUR);

   -- Les operations possibles dans une condition

   type L_EXEMPLE_OPERATEURS_CONDITION is (
      OPERATION_OU,
      OPERATION_ET,
      OPERATION_COMPARAISON);

   -- Les differents types de noeuds dans l'arbre de description d'une condition

   type L_EXEMPLE_TYP_ND_ARBRE_DESC_COND is (
      OPERATEUR_CONDITION,
      VALEUR_IMMEDIATE,
      DESIGNATEUR_RESSOURCE);

   -- Les conditions sont representees sous une forme arborescente

   -- Description d'un noeud de l'arbre

   type L_EXEMPLE_NOEUD_DESC_CONDITION
                     (CATEGORIE : L_EXEMPLE_TYP_ND_ARBRE_DESC_COND);

   type L_EXEMPLE_DESC_CONDITION is access L_EXEMPLE_NOEUD_DESC_CONDITION;

   type L_EXEMPLE_NOEUD_DESC_CONDITION
                     (CATEGORIE : L_EXEMPLE_TYP_ND_ARBRE_DESC_COND) is
      record
         SUIVANT : L_EXEMPLE_DESC_CONDITION;
         case CATEGORIE is
            when OPERATEUR_CONDITION =>
               OPERATEUR : L_EXEMPLE_OPERATEURS_CONDITION;
               OPERANDES : L_EXEMPLE_DESC_CONDITION;
               TYPE_CMP_SI_CMP : LES_COMPARAISONS := EGALITE;
            when VALEUR_IMMEDIATE =>
               VALEUR_IMMEDIATE : MARQUAGE_L_EXEMPLE;
            when DESIGNATEUR_RESSOURCE =>
               RESSOURCE : L_EXEMPLE_RESSOURCES;
               -- Si NUM_CHAMP vaut 0, alors on designe l'ensemble de la ressource
               NUM_CHAMP : NATURAL := 0;
               MODIFICATEUR : LES_MODIFICATEURS := PAS_DE_MODIFICATEUR;
               RANG_MODIF : POSITIVE := 1;
         end case;
      end record;

   -- La constante representant un arbre de description de condition vide

   L_EXEMPLE_ARBRE_CONDITION_VIDE : constant L_EXEMPLE_DESC_CONDITION := null;

   -- Le descripteur de conditions

   type L_EXEMPLE_CONDITION is
      record
```

```
         UNE_INTERFACE : BOOLEAN := FALSE;
         NOM_INTERFACE : L_EXEMPLE_INTERFACES;
         MRQ_INTERFACE : MARQUAGE_L_EXEMPLE := MARQUAGE_NULL_L_EXEMPLE;
         -- utilise pour les postcondition
         COND_RESSOURCE : L_EXEMPLE_DESC_CONDITION
                          := L_EXEMPLE_ARBRE_CONDITION_VIDE;
      end record;

   -- Creation d'un noeud de type OPERATEUR

   function L_EXEMPLE_CREATION_NOEUD_OP
                     (OPERATEUR : in L_EXEMPLE_OPERATEURS_CONDITION;
                      ARGUMENT : in L_EXEMPLE_DESC_CONDITION;
                      TYP_CMP_SI_CMP : in LES_COMPARAISONS := EGALITE)
                                       return L_EXEMPLE_DESC_CONDITION;

   -- Creation d'un noeud de type RESSOURCE

   function L_EXEMPLE_CREATION_NOEUD_RESS (RESSOURCE : in L_EXEMPLE_RESSOURCES;
                                           NUM_CHAMP : in NATURAL := 0;
                                           MODIFICATEUR : in LES_MODIFICATEURS
                                                          := PAS_DE_MODIFICATEUR;
                                           RANG_MODIF : in POSITIVE := 1)
                                                return L_EXEMPLE_DESC_CONDITION;

   -- Creation d'un noeud de type VALEUR

   function L_EXEMPLE_CREATION_NOEUD_VAL
                     (VALEUR : in MARQUAGE_L_EXEMPLE)
                                       return L_EXEMPLE_DESC_CONDITION;

   -- Ajouter un argument a un OPERATEUR
   -- L'argument est toujours mis en derniere position

   function L_EXEMPLE_ACCROCHER_ARGUMENT_OP
                     (NOEUD_OP : in L_EXEMPLE_DESC_CONDITION;
                      ARGUMENT : in L_EXEMPLE_DESC_CONDITION)
                                          return L_EXEMPLE_DESC_CONDITION;

   -- Passer au frere immediatement a droite dans l'arbre (argument suivant d'un
   -- noeud operateur)

   function L_EXEMPLE_ARG_SUIVANT_DSC_COND (NOEUD : in L_EXEMPLE_DESC_CONDITION)
                                           return L_EXEMPLE_DESC_CONDITION;

   -- Recuperer le nombre d'arguments d'un noeud-operateur

   function L_EXEMPLE_NB_ARG_OPERATEUR
                     (NOEUD_OPERATEUR : in L_EXEMPLE_DESC_CONDITION)
                                       return NATURAL;

   -- Recuperer le Nieme argument d'un noeud-operateur

   function L_EXEMPLE_NIEME_ARGUMENT
                     (NOEUD_OPERATEUR : in L_EXEMPLE_DESC_CONDITION;
                      POSITION_ARGUMENT : in POSITIVE := 1)
                                       return L_EXEMPLE_DESC_CONDITION;

   -- Indiquer le nom de la place d'interface participant a la condition

   -- ATTENTION : Il ne peut y avoir plus d'une place d'interface par condition.
   --             En cas de de probleme, on leve
   --             L_EXEMPLE_PANIQUE_DESCRIPTION_CONDITION

   function L_EXEMPLE_AJOUTE_INTERF_DANS_COND (CONDI : in L_EXEMPLE_CONDITION;
                                               INTERF : in L_EXEMPLE_INTERFACES;
                                                                                  MASQUE :
MARQUAGE_NULL_L_EXEMPLE) return L_EXEMPLE_CONDITION;

   -- Affecter l'arbre decrivant la condition au descripteur de condition
```

```
   function L_EXEMPLE_AJOUTE_ARBRE_DESC_DANS_COND
                    (CONDI : in L_EXEMPLE_CONDITION;
                     ARBRE : in L_EXEMPLE_DESC_CONDITION)
                                        return L_EXEMPLE_CONDITION;

   -- Recuperer l'arbre decrivant une condition

   function L_EXEMPLE_DONNE_ARBRE_DESC_DE_COND
                    (COND : in L_EXEMPLE_CONDITION)
                                        return L_EXEMPLE_DESC_CONDITION;

   -- Recuperer le nom de la ressource designee dans un noeud-ressource

   -- ATTENTION : Si le type du noeud ne correspond pas, on leve
   --             L_EXEMPLE_PANIQUE_DESCRIPTION_CONDITION

   function L_EXEMPLE_DONNE_NOM_RESS_NOEUD_RESS
                    (NOEUD_RESSOURCE : in L_EXEMPLE_DESC_CONDITION)
                                        return L_EXEMPLE_RESSOURCES;

   -- Recuperer le numero du champ de la ressource designee dans un noeud-ressource

   -- ATTENTION : Si le type du noeud ne correspond pas, on leve
   --             L_EXEMPLE_PANIQUE_DESCRIPTION_CONDITION

   function L_EXEMPLE_DONNE_NUM_CHAMP_NOEUD_RESS
                    (NOEUD_RESSOURCE : in L_EXEMPLE_DESC_CONDITION)
                                        return NATURAL;

   -- Recuperer l'operateur associe a un noeud-operateur

   -- ATTENTION : Si le type du noeud ne correspond pas, on leve
   --             L_EXEMPLE_PANIQUE_DESCRIPTION_CONDITION

   function L_EXEMPLE_DONNE_OP_NOEUD_OPERATEUR
                    (NOEUD_OPERATEUR : in L_EXEMPLE_DESC_CONDITION)
                                        return L_EXEMPLE_OPERATEURS_CONDITION;

   -- Recuperer le type de comparaison si c'est une comparaison

   -- ATTENTION : Si le type du noeud ne correspond pas, on leve
   --             L_EXEMPLE_PANIQUE_DESCRIPTION_CONDITION

   function L_EXEMPLE_DONNE_TYPE_COMPARAISON
                    (NOEUD_OPERATEUR : in L_EXEMPLE_DESC_CONDITION)
                                        return LES_COMPARAISONS;

   -- Recuperer le modificateur associe a un noeud-ressource

   -- ATTENTION : Si le type du noeud ne correspond pas, on leve
   --             L_EXEMPLE_PANIQUE_DESCRIPTION_CONDITION

   function L_EXEMPLE_DONNE_MODIFICATEUR
                    (NOEUD_RESSOURCE : in L_EXEMPLE_DESC_CONDITION)
                                        return LES_MODIFICATEURS;

   -- Recuperer le rang du modificateur si c'est un noeud-ressource

   -- ATTENTION : Si le type du noeud ne correspond pas, on leve
   --             L_EXEMPLE_PANIQUE_DESCRIPTION_CONDITION

   function L_EXEMPLE_DONNE_RANG_MODIFICATEUR
                    (NOEUD_RESSOURCE : in L_EXEMPLE_DESC_CONDITION)
                                        return POSITIVE;

   -- Recuperer la valeur immediate associee a un noeud-valeur

   -- ATTENTION : Si le type du noeud ne correspond pas, on leve
   --             L_EXEMPLE_PANIQUE_DESCRIPTION_CONDITION

   function L_EXEMPLE_DONNE_VALEUR_NOEUD_VALEUR
```

```
                    (NOEUD_VALEUR : in L_EXEMPLE_DESC_CONDITION)
                                        return MARQUAGE_L_EXEMPLE;

   -- Recuperer le type du noeud d'un arbre de description d'une condition

   -- ATTENTION : Si le type du noeud ne correspond pas, on leve
   --             L_EXEMPLE_PANIQUE_DESCRIPTION_CONDITION

   function L_EXEMPLE_DONNE_TP_NOEUD_ARBRE_DSC_COND
                    (NOEUD : in L_EXEMPLE_DESC_CONDITION)
                                        return L_EXEMPLE_TYP_ND_ARBRE_DESC_COND;

   -- Tester si la condition implique une interface ou non

   function L_EXEMPLE_CONDITION_AVEC_INTERFACE (COND : in L_EXEMPLE_CONDITION)
                                                return BOOLEAN;

   -- Recuperer le nom de l'interface impliquee dans une condition

   function L_EXEMPLE_DONNE_NOM_INTERF_CONCERNEE (COND : in L_EXEMPLE_CONDITION)
                                                  return L_EXEMPLE_INTERFACES;

   -- Recuperer la marque a produire ou le masque a consommer dans l'interface

   function L_EXEMPLE_DONNE_MARQUE_INTERF_CONDITION (COND : in L_EXEMPLE_CONDITION)
                                                     return MARQUAGE_L_EXEMPLE;

   -- Desallocation d'un arbre de description d'une condition

   procedure L_EXEMPLE_DESALLOUE_ARBRE_DSC_COND
                    (ARBRE_A_DESALLOUER : in out L_EXEMPLE_DESC_CONDITION);

   -- Vider une condition (desallouer l'arbre et les renseignements sur la
   -- Ressource externe)

   procedure VIDE_CONDITION_L_EXEMPLE
                    (CONDITION_A_VIDER : in out L_EXEMPLE_CONDITION);

end L_EXEMPLE_DESCRIPTION_CONDITION;
```

Code 10 :spécification de l'unité de gestion du résultat d'une consommation

```
with L_EXEMPLE_DESCRIPTION_RESSOURCE,
     L_EXEMPLE_DESCRIPTION_MARQUAGE;

use  L_EXEMPLE_DESCRIPTION_RESSOURCE,
     L_EXEMPLE_DESCRIPTION_MARQUAGE;

package L_EXEMPLE_DESCRIPTION_RESULTAT_CONSOMMATION is

   -- Definition de l'exception L_EXEMPLE_PANIQUE_RESULTAT_CONSOMMATION

   L_EXEMPLE_PANIQUE_RESULTAT_CONSOMMATION : exception;

   -- Description d'un ensemble de marques consommables sur l'ensemble
   -- des Ressources du modele

   type UN_ENSEMBLE_DE_MARQUAGE is array
                    (L_EXEMPLE_RESSOURCES) of MARQUAGE_L_EXEMPLE;

   type L_EXEMPLE_UN_MARQUAGE;

   type L_EXEMPLE_ENSEMBLE_DE_MARQUES is access L_EXEMPLE_UN_MARQUAGE;

   type L_EXEMPLE_UN_MARQUAGE is
      record
         LES_MARQUES : UN_ENSEMBLE_DE_MARQUAGE
                       := (others => MARQUAGE_NULL_L_EXEMPLE);
         E_M_SUIVANT : L_EXEMPLE_ENSEMBLE_DE_MARQUES;
      end record;

   -- Constante correspondant a un “ensemble de marques vide”

   L_EXEMPLE_PAS_DE_MARQUES : constant L_EXEMPLE_ENSEMBLE_DE_MARQUES := null;

   -- Description du resultat d'une consommation

   type L_EXEMPLE_RESULTAT_CONSOMMATION is
      record
         RESULTAT_INTERFACE : MARQUAGE_L_EXEMPLE := MARQUAGE_NULL_L_EXEMPLE;
         RESULTAT_RESSOURCES : L_EXEMPLE_ENSEMBLE_DE_MARQUES
                               := L_EXEMPLE_PAS_DE_MARQUES;
      end record;

   -- Ensemble correspondant a un “resultat de consommation vide”

   L_EXEMPLE_ECHEC_CONSOMMATION : constant L_EXEMPLE_RESULTAT_CONSOMMATION
                        := (MARQUAGE_NULL_L_EXEMPLE, L_EXEMPLE_PAS_DE_MARQUES);
   -- Desallocation du resultat d'une consommation

   procedure L_EXEMPLE_DESALLOUE_RES_CONSO
                   (RESULTAT_A_DETRUIRE : in out L_EXEMPLE_RESULTAT_CONSOMMATION);

   -- Affichage du resultat d'une consommation (mode DEBUG)

   procedure L_EXEMPLE_AFFICHE_RES_CONSO
                   (RESULTAT_A_AFFICHER : in L_EXEMPLE_RESULTAT_CONSOMMATION);

   -- Affichage d'un ensemble de marques consommables (mode DEBUG)

   procedure L_EXEMPLE_AFFICHE_ENS_MARQUES
                   (ENS_MARQUES_A_AFFICHER : in L_EXEMPLE_ENSEMBLE_DE_MARQUES);

   -- Creer un ensemble de marques consommables

   function L_EXEMPLE_CREE_ENS_MARQUES return L_EXEMPLE_ENSEMBLE_DE_MARQUES;

   -- Desallouer un ensemble de marques consommables

   procedure L_EXEMPLE_DESALLOUE_ENS_MARQUES
                   (ENS_MARQUES_A_DETRUIRE : in out L_EXEMPLE_ENSEMBLE_DE_MARQUES);
```

```
   -- Ajout d'une marque dans un ensemble de marques consommables

   function L_EXEMPLE_AJOUTE_MARQUE_DANS_ENS
                  (ENS_MARQUES : in L_EXEMPLE_ENSEMBLE_DE_MARQUES;
                   RESSOURCE : in L_EXEMPLE_RESSOURCES;
                   MARQUE : in MARQUAGE_L_EXEMPLE)
                                    return L_EXEMPLE_ENSEMBLE_DE_MARQUES;

   -- Recuperer une marque dans un ensemble de marques consommables

   function L_EXEMPLE_DONNE_MARQUE_DANS_ENS
                  (ENS_MARQUES : in L_EXEMPLE_ENSEMBLE_DE_MARQUES;
                   RESSOURCE : in L_EXEMPLE_RESSOURCES) return MARQUAGE_L_EXEMPLE;

   -- Modifier une marque dans un ensemble de marques deja place dans un resultat
   -- de consommation

   function L_EXEMPLE_MODIFIE_MARQUE_DANS_RES_CONSO
                  (RES_CONSO : in L_EXEMPLE_RESULTAT_CONSOMMATION;
                   RESSOURCE : in L_EXEMPLE_RESSOURCES;
                   MARQUE : in MARQUAGE_L_EXEMPLE;
                   POS_ENS_MRQ : in POSITIVE := 1)
                                    return L_EXEMPLE_RESULTAT_CONSOMMATION;

   -- Ajouter un ensemble de marques consommables dans un resultat de consommation

   function L_EXEMPLE_AJOUTE_ENS_MARQUES
                  (RES_CONSO : in L_EXEMPLE_RESULTAT_CONSOMMATION;
                   ENS_MARQUES : in L_EXEMPLE_ENSEMBLE_DE_MARQUES)
                                    return L_EXEMPLE_RESULTAT_CONSOMMATION;

   -- Détacher un ensemble de marques consommables dans un resultat de
   -- consommation

   -- ATTENTION : cette primitive ne desalloue pas l'espace memoire occupe par
   --             l'ensemble de marques

   function L_EXEMPLE_DETACHE_ENS_MARQUES
                  (RES_CONSO : in L_EXEMPLE_RESULTAT_CONSOMMATION;
                   POS_ENS_MRQ : in POSITIVE := 1)
                                    return L_EXEMPLE_RESULTAT_CONSOMMATION;

   -- Recuperer un ensemble de marques consommables dans un resultat de
   -- consommation

   function L_EXEMPLE_DONNE_ENS_MARQUES
                  (RES_CONSO : in L_EXEMPLE_RESULTAT_CONSOMMATION;
                   POS_ENS_MRQ : in POSITIVE := 1)
                                    return L_EXEMPLE_ENSEMBLE_DE_MARQUES;

   -- Dupliquer un ensemble de marques

   function L_EXEMPLE_DUPLIQUE_ENS_MARQUES
                  (ENS_MRQ_A_DUPLIQUER : in L_EXEMPLE_ENSEMBLE_DE_MARQUES)
                                    return L_EXEMPLE_ENSEMBLE_DE_MARQUES;

   -- Recuperer le nombre d'ensembles de marques contenu dans un resultat de
   -- consommation

   function L_EXEMPLE_DONNE_NB_ENS_MARQUES
                  (RES_CONSO : in L_EXEMPLE_RESULTAT_CONSOMMATION) return NATURAL;

   -- Passer a l'ensemble de marques suivant dans une liste d'ensembles de marques

   function L_EXEMPLE_NEXT_ENS_MARQUES
                  (ENS_MARQUES : in L_EXEMPLE_ENSEMBLE_DE_MARQUES)
                                    return L_EXEMPLE_ENSEMBLE_DE_MARQUES;

   -- Affecter le resultat associe a la consommation dans la Ressource externe
   -- liee a la precondition
```

```
   function L_EXEMPLE_AJOUTE_RES_INTERFACE
                  (RES_CONSO : in L_EXEMPLE_RESULTAT_CONSOMMATION;
                   MARQUE : in MARQUAGE_L_EXEMPLE)
                                    return L_EXEMPLE_RESULTAT_CONSOMMATION;

   -- Recuperer le resultat de la consommation dans la Ressource externe liee a la
   -- precondition

   function L_EXEMPLE_DONNE_RES_INTERFACE
                  (RES_CONSO : in L_EXEMPLE_RESULTAT_CONSOMMATION)
                                    return MARQUAGE_L_EXEMPLE;

end L_EXEMPLE_DESCRIPTION_RESULTAT_CONSOMMATION;
```

Code 11 : description des ressources du modèle

```
with L_EXEMPLE_DESCRIPTION_MARQUAGE;

use  L_EXEMPLE_DESCRIPTION_MARQUAGE;

package L_EXEMPLE_DESCRIPTION_RESSOURCE is

   -- Type enumeratif permettant d'identifier les Ressources

   type L_EXEMPLE_LISTE_DES_RESSOURCES is (
      LITT_ID_RESS_L_EXEMPLE_TOP_HORL,
      LITT_ID_RESS_L_EXEMPLE_BASE_P2,
      LITT_ID_RESS_L_EXEMPLE_BASE_P1);

   -- declaration du sous-type decrivant les Ressources externes

   subtype L_EXEMPLE_INTERFACES is L_EXEMPLE_LISTE_DES_RESSOURCES

LITT_ID_RESS_L_EXEMPLE_TOP_HORL;

   -- declaration du sous-type decrivant les Ressources internes

   subtype L_EXEMPLE_RESSOURCES is L_EXEMPLE_LISTE_DES_RESSOURCES

LITT_ID_RESS_L_EXEMPLE_BASE_P1;

   -- Definition du type decrivant une Ressource

   type DESC_RESSOURCE_L_EXEMPLE is
      record
         L_EXEMPLE_DSC_RESS_DOM_COUL : LISTE_DE_DOMAINES_L_EXEMPLE;
         L_EXEMPLE_DSC_RESS_MARQUAGE : MARQUAGE_L_EXEMPLE;
      end record;

   -- Variable decrivant l'ensemble des ressources du modele
   -- ATTENTION : le marquage initial des ressource n'est pas indique ici.
   --             Pour les Ressources externes, seul le domaine de couleur
   --             constitue une information interessante pour les controles.

     RESSOURCES_DU_MODELE_L_EXEMPLE  :  array  (L_EXEMPLE_LISTE_DES_RESSOURCES)  of
DESC_RESSOURCE_L_EXEMPLE := (
      LITT_ID_RESS_L_EXEMPLE_BASE_P1 =>
         (L_EXEMPLE_DSC_RESS_DOM_COUL => IDENT_L_EXEMPLE_DOM,
          L_EXEMPLE_DSC_RESS_MARQUAGE => null),
      LITT_ID_RESS_L_EXEMPLE_BASE_P2 =>
         (L_EXEMPLE_DSC_RESS_DOM_COUL => IDENT_L_EXEMPLE_DOM,
          L_EXEMPLE_DSC_RESS_MARQUAGE => null),
      LITT_ID_RESS_L_EXEMPLE_TOP_HORL =>
         (L_EXEMPLE_DSC_RESS_DOM_COUL => IDENT_L_EXEMPLE_MONOCHROME,
          L_EXEMPLE_DSC_RESS_MARQUAGE => null));

end L_EXEMPLE_DESCRIPTION_RESSOURCE;
```

Code 12 : spécification du gestionnaire des Ressources

```
with L_EXEMPLE_DESCRIPTION_CONDITION,
     L_EXEMPLE_DEFINITIONS_POUR_LES_PROCESSUS,
     L_EXEMPLE_DESCRIPTION_RESSOURCE,
     L_EXEMPLE_DESCRIPTION_RESULTAT_CONSOMMATION;

use  L_EXEMPLE_DESCRIPTION_CONDITION,
     L_EXEMPLE_DEFINITIONS_POUR_LES_PROCESSUS,
     L_EXEMPLE_DESCRIPTION_RESSOURCE,
     L_EXEMPLE_DESCRIPTION_RESULTAT_CONSOMMATION;

package L_EXEMPLE_GESTION_DES_RESSOURCES is

   -- Definition de l'exception L_EXEMPLE_PANIQUE_GESTION_RESSOURCES

   L_EXEMPLE_PANIQUE_GESTION_RESSOURCES : exception;

   -- Evaluation d'une precondition

   procedure L_EXEMPLE_SERVICE_CONSOMMER
                      (IDENTITE_CLIENT : in IDENTITE_L_EXEMPLE;
                       PRECONDITION : in L_EXEMPLE_CONDITION;
                       RESULTAT : out L_EXEMPLE_RESULTAT_CONSOMMATION;
                       ESTAMPILLE_SI_ATTENTE : out ESTAMPILLE_L_EXEMPLE);

   -- Annulation d'une demande d'evaluation d'une precondition

   procedure L_EXEMPLE_SERVICE_ANNULER
                      (IDENTITE_CLIENT : in IDENTITE_L_EXEMPLE;
                       ESTAMPILLE_DEMANDE : in ESTAMPILLE_L_EXEMPLE;
                       ANNULATION_OK : out BOOLEAN);

   -- Production d'une postcondition

   procedure L_EXEMPLE_SERVICE_PRODUIRE (IDENTITE_CLIENT : in IDENTITE_L_EXEMPLE;
                                         POSTCONDITION : in L_EXEMPLE_CONDITION);

   -- Initialisation du gestionnaire des Ressources

   procedure L_EXEMPLE_SERVICE_INIT_RESSOURCES;

   -- Terminer le gestionnaire des Ressources

   procedure L_EXEMPLE_SERVICE_TERMINE_RESSOURCES;

end L_EXEMPLE_GESTION_DES_RESSOURCES;
```

Code 13 spécification de l'unité chargée de l'évaluation des requêtes sur les Ressources

```
with L_EXEMPLE_DESCRIPTION_RESSOURCE,
     L_EXEMPLE_DESCRIPTION_CONDITION,
     L_EXEMPLE_DESCRIPTION_RESULTAT_CONSOMMATION;

use  L_EXEMPLE_DESCRIPTION_RESSOURCE,
     L_EXEMPLE_DESCRIPTION_CONDITION,
     L_EXEMPLE_DESCRIPTION_RESULTAT_CONSOMMATION;

package L_EXEMPLE_UTILITAIRES_RESSOURCE_DE_HAUT_NIVEAU is

   -- Definition de l'exception
   -- L_EXEMPLE_PANIQUE_UTILITAIRES_RESSOURCE_DE_HAUT_NIVEAU

   L_EXEMPLE_PANIQUE_UTILITAIRES_RESSOURCE_DE_HAUT_NIVEAU : exception;

   -- Type decrivant le "resultat" d'une production, c'est a dire les ressources
   -- sur lesquelles un "evenement positif" intervient

   type L_EXEMPLE_RES_PRODUCTION is array (L_EXEMPLE_RESSOURCES) of BOOLEAN;
```

```
   -- Type permettant de classer les types de problemes provoquant l'echec
   -- d'une evaluation

   type L_EXEMPLE_LISTE_DES_PB_CONSO is (
      UNE_RESSOURCE,
      UNE_INTERFACE,
      COMBINAISON_DE_DEPENDANCES);

   -- Type permettant au serveur de Ressources de recuperer des informations
   -- sur l'echec d'une precondition

   type L_EXEMPLE_DIAGNOSTIC is
      record
         CATEGORIE_PBLM : L_EXEMPLE_LISTE_DES_PB_CONSO;
         NOM_RESPONSABLE : L_EXEMPLE_LISTE_DES_RESSOURCES;
      end record;

   -- Recuperation, a partir d'un diagnostic, du type de probleme intervenu pendant
   -- la consommation

   function L_EXEMPLE_DONNE_ORIGINE_PB (DIAGNOSTIC : in L_EXEMPLE_DIAGNOSTIC)
                    return L_EXEMPLE_LISTE_DES_PB_CONSO;

   -- Recuperation de la ressource ou de l'interface responsable du probleme
   -- ATTENTION, pas de sens si le probleme est de type COMBINAISON_DE_DEPENDANCES

   function L_EXEMPLE_DONNE_NOM_R_OU_I (DIAGNOSTIC : in L_EXEMPLE_DIAGNOSTIC)
                    return L_EXEMPLE_LISTE_DES_RESSOURCES;

   -- Recuperer le diagnostic du probleme. Attention, il n'y a memoire que pour
   -- le dernier echec

   function L_EXEMPLE_DIAGNOSTIC_DU_PB return L_EXEMPLE_DIAGNOSTIC;

   -- Transformer un diagnostic en chaine de caracteres (mode TRACE)

   function L_EXEMPLE_DIAGNOSTIC_VERS_CHAINE (DIAGNOSTIC : in L_EXEMPLE_DIAGNOSTIC)
                    return STRING;

   -- Production d'une POST-condition

   function L_EXEMPLE_PRODUIRE_POSTCOND (CONDITION : in L_EXEMPLE_CONDITION)
                    return L_EXEMPLE_RES_PRODUCTION;

   -- Consommation d'une PRE-condition

   function L_EXEMPLE_CONSOMMER_PRECOND (CONDITION : in L_EXEMPLE_CONDITION)
                    return L_EXEMPLE_RESULTAT_CONSOMMATION;

end L_EXEMPLE_UTILITAIRES_RESSOURCE_DE_HAUT_NIVEAU;
```

Code 14 spécification de l'unité décrivant les opérations élémentaires sur les Ressources

```
with L_EXEMPLE_DESCRIPTION_RESSOURCE,
     L_EXEMPLE_DESCRIPTION_MARQUAGE;

use  L_EXEMPLE_DESCRIPTION_RESSOURCE,
     L_EXEMPLE_DESCRIPTION_MARQUAGE;

package L_EXEMPLE_UTILITAIRES_RESSOURCES_DE_BAS_NIVEAU is

   -- Production d'un marquage dans une ressource

   procedure L_EXEMPLE_PRODUIRE (RESS : in L_EXEMPLE_LISTE_DES_RESSOURCES;
                                 MARQ : in MARQUAGE_L_EXEMPLE);

   -- Ramene la liste des marques correspondant a un masque donne
   -- ATTENTION : Penser a desallouer le resultat de cette fonction avec
   --             DETRUIRE_LST_MARQUE_L_EXEMPLE

   function L_EXEMPLE_TROUVE_MARQUES_OK (RESS : in L_EXEMPLE_LISTE_DES_RESSOURCES;
                                         MASQ : in MARQUAGE_L_EXEMPLE;
                                         TYP_CMP : in LES_COMPARAISONS := EGALITE;
                                         NUM_CHP : in NATURAL := 0)
                                                return MARQUAGE_L_EXEMPLE;

   -- Consommation (a coup sur) d'un marquage dans une ressource
   -- La primitive ramene la liste des marques consommees
   -- ATTENTION : Penser a desallouer le resultat de cette fonction via
   --             DETRUIRE_LST_MARQUE_L_EXEMPLE

   function L_EXEMPLE_CONSOMME (RESS : in L_EXEMPLE_LISTE_DES_RESSOURCES;
                                MASQ : in MARQUAGE_L_EXEMPLE)
                                                return MARQUAGE_L_EXEMPLE;

end L_EXEMPLE_UTILITAIRES_RESSOURCES_DE_BAS_NIVEAU;
```

Code 15 : spécification de l'unité permettant la manipulation du mot d'état de P1

```
with L_EXEMPLE_DESCRIPTION_CLASSES,
     L_EXEMPLE_DESCRIPTION_MARQUAGE;

use  L_EXEMPLE_DESCRIPTION_CLASSES,
     L_EXEMPLE_DESCRIPTION_MARQUAGE;

package DESCRIPTION_MOT_D_ETAT_DE_L_EXEMPLE_P1 is

   -- Le type enumeratif permettant d'identifier les champs du mot d'etat

   type CHAMP_L_EXEMPLE_P1 is (
      P1_IDENT_I1,
      P1_IDENT_OU1,
      P1_IDENT_C1);

   -- Le type decrivant le mot d'etat

   type MOT_D_ETAT_L_EXEMPLE_P1 is
      record
         CHAMP_I1 : L_EXEMPLE_IDENTIFICATEUR;
         CHAMP_OU1 : L_EXEMPLE_OU_ALLER;
         CHAMP_C1 : L_EXEMPLE_COMPTEUR;
      end record;

   -- La primitive ramenant la marque contenue dans un champ donne du mot d'etat

   function L_EXEMPLE_DONNE_CHP_M_ETAT_P1 (MOT_D_ETAT : in MOT_D_ETAT_L_EXEMPLE_P1;
                                           CHP_CONCERNE : in CHAMP_L_EXEMPLE_P1)
                                                return MARQUAGE_L_EXEMPLE;

   -- La primitive affectant un champ donne du mot d'etat

   function L_EXEMPLE_ECRIT_CHP_M_ETAT_P1
                     (MOT_INITIAL : in MOT_D_ETAT_L_EXEMPLE_P1;
                      CHP_A_MODIFIER : in CHAMP_L_EXEMPLE_P1;
                      MARQUE : in MARQUAGE_L_EXEMPLE)
                                      return MOT_D_ETAT_L_EXEMPLE_P1;

end DESCRIPTION_MOT_D_ETAT_DE_L_EXEMPLE_P1;
```

Code 16 : spécification du paquetage décrivant le comportement de P1

```
with L_EXEMPLE_DEFINITIONS_POUR_LES_PROCESSUS,
     DESCRIPTION_MOT_D_ETAT_DE_L_EXEMPLE_P1;

use L_EXEMPLE_DEFINITIONS_POUR_LES_PROCESSUS,
     DESCRIPTION_MOT_D_ETAT_DE_L_EXEMPLE_P1;

package DEF_PROCESSUS_L_EXEMPLE_P1 is

   -- Le type enumerant les Etats Actifs de P1

   subtype L_EXEMPLE_ETATS_ACTIFS_DE_P1 is ETAT_ACTIFS_DE_L_EXEMPLE range
                                           P1_E_ACTIF_P1A .. P1_E_ACTIF_TSYNC;

   -- Initialiser une instance du processus

   procedure L_EXEMPLE_INITIALISE_L_INSTANCE_DE_P1
                      (ID_INSTANCE : in POSITIVE;
                       MOT_D_ETAT : in MOT_D_ETAT_L_EXEMPLE_P1;
                       ETAT_ACTIF : in L_EXEMPLE_ETATS_ACTIFS_DE_P1);

   -- Creer les instance du processus

   procedure L_EXEMPLE_CREER_LES_INSTANCES_DE_P1 (NB_D_INSTANCES : in POSITIVE);

   -- Tuer brutalement toutes les instances du processus

   procedure L_EXEMPLE_DETRUIT_LES_INSTANCES_DE_P1;

end DEF_PROCESSUS_L_EXEMPLE_P1;
```

Code 17 : modèle de tâche associé à P1

```
   -- Le type tache decrivant un processus du modele

   task type PROCESSUS_P1 is
      entry INITIALISE (NUMERO_D_INSTANCE : in POSITIVE;
                        ETAT_ACTIF_INITIAL : in L_EXEMPLE_ETATS_ACTIFS_DE_P1;
                        MOT_D_ETAT_INITIAL : in MOT_D_ETAT_L_EXEMPLE_P1);
   end PROCESSUS_P1;

   -- Le corps de la tache decrivant un processus du modele

   task body PROCESSUS_P1 is

   ID_INSTANCE : POSITIVE;
   ETAT_ACTIF_COURANT : L_EXEMPLE_ETATS_ACTIFS_DE_P1;
   VIVRE : BOOLEAN := TRUE;
   MOT_D_ETAT : MOT_D_ETAT_L_EXEMPLE_P1;

   begin
      accept INITIALISE (NUMERO_D_INSTANCE : in POSITIVE;
                         ETAT_ACTIF_INITIAL : in L_EXEMPLE_ETATS_ACTIFS_DE_P1;
                         MOT_D_ETAT_INITIAL : in MOT_D_ETAT_L_EXEMPLE_P1) do
         ID_INSTANCE := NUMERO_D_INSTANCE;
         ETAT_ACTIF_COURANT := ETAT_ACTIF_INITIAL;
         MOT_D_ETAT := MOT_D_ETAT_INITIAL;
      end INITIALISE;

      while VIVRE loop
         case ETAT_ACTIF_COURANT is
            -- L'etat processus P1a

            when P1_E_ACTIF_P1A =>
               ETAT_ACTIF_COURANT := P1_EFFECTUER_CHOIX_P1A (ETAT_ACTIF_COURANT,
                                                            TOURNIQUET);

            -- L'etat processus P1b

            -- L'etat processus P1c

            when P1_E_ACTIF_P1C =>
               ETAT_ACTIF_COURANT := P1_EFFECTUER_CHOIX_P1C (ETAT_ACTIF_COURANT,
                                                            TOURNIQUET);

            -- L'etat processus P1d

            -- L'etat processus P1fin

            -- L'action Tloc1

            when P1_E_ACTIF_TLOC1 =>
               P1_TENTER_TLOC1 (MOT_D_ETAT,
                                VIVRE,
                                ETAT_ACTIF_COURANT,
                                ID_INSTANCE);

            -- L'action Tincr1

            when P1_E_ACTIF_TINCR1 =>
               P1_TENTER_TINCR1 (MOT_D_ETAT,
                                 VIVRE,
                                 ETAT_ACTIF_COURANT,
                                 ID_INSTANCE);

            -- L'action Tfin1

            when P1_E_ACTIF_TFIN1 =>
               P1_TENTER_TFIN1 (MOT_D_ETAT,
                                VIVRE,
                                ETAT_ACTIF_COURANT,
                                ID_INSTANCE);

            -- L'action Tcons1
```

```
            when P1_E_ACTIF_TCONS1 =>
               P1_TENTER_TCONS1 (MOT_D_ETAT,
                                 VIVRE,
                                 ETAT_ACTIF_COURANT,
                                 ID_INSTANCE);

            -- L'action Tprod1

            when P1_E_ACTIF_TPROD1 =>
               P1_TENTER_TPROD1 (MOT_D_ETAT,
                                 VIVRE,
                                 ETAT_ACTIF_COURANT,
                                 ID_INSTANCE);

            -- L'action Tsync

            when P1_E_ACTIF_TSYNC =>
               P1_TENTER_TSYNC (MOT_D_ETAT,
                                VIVRE,
                                ETAT_ACTIF_COURANT,
                                ID_INSTANCE);

         end case;

         -- Provoquer une election au cas ou l'executable n'est pas preemptif!!!

         L_EXEMPLE_PROVOQUE_ELECTION;
      end loop;

      TRACE_AVEC_RC ("Terminaison de P1." & INTEGER'IMAGE (ID_INSTANCE));
      L_EXEMPLE_JE_ME_TERMINE (L_EXEMPLE_ID_PROC_P1, FALSE);

   exception
      when CONSTRAINT_ERROR =>
         TRACE_AVEC_RC ("CONSTRAINT_ERROR -> PROCESSUS_P1");
         L_EXEMPLE_JE_ME_TERMINE;
      when TASKING_ERROR =>
         TRACE_AVEC_RC ("TASKING_ERROR -> PROCESSUS_P1");
         L_EXEMPLE_JE_ME_TERMINE;
      when PROGRAM_ERROR =>
         TRACE_AVEC_RC ("PROGRAM_ERROR -> PROCESSUS_P1");
         L_EXEMPLE_JE_ME_TERMINE;
      when others =>
         TRACE_AVEC_RC ("Exception -> PROCESSUS_P1");
         L_EXEMPLE_JE_ME_TERMINE;
   end PROCESSUS_P1;
```

Code 18 spécification de l'unité décrivant les Etats_Processus alternatifs de P1

```
with DEF_PROCESSUS_L_EXEMPLE_P1,
     L_EXEMPLE_DEFINITIONS_POUR_LES_PROCESSUS;

use  DEF_PROCESSUS_L_EXEMPLE_P1,
     L_EXEMPLE_DEFINITIONS_POUR_LES_PROCESSUS;

package L_EXEMPLE_ALTERNATIVES_P1 is

   -- Effectuer le choix associe a l'alternative P1a

   function P1_EFFECTUER_CHOIX_P1A (VALEUR_IN : in L_EXEMPLE_ETATS_ACTIFS_DE_P1;
                                    POLITIQUE : in POSITIVE)
                                              return L_EXEMPLE_ETATS_ACTIFS_DE_P1;

   -- Effectuer le choix associe a l'alternative P1c

   function P1_EFFECTUER_CHOIX_P1C (VALEUR_IN : in L_EXEMPLE_ETATS_ACTIFS_DE_P1;
                                    POLITIQUE : in POSITIVE)
                                              return L_EXEMPLE_ETATS_ACTIFS_DE_P1;

end L_EXEMPLE_ALTERNATIVES_P1;
```

Code 19 : spécification de l'unité décrivant les Actions de P1

```
with DEF_PROCESSUS_L_EXEMPLE_P1,
     DESCRIPTION_MOT_D_ETAT_DE_L_EXEMPLE_P1,
     L_EXEMPLE_DEFINITIONS_POUR_LES_PROCESSUS;

use  DEF_PROCESSUS_L_EXEMPLE_P1,
     DESCRIPTION_MOT_D_ETAT_DE_L_EXEMPLE_P1,
     L_EXEMPLE_DEFINITIONS_POUR_LES_PROCESSUS;

package L_EXEMPLE_ACTIONS_DE_P1 is

   -- Tenter l'Action Tloc1

   procedure P1_TENTER_TLOC1 (MOT_D_ETAT : in out MOT_D_ETAT_L_EXEMPLE_P1;
                              CONTINUER : in out BOOLEAN;
                              ETAT_COURANT : in out L_EXEMPLE_ETATS_ACTIFS_DE_P1;
                              NUM_INSTANCE : in POSITIVE);

   -- Tenter l'Action Tincr1

   procedure P1_TENTER_TINCR1 (MOT_D_ETAT : in out MOT_D_ETAT_L_EXEMPLE_P1;
                               CONTINUER : in out BOOLEAN;
                               ETAT_COURANT : in out L_EXEMPLE_ETATS_ACTIFS_DE_P1;
                               NUM_INSTANCE : in POSITIVE);

   -- Tenter l'Action Tfin1

   procedure P1_TENTER_TFIN1 (MOT_D_ETAT : in out MOT_D_ETAT_L_EXEMPLE_P1;
                              CONTINUER : in out BOOLEAN;
                              ETAT_COURANT : in out L_EXEMPLE_ETATS_ACTIFS_DE_P1;
                              NUM_INSTANCE : in POSITIVE);

   -- Tenter l'Action Tcons1

   procedure P1_TENTER_TCONS1 (MOT_D_ETAT : in out MOT_D_ETAT_L_EXEMPLE_P1;
                               CONTINUER : in out BOOLEAN;
                               ETAT_COURANT : in out L_EXEMPLE_ETATS_ACTIFS_DE_P1;
                               NUM_INSTANCE : in POSITIVE);

   -- Tenter l'Action Tprod1

   procedure P1_TENTER_TPROD1 (MOT_D_ETAT : in out MOT_D_ETAT_L_EXEMPLE_P1;
                               CONTINUER : in out BOOLEAN;
                               ETAT_COURANT : in out L_EXEMPLE_ETATS_ACTIFS_DE_P1;
                               NUM_INSTANCE : in POSITIVE);

   -- Tenter l'Action Tsync

   procedure P1_TENTER_TSYNC (MOT_D_ETAT : in out MOT_D_ETAT_L_EXEMPLE_P1;
                              CONTINUER : in out BOOLEAN;
                              ETAT_COURANT : in out L_EXEMPLE_ETATS_ACTIFS_DE_P1;
                              NUM_INSTANCE : in POSITIVE);

end L_EXEMPLE_ACTIONS_DE_P1;
```

Code 20 : code associé à l'Etat_Processus P1a

```
   function P1_EFFECTUER_CHOIX_P1A (VALEUR_IN : in L_EXEMPLE_ETATS_ACTIFS_DE_P1;
                                    POLITIQUE : in POSITIVE)
                                         return L_EXEMPLE_ETATS_ACTIFS_DE_P1 is

   VAL_RET : L_EXEMPLE_ETATS_ACTIFS_DE_P1 := VALEUR_IN;
   NOMBRE_DE_SUCCESSEURS : constant POSITIVE := 2;

   -- La liste des choix possibles depuis l'alternative

        ACTIONS_POSSIBLES  :  constant  array  (1  ..  NOMBRE_DE_SUCCESSEURS)  of
L_EXEMPLE_ETATS_ACTIFS_DE_P1 := (
      1 => P1_E_ACTIF_TLOC1,
      2 => P1_E_ACTIF_TSYNC);

   begin

      -- Trace puisque DEBUG a ete selectionne en mode DEVELOPPEUR
      if TRACE then
         TRACE_AVEC_RC ("--> P1_EFFECTUER_CHOIX_P1A (" &
                        L_EXEMPLE_ETATS_ACTIFS_DE_P1'IMAGE (VALEUR_IN) & "," &
                        INTEGER'IMAGE (POLITIQUE) & ")");
      end if;

      -- Apres la trace, les instructions

      DERNIERE_VALEUR_P1A := EFFECTUE_CHOIX (DERNIERE_VALEUR_P1A,
                                             NOMBRE_DE_SUCCESSEURS, POLITIQUE);
      VAL_RET := ACTIONS_POSSIBLES (DERNIERE_VALEUR_P1A);

      -- Trace puisque DEBUG a ete selectionne en mode DEVELOPPEUR
      if TRACE then
         TRACE_AVEC_RC ("<-- P1_EFFECTUER_CHOIX_P1A avec " &
                        L_EXEMPLE_ETATS_ACTIFS_DE_P1'IMAGE (VAL_RET) & ")");
      end if;

      return VAL_RET;
   end P1_EFFECTUER_CHOIX_P1A;
```

Code 21 : code associé à la transition Tcons1

```
procedure P1_TENTER_TCONS1 (MOT_D_ETAT : in out MOT_D_ETAT_L_EXEMPLE_P1;
                            CONTINUER : in out BOOLEAN;
                            ETAT_COURANT : in out L_EXEMPLE_ETATS_ACTIFS_DE_P1;
                            NUM_INSTANCE : in POSITIVE) is

PRE_OU_POST_CONDITION : L_EXEMPLE_CONDITION;
RESULTAT_PRECONDITION : L_EXEMPLE_RESULTAT_CONSOMMATION;
ESTAMPILLE_SI_ATTENTE : ESTAMPILLE_L_EXEMPLE;
ANNULATION_OK : BOOLEAN;

begin

   -- Trace puisque DEBUG a ete selectionne en mode DEVELOPPEUR
   if TRACE then
      TRACE_AVEC_RC ("--> P1_TENTER_TCONS1 (MOT_D_ETAT, CONTINUER, " &
                     "ETAT_COURANT," & INTEGER'IMAGE (NUM_INSTANCE) & ")");
   end if;

   -- Apres la trace, les instructions

   -- Construire la "precondition ressource"

   PRE_OU_POST_CONDITION := P1_PRE_CONDITION_TCONS1 (MOT_D_ETAT);

   -- Evaluer la "precondition ressource"

   L_EXEMPLE_SERVICE_CONSOMMER (L_EXEMPLE_CREER_IDENTITE (L_EXEMPLE_ID_PROC_P1,
                                                          NUM_INSTANCE),
                                PRE_OU_POST_CONDITION,
                                RESULTAT_PRECONDITION,
                                ESTAMPILLE_SI_ATTENTE);

   -- Si elle n'est pas immediatement realisable, se mettre en attente

   if RESULTAT_PRECONDITION = L_EXEMPLE_ECHEC_CONSOMMATION then
      L_EXEMPLE_J_ATTEND_PRECONDITION (L_EXEMPLE_CREER_IDENTITE
                                          (L_EXEMPLE_ID_PROC_P1, NUM_INSTANCE),
                                       RESULTAT_PRECONDITION,
                                       MOMENTANEMENT);

      if RESULTAT_PRECONDITION = L_EXEMPLE_ECHEC_CONSOMMATION then
         L_EXEMPLE_SERVICE_ANNULER (L_EXEMPLE_CREER_IDENTITE
                                       (L_EXEMPLE_ID_PROC_P1, NUM_INSTANCE),
                                    ESTAMPILLE_SI_ATTENTE,
                                    ANNULATION_OK);
         if not ANNULATION_OK then

            -- Croisement des messages, attendre

            L_EXEMPLE_J_ATTEND_PRECONDITION (L_EXEMPLE_CREER_IDENTITE
                                                (L_EXEMPLE_ID_PROC_P1,
                                                 NUM_INSTANCE),
                                             RESULTAT_PRECONDITION,
                                             DEFINITIVEMENT);
         end if;
      end if;
   end if;
   VIDE_CONDITION_L_EXEMPLE (PRE_OU_POST_CONDITION);
   if RESULTAT_PRECONDITION /= L_EXEMPLE_ECHEC_CONSOMMATION then

      -- Modifier le mot d'etat en fonction de ce que l'on a consomme

      MOT_D_ETAT := L_EXEMPLE_ECRIT_CHP_M_ETAT_P1 (MOT_D_ETAT, P1_IDENT_C1,
                    L_EXEMPLE_GET_CHAMP (L_EXEMPLE_DONNE_MARQUE_DANS_ENS
                                            (L_EXEMPLE_DONNE_ENS_MARQUES
                                                 (RESULTAT_PRECONDITION),
                                             LITT_ID_RESS_L_EXEMPLE_BASE_P1),
                                         2));

      -- Appel de la procedure associee a la transition Tcons1
```

```
         PROC_P1_TCONS1 (MOT_D_ETAT,
                         L_EXEMPLE_CREER_IDENTITE (L_EXEMPLE_ID_PROC_P1,
                                                   NUM_INSTANCE));
         L_EXEMPLE_DESALLOUE_RES_CONSO (RESULTAT_PRECONDITION);

         -- Passer a l'Etat Actif suivant

         ETAT_COURANT := P1_E_ACTIF_TPROD1;
      else
         ETAT_COURANT := P1_E_ACTIF_P1C;
      end if;

      -- Trace puisque DEBUG a ete selectionne en mode DEVELOPPEUR
      if TRACE then
         TRACE_AVEC_RC ("<-- P1_TENTER_TCONS1");
      end if;


   -- Generer la pile des appels de procedure en cas d'erreur
   exception
      when others =>
         POSITIONNER_DIAGNOSTIC ("P1_TENTER_TCONS1", "");
         raise;
   end P1_TENTER_TCONS1;
```

Code 22 : spécification du paquetage réalisant l'Action synchronisée Tsync

```
with DESCRIPTION_MOT_D_ETAT_DE_L_EXEMPLE_P1,
    DESCRIPTION_MOT_D_ETAT_DE_L_EXEMPLE_P2,
     L_EXEMPLE_DEFINITIONS_POUR_LES_PROCESSUS;

use  DESCRIPTION_MOT_D_ETAT_DE_L_EXEMPLE_P1,
    DESCRIPTION_MOT_D_ETAT_DE_L_EXEMPLE_P2,
     L_EXEMPLE_DEFINITIONS_POUR_LES_PROCESSUS;

package GESTION_DE_L_EXEMPLE_TSYNC is

   -- Definition de l'exception L_EXEMPLE_PANIQUE_TSYNC

   L_EXEMPLE_PANIQUE_TSYNC : exception;

   -- Initialisation des mecanismes de gestion de l'action synchronisee

   procedure L_EXEMPLE_SERVICE_INIT_TSYNC;

   -- Terminaison des mecanismes de gestion de l'action synchronisee

   procedure L_EXEMPLE_SERVICE_TERMINE_TSYNC;

   -- Primitive par laquelle un client se retire

   procedure L_EXEMPLE_SERVICE_J_ABANDONNE_TSYNC
                               (IDENTITE_DU_PARTICIPANT : in IDENTITE_L_EXEMPLE;
                                ACCEPTE : out BOOLEAN);

   -- Primitive de connexion a la synchronisation pour P1

   procedure L_EXEMPLE_SERVICE_CONNEXION_TSYNC_P1
                               (MOT_D_ETAT : in MOT_D_ETAT_L_EXEMPLE_P1;
                                NUM_INSTANCE : in POSITIVE);

   -- Primitive de connexion a la synchronisation pour P2

   procedure L_EXEMPLE_SERVICE_CONNEXION_TSYNC_P2
                               (MOT_D_ETAT : in MOT_D_ETAT_L_EXEMPLE_P2;
                                NUM_INSTANCE : in POSITIVE);
```

```
end GESTION_DE_L_EXEMPLE_TSYNC;
```

Code 23 : spécification du module de contrôle

```
with L_EXEMPLE_DEFINITIONS_POUR_LES_PROCESSUS;

use  L_EXEMPLE_DEFINITIONS_POUR_LES_PROCESSUS;

package GESTION_DU_CONTROLE_DE_L_EXEMPLE is

   -- Primitive permettant a une tache de signaler sa terminaison

   -- ATENTION : Deux types de taches l'utilisent :
   --            -> Les taches de service qui ne signalent que des terminaisons
   --               anormales. Elles ne donnent pas leur identite et ne specifient
   --               pas non plus le type de leur terminaison (utilisation des
   --               valeurs par defaut des parametres)
   --            -> Les instances des taches issues de la decomposition du modele.
   --               Si elles se terminent normalement, elle donnent leur identite,
   --               dans le cas contraire, l'appel de la primitive sera similaire
   --               a celui effectue par les taches de service.

 procedure L_EXEMPLE_JE_ME_TERMINE (TYPE_DE_PROCESSUS : in PROCESSUS_DE_L_EXEMPLE
:= PROCESSUS_DE_L_EXEMPLE'FIRST;
                                    SUR_ERREUR : in BOOLEAN := TRUE);

   -- Initialisation du gardien qui lancera les taches de l'application

   procedure LANCER_L_EXEMPLE;

end GESTION_DU_CONTROLE_DE_L_EXEMPLE;
```

Code 24 : le programme principal

```
with GESTION_DU_CONTROLE_DE_L_EXEMPLE,
     UTILITAIRES_DE_TRACE,
     MECANISME_DE_DIAGNOSTIC;

use  GESTION_DU_CONTROLE_DE_L_EXEMPLE,
     UTILITAIRES_DE_TRACE,
     MECANISME_DE_DIAGNOSTIC;

procedure L_EXEMPLE is

begin
   INITIALISE_TRACES;
   LANCER_L_EXEMPLE;
end L_EXEMPLE;
```

		Introduction générale
Chapitre I	:	*Prototypage de systèmes parallèles*
Chapitre II	:	*Interprétation sémantique d'un réseau de Petri*
Chapitre III	:	*Méthodologie de prototypage*
Chapitre IV	:	*Structure du prototype généré*
Chapitre V	:	*Caractéristiques du prototype Ada*
Chapitre VI	:	*Le prototype centralisé*
Chapitre VII	:	*Le prototype réparti*
Chapitre VIII	:	*CPN/TAGADA : L'outil de prototypage*
		Conclusion générale
		Bibliographie
+		*Annexes & Appendices*

Annexe C :

La description LANBADA

1. Introduction....397
2. Définitions....397
3. Description de l'ensemble du réseau de Petri....397
4. Description d'une classe de couleurs....398
5. Description d'un domaine de couleurs....399
6. Description des fonctions non prédéfinies du modèle....400
7. Description des ressources du modèle....401
8. Description d'un processus du modèle....403

1. Introduction

Le langage LANBADA permet de décrire un réseau de Petri en termes d'ensembles de G-objets. Cette description fournit des informations permettant la génération d'un prototype dans n'importe quel langage cible. Il s'agit d'un lien entre l'application qui réalise les phases d'Identification et d'Analyse et l'application qui effectue les phases de Placement et de Génération.

LANBADA a donc été conçu de façon à être le plus large possible. Il ne tient pas compte des hypothèses appliquées aux réalisations actuelles.

A la description du modèle analysé s'ajoutent un certain nombre d'informations que nous transmettons à la phase de génération :

- le nom de la procédure associée à chaque Action;
- le nom du modèle.

Les informations concernant les composants externes sont directement puisées dans la bibliothèque, gérée par l'outil de prototypage. Les informations que nous transmettons sont déduites du formalisme utilisé pour définir les réseaux de Petri colorés dans l'atelier AMI : les AMI-nets.

2. DÉFINITIONS

Dans les définitions grammaticales que nous donnons, les mots clefs du langage seront indiqués en petites capitales gras **COURRIER**.

Toute ligne commençant par la séquence "--" est considérée comme un commentaire et ignorée.

Les nombres sont définis comme suit :

nombre ::= car_num <nombre> | car_num

Les identificateurs sont définis comme suit :

identificateur ::= car_alpha <identificateur2>
identificateur2 ::= car <identificateur2> | car
car ::= car_alpha | car_num
car_alpha ::= a | b | c | d | e | f | g | h | i | j | k | l | m | n | o | p | q | r | s | t | u | v |
w | x | y | z | _ | A | B | C | D | E | F | G | H | I | J | K | L | M | N | O | P | Q |
R | S | T | U | V | W | X | Y | Z
car_num ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

Nous utilisons les majuscules et minuscules afin de rendre la description plus claire; cependant, aucune différence n'est considérée.

3. DESCRIPTION DE L'ENSEMBLE DU RÉSEAU DE PETRI

La description d'un réseau de Petri suit les règles de syntaxe données ci-dessous :

description_reseau ::=
RESEAU identificateur
{description_classes}
{description_domaines}
{description_fonctions}
{description_ressources}
liste_des_processus
FIN_RESEAU

Le modèle doit être nommé (la description fournit un identificateur) et comporter au moins un processus.

Si le modèle n'est relié à aucun composant externe, la description de leurs interfaces est inutile. Si le modèle ne comporte pas de Ressources, la description correspondante n'existe pas. De même, si le réseau de Petri est ordinaire, aucun domaine de couleurs n'est décrit.

4. DESCRIPTION D'UNE CLASSE DE COULEURS

La description d'une classe de couleurs suit les règles de syntaxe données ci-dessous :

description_classes ::=
description_classe
{description_classe}
description_classe ::=
CLASSE identificateur
enumeration_de_valeur | classe_predefinie
FIN_CLASSE
enumeration_de_valeur ::=
ligne_d_identificateurs
{ligne_d_identificateurs}
ligne_d_identificateurs ::=
identificateur {identificateur}
classe_predefinie ::=
ENTIERS | **CARACTERES** | definition_d_intervalle
definition_d_intervalle ::=
INTERVALLE type_intervalle borne_inferieure borne_superieure
type_intervalle ::=
ENTIERS | **CARACTERES**
borne_inferieure ::=
[-]nombre | car_alpha
borne_superieure ::=
[-]nombre | car_alpha

Les seules valeurs énumérées interdites dans LANBADA sont *entiers* et *caracteres*.

Exemple C.1 : Nous donnons, dans Lanbada C.1 des exemples de définition de classes d'un modèle :

- la classe *toto* correspond à une liste de valeurs : rouge, vert, bleu, jaune, orange et rose. Le successeur de rose sera rouge;
- la classe *nombre* correspond à l'ensemble des entiers représentables dans le langage cible considéré;
- la classe *intervalle_1_50* correspond aux entiers compris entre 1 et 50. Le successeur de 50 sera 1.

```
CLASSE toto
   rouge vert
   bleu jaune orange
   rose
FIN_CLASSE

CLASSE nombres
   ENTIERS
FIN_CLASSE

CLASSE intervalle_1_50
   INTERVALLE 1 50
FIN_CLASSE
```

Lanbada C.1 : Exemples de description de classes.

5. DESCRIPTION D'UN DOMAINE DE COULEURS

La description d'un domaine de couleurs suit les règles de syntaxe données ci-dessous :

description_domaines ::=
    description_domaine
    {description_domaine}
description_domaine ::=
    DOMAINE identificateur
        declaration_produit | declaration_union
    FIN_DOMAINE
declaration_produit ::=
    PRODUIT
        liste_de_composantes
    FIN_PRODUIT
declaration_union ::=
    UNION
        liste_de_composantes
    FIN_UNION
liste_de_composantes ::=
    composante {composante}
composante ::=
    identificateur | declaration_produit | declaration_union

Un domaine de couleurs est une composition d'autres domaines ou d'autres classes. Les deux types de compositions autorisées en LANBADA sont les produits et les unions au sens donné dans les AMI-nets [Bonnaire 92a].

Exemple C.2 : Nous donnons, dans Lanbada C.2, des exemples de définition de domaines d'un modèle :

- le domaine *dom* correspond définit une marque composée de trois champs de type *classe1*, *classe2* et *domaine1*;
- le domaine *dom2* définit une marque polymorphe pouvant être de type *classe1*, *classe2* ou *classe3*.

```
DOMAINE dom
   PRODUIT
      classe1
      classe2
      domaine1
   FIN_PRODUIT
FIN_DOMAINE

DOMAINE dom2
   UNION
      classe1
      classe2
      classe3
   FIN_UNION
FIN_DOMAINE
```

Lanbada C.2 : Exemples de définition de domaines de couleurs.

6. DESCRIPTION DES FONCTIONS NON PRÉDÉFINIES DU MODÈLE

La description d'une fonction suit les règles de syntaxe données ci-dessous :

description_fonctions ::=
    description_fonction
    {description_fonction}
description_fonction ::=
    FONCTION identificateur
        {liste_parametres}
        domaine_resultat
        corps_de_la_fonction
    FIN_FONCTION
liste_parametres ::=
    PARAMETRE
        definition_de_parametre
        {definition_de_parametre}
    FIN_PARAMETRE
definition_de_parametre ::=
    type_parametre identificateur
type_parametre ::=identificateur
domaine_resultat ::=
    RESULTAT identificateur
corps_de_la_fonction ::=
    CORPS
        expression | parametre
    FIN_CORPS
expression ::=
    EXPR identificateur
        {liste_de_parametres} | expression
    FIN_EXPR
liste_de_parametres ::=
    identificateur
    {identificateur}

Lorsque l'on spécifie une expression, l'identificateur indiqué donne le nom de la fonction. Cette identificateur peut correspondre à une fonction prédéfinie ou à une fonction préalablement définie par l'utilisateur.

Les fonctions prédéfinies sont :

Nom	Arité	Définition
– –	2	Fonction prédécesseur d'ordre N.
++	2	Fonction successeur d'ordre N.
*	2	Multiplication de deux sous-expressions.
+	2	Addition de deux sous-expressions.
-	2	Soustraction de deux sous-expressions.
ALL	1	Diffusion pour un domaine ou une classe donnée.
PRODUCT	n	Création d'un tuple à partir d'éléments distincts.
DIFF	2	Différence de deux sous-ensembles de couleurs.
INTER	2	Intersection de deux sous-ensembles de couleurs.
SET	1	Transformation de x en le singlet {x}.

Exemple C.3 : Nous donnons, dans Lanbada C.3, la définition de la fonction *fns* qui transforme un couple de marque x (de type *classe1*) et y (de type *classe2*), en une marque composée de domaine *dclasse1_classe2* (produit des deux classes) tel que *fns* (x, y) = <x++2, y--2>.

```
FONCTION fns
   PARAMETRE
      classe1 x
      classe2 y
   FIN_PARAMETRE
   RESULTAT dclasse1_classe2
   CORPS
      EXPR COMPOSE
         EXPR ++
            VARIABLE x
            CONSTANTE 2
         FIN_EXPR
         EXPR --
            VARIABLE y
            CONSTANTE 2
         FIN_EXPR
      FIN_EXPR
   FIN_CORPS
FIN_FONCTION
```

Lanbada C.3 : Exemple de définition de fonction.

7. DESCRIPTION DES RESSOURCES DU MODÈLE

La description des Ressources d'un modèle réseau de Petri suit les règles de syntaxe données ci-après :

description_ressource ::=
RESSOURCE identificateur [domaine]
[description]
FIN_RESSOURCE
description ::=
marquage_initial | desc_interface
desc_interface ::=
INTERFACE type_interface composant_externe_associe

type_interface ::= **in** | **out**
composant_externe_associe::= identificateur
marquage_initial ::=
    marquage_simple | marquage_compose
marquage_simple ::=
    MARQ_INIT nombre [marque]
marquage_compose ::=
    COMPOSE nombre
        marquage_initial2
        {marquage_initial2}
    FIN_COMPOSE
marquage_initial2 ::=
    marque | marquage_compose2
marquage_compose2 ::=
    COMPOSE
        marquage_initial2
        {marquage_initial2}
    FIN_COMPOSE
marque ::= identificateur

Il faut tout d'abord indiquer le nom de la Ressource et s'il y a lieu, le domaine de couleurs qui lui est associé. Si aucun domaine n'est spécifié, les marques qu'elle contient sont non colorées.

S'il s'agit d'une Ressource externe, il faut donner sa "direction" (entrée ou sortie). On indique également le composant externe auquel elle est associée.

Il existe deux façons de décrire le marquage initial d'une Ressource selon que son domaine de couleurs est simple ou composé. A chaque fois qu'un profil de marque est défini, il faut spécifier le nombre d'exemplaires contenus dans la Ressource.

Si le domaine de couleurs associé à la Ressource est composé, il faut spécifier systématiquement toutes les composantes des marques présentes.

Exemple C.4 : Nous donnons, dans Lanbada C.4, des exemples de description de Ressources :

- *monochrome* est une Ressource sans domaine de couleurs contenant initialement 10 marques;
- *ress* contient des marques de type *classe1*, aucun marquage initial ne lui est associé;
- *inter* est une place d'interface en sortie du composant externe *truc*. Les données qui y sont récupérées sont de type *classe1*;
- *ress_simple* contient des marques de type *classe1*; initialement, il y a 10 marques rouges et une marque verte;
- *ress_composee* contient des marques composées de type dom_comp; initialement, il y a dix marques de profil <rouge, vert> et une marque de profil <bleu, jaune>.

```
RESSOURCE monochrome
   MARQ_INIT 10
FIN_RESSOURCE

RESSOURCE ress classe1
FIN_RESSOURCE

RESSOURCE inter classe1
   INTERFACE IN truc
FIN_RESSOURCE

RESSOURCE ress_simple classe1
   MARQ_INIT 10 rouge
   MARQ_INIT 1 vert
FIN_RESSOURCE

RESSOURCE ress_composee dom_comp
   COMPOSE 10
      rouge
      vert
   FIN_COMPOSE
   COMPOSE 1
      bleu
      jaune
   FIN_COMPOSE
FIN_RESSOURCE
```

Lanbada C.3 : Exemples de description de Ressources et de places d'interface.

8. DESCRIPTION D'UN PROCESSUS DU MODÈLE

La description d'un Processus du modèle réseau de Petri suit les règles de syntaxe données ci-dessous :

liste_des_processus :=
description_processus
{description_processus}
description_processus :=
PROCESSUS nom_du_processus
{desc_mot_d_etat}
description_des_instances
description_des_etats_processus
description_des_actions
FIN_PROCESSUS

Description du mot d'état d'un processus

La description du mot d'état d'un processus suit les règles de syntaxe données ci-dessous :

desc_mot_d_etat ::=
MOT_D_ETAT
TYPE type_domaine nom_domaine nom_composante
{**TYPE** type_domaine nom_domaine nom_composante}
FIN_MOT_D_ETAT
type_domaine ::= **DOMAINE** | **CLASSE**
nom_domaine ::= identificateur
nom_composante ::= identificateur

Les domaines des composantes d'un mot d'état sont spécifiés composante par composante. Chaque composante est nommée.

Définition d'un processus instancié

La définition des processus instanciés respecte les règles de syntaxe données ci-dessous :

description_des_instances ::=
    desc_instances
    {desc_instances}
desc_instances ::=
    INSTANCE nom_place nombre
        [marquage_initial_composante]
    FIN_INSTANCE
marquage_initial_composante::=
    marquage_simple_composante | marquage_compose_composante
marquage_simple_composante ::=
    MARQ_INIT marque
marquage_compose_composante ::=
    COMPOSE
        marquage_initial_composante
    FIN_COMPOSE
nom_composante ::= identificateur
marque ::= identificateur | **SANS_VALEUR**

Si aucun mot d'état n'est associé au Processus, la description d'une instance se réduit alors à la spécification de l'Etat_Processus initial. Le nombre de processus instanciés d'un même profil (même Etat_Processus initial et même mot d'état initial) est systématiquement indiqué.

Les valeurs à affecter aux composantes du mot d'état sont transmises *par position*[19]. Elles doivent respecter exactement le format du mot d'état. La constante prédéfinie SANS_VALEUR permet de spécifier les champs pour lesquels aucune valeur significative n'est affectée au démarrage d'un processus instancié.

Exemple C.5 : Nous donnons, dans Lanbada C.5, plusieurs exemples d'instanciation de Processus :

- *exemple 1* : le mot d'état ne comporte qu'une seule composante de type classe;
- *exemple 2* : le mot d'état ne comporte qu'une seule composante d'un type domaine défini comme le produit de deux classes;
- *exemple 3* : le mot d'état possède plusieurs composantes, l'une d'elles est de type domaine (défini comme le produit de deux classes), l'autre est de type classe;
- *exemple 4* : le mot d'état n'est pas complètement initialisé, d'ou l'utilisation de la désignation SANS_VALEUR.

[19] Au sens Ada du terme.

```
-- exemple 1
INSTANCE etat_initial 1
   MARQ_INIT rouge
FIN_INSTANCE
-- exemple 2
INSTANCE etat_initial 1
   COMPOSE
      MARQ_INIT rouge
      MARQ_INIT bleu
   FIN_COMPOSE
FIN_INSTANCE
-- exemple 3
INSTANCE etat_initial 1
   COMPOSE
      COMPOSE
         MARQ_INIT rouge
         MARQ_INIT bleu
      FIN_COMPOSE
      MARQ_INIT noir
   FIN_COMPOSE
FIN_INSTANCE
-- exemple 4
INSTANCE etat_initial 1
   COMPOSE
      COMPOSE
         MARQ_INIT SANS_VALEUR
         MARQ_INIT SANS_VALEUR
      FIN_COMPOSE
      MARQ_INIT SANS_VALEUR
      MARQ_INIT couleur
   FIN_COMPOSE
FIN_INSTANCE
```

Lanbada C.5 : Exemples d'instanciation de processus.

Description des Etats_Processus

La description des Etats_Processus d'un Processus s'effectue d'après les règles de syntaxe indiquées ci-dessous :

description_des_etats_processus ::=
desc_etats_processus
{desc_etats_processus}
desc_etats_processus ::=
ETAT_PROCESSUS attribut_etat_processus nom
{**SUCCESSEUR** nom_action}
FIN_ETAT_PROCESSUS
nom::= identificateur
nom_action::= identificateur
attribut_etat_processus ::= **SIMPLE** | **ALTERNATIF** | **TERMINAISON**

Exemple C.6 : Nous donnons, dans Lanbada C.6, plusieurs exemples de définition d'Etats_Processus :

- *fin* n'a pas de successeur, c'est un Etat_processus de terminaison;
- *etat* a un seul successeur, c'est un Etat_processus simple;
- *etat2* a trois successeurs, c'est un Etat_processus alternatif;

```
ETAT_PROCESSUS TERMINAISON fin
FIN_ETAT_PROCESSUS

ETAT_PROCESSUS SIMPLE etat
   SUCCESSEUR action
FIN_ETAT_PROCESSUS

ETAT_PROCESSUS ALTERNATIF etat2
   SUCCESSEUR action1
   SUCCESSEUR action2
   SUCCESSEUR action3
FIN_ETAT_PROCESSUS
```

Lanbada C.6 : Exemples de définition d'Etats_Processus.

Description des Actions

La description des Actions d'un Processus suit les règles de syntaxe données ci-dessous :

description_des_actions ::=
desc_d_une_action
{desc_d_une_action}
desc_d_une_action ::=
ACTION attribut_action_1 nom_action
[**PROCEDURE** identificateur]
{liste_de_contraintes}
[predicat]
PRÉDÉCESSEUR etat_pus
[**ARC_ENTRANT** nom_contrainte]
{**PRECONDITION** ressource nom_contrainte }
{**POSTCONDITION** ressource nom_contrainte}
SUCCESSEUR etat_pus
[**ARC_SORTANT** nom_contrainte]
FIN_ACTION
attribut_action_1 ::= **SIMPLE** | **SYNCHRONISEE**
ressource ::= identificateur
etat_pus ::= identificateur
nom_action ::= identificateur
nom_contrainte ::= identificateur

ARC_ENTRANT permet de décrire l'arc reliant l'Action à l'Etat_Processus dont elle est postcondition. **ARC_SORTANT** permet de décrire l'arc reliant l'Action à l'Etat_Processus postcondition. Si ces informations sont omises, l'arc est considéré comme ayant une valuation par défaut ("1").

PREDICAT permet de décrire le prédicat associé à la transition. Si aucun prédicat n'est défini pour l'Action, TRUE (prédicat toujours vrai) est assumé.

Si aucune procédure n'est associée à l'Action, un nom par défaut est créé par la phase de Génération.

Les noms des Etat_Processus prédécesseur et successeur doivent être systématiquement indiqués.

Les contraintes permettent de décrire les arcs reliant la transition aux places précondition ou postcondition. Lorsqu'une contrainte est référencée dans

arc_entrant ou dans **arc_sortant**, il s'agit de l'arc reliant l'Etat_Processus précondition ou postcondition à l'Action. Si elle est référencée dans **PRECONDITION** ou dans **POSTCONDITION**, il s'agit d'un arc lié à une composante de précondition ou postcondition ressource.

Les contraintes sont définies comme suit :

liste_de_contraintes ::=
    definition_contrainte
    {definition_contrainte }
definition_contrainte ::=
    CONTRAINTE nom_contrainte
        expression_contrainte
        {expression_contrainte}
    FIN_CONTRAINTE
expression_contrainte ::=
        [predicat]
        desc_contrainte
desc_contrainte ::= expression | parametre
expression ::=
    EXPR fonction_predefinie
        [liste_de_parametres] | expression
    FIN_EXPR
liste_de_parametres ::=
    parametre
    {parametre}
parametre ::= type_parametre identificateur
type_parametre ::= **VARIABLE** | **CONSTANTE**
fonction_predefinie ::= identificateur

Les fonctions prédéfinies sont les mêmes que pour les expressions permettant la définition d'une fonction.

Les prédicats se définissent de manière analogue aux contraintes. La grammaire utilisée est la suivante :

predicat ::=
    PREDICAT
        expression_predicat
    FIN_PREDICAT
expression_predicat ::=
    EXPR fonction_predicat
        {liste_de_parametres} | expression_predicat | expression
    FIN_EXPR
liste_de_parametres ::=
    parametre
    {parametre}
parametre ::= type_parametre identificateur
type_parametre ::= **VARIABLE** | **CONSTANTE**
fonction_predicat ::= identificateur

Les fonctions de prédicat sont données dans le tableau ci-dessous :

Nom	Arité	Définition

AND	2	ET logique entre deux expressions de prédicats.
IN	2	Test d'appartenance d'une variable à une classe ou un domaine (de type UNION)
NOT	1	NON logique d'une expression predicat.
OR	2	OU logique entre deux expressions prédicats.
<	2	Comparaison < entre deux expressions.
>	2	Comparaison > entre deux expressions.
<=	2	Comparaison ≤ entre deux expressions.
>=	2	Comparaison ≥ entre deux expressions.
=	2	Comparaison = entre deux expressions.
<>	2	Comparaison ≠ entre deux expressions.
– –	2	Fonction prédécesseur d'ordre N.
++	2	Fonction successeur d'ordre N.

		Introduction générale
Chapitre I	:	*Prototypage de systèmes parallèles*
Chapitre II	:	*Interprétation sémantique d'un réseau de Petri*
Chapitre III	:	*Méthodologie de prototypage*
Chapitre IV	:	*Structure du prototype généré*
Chapitre V	:	*Caractéristiques du prototype Ada*
Chapitre VI	:	*Le prototype centralisé*
Chapitre VII	:	*Le prototype réparti*
Chapitre VIII	:	*CPN/TAGADA : L'outil de prototypage*
		Conclusion générale
		Bibliographie
+		*Annexes & Appendices*

Appendice D :

Index des définitions

Hypothèses et Objectifs

Hypothèses
- contraintes de la méthodologie 309
- interprétation du modèle 311; 312
- modèles prototypables 81; 82; 83; 84
- réalisation du générateur de code 309; 310; 311
- Répartition
  - *décomposition en composantes 285*
  - *durée des traitements 286*

Répartition
- communications 252
- placement dynamique 240; 248
- transparence 240; 249

Le prototype

Composants externes
- interfaces (composants non-réversibles) 114
- interfaces (primitives associées) 108; 155; 156; 158

Condition 163

Constantes de démarrage 183

Etat actif
- définition 142
- graphe 142

Evénement positif
- définition 159
- traitement 159

Paquetage écran 202

Répartition
- Placement 282; 283; 284; 285
- Principes
  - *exécutable unique 248*
  - *rendez-vous distant 249*
  - *transformation des tâches 249*
- Réalisation
  - *Contexte global 255*
  - *L'aiguilleur 254*
- Tâches transparentes à la répartition 248

Ressource
- externe 132; 155
- interne 132

Squelette de contrôle 127

Méthodologie

Action
- différenciée 98; 99
- ensembles 131; 135
- fork 94; 98
- gardée 73
- hybride 98; 99
- join 94; 99
- locale 98; 99
- simple 73
- synchronisée 73; 98
- synchronisée (instance) 175

Composants externes
- abstraction 106
- définition 105
- interfaces (non-réversibles) 114
- interfaces (primitives associées) 108; 114
- non-réversible 113; 156
- places d'interface 107; 129
- réversible 114; 158; 311

Condition 163

Constantes de démarrage 183

Ensembles
- Actions 135; 140
- Etats_Processus 139
- Etat_Processus 135
- G-objets 67; 129
- séquentiels 54

Etats_Processus
- ensemble 135

Etat_Processus
- alternatif 74
- de terminaison 74
- neutre 139
- simple 74; 309

Evénement positif 159

Modèle structurel 77

Partie visible et partie privée d'un module 24

Postcondition

définition 138
processus 138
ressource 138
Précondition
contrainte 162
définition 136
dégénérée 137
jointure 161
libre 162
processus 137
ressource 137
restriction 161
sélection 161
statique 173
synchrone 173
Processus
compteur ordinal 69
contraintes 135
mot d'état 70; 142
pilote 94
Ressource
définition 74
externe 132; 155; 156; 158; 159; 309
interne 132
Système 18

Modélisation

Composants externes 105
Formalisme 303
Modèle fermé 22
Modèle ouvert 23
Partie visible et partie privée d'un module 24
Propriété
cohérence comportementale du modèle 84
conservation des processus instanciés 83
couverture de l'ensemble des Actions 81
indépendance des Processus d'un modèle 82
Réseau de Petri coloré 40
Système 18; 202
Transformation
dégrouper des Processus 90
processus implicites 89

Réseaux de Petri

Classes de couleurs 42
Définition 40
Domaines de couleurs 43
Ensembles séquentiels 54
Invariants
famille génératrice de semi-flots 76
flots 76
semi-flots 76
Marquage 40
Matrice d'incidence 76
Modèle structurel 77
Places d'interface 107
Postcondition 138
Précondition 136; 137; 161; 162; 173
Propriété
cohérence comportementale du modèle 84
conservation des processus instanciés 83
couverture de l'ensemble des Actions 81; 84
indépendance des Processus d'un modèle 82; 85
Règles de franchissement 40
Transformation
dégrouper des Processus 90
processus implicites 89

f

		Introduction générale
Chapitre I	:	*Prototypage de systèmes parallèles*
Chapitre II	:	*Interprétation sémantique d'un réseau de Petri*
Chapitre III	:	*Méthodologie de prototypage*
Chapitre IV	:	*Structure du prototype généré*
Chapitre V	:	*Caractéristiques du prototype Ada*
Chapitre VI	:	*Le prototype centralisé*
Chapitre VII	:	*Le prototype réparti*
Chapitre VIII	:	*CPN/TAGADA : L'outil de prototypage*
		Conclusion générale
		Bibliographie
+		*Annexes & Appendices*

Appendice E :

Index des figures

Introduction générale

Figure a (réalisation d'un système, étapes) 5

Introduction à la partie I

Figure b ("crise du logiciel", conséquences économiques) 11
Figure c (coût de correction d'une erreur, évolution) 12

Chapitre I

Figure I.01 (cycle de vie du logiciel) 18
Figure I.02 (étapes de réalisation) 19
Figure I.03 (interaction entre les outils de la plate-forme) 20
Figure I.04 (étapes du cycle de vie couvertes par la méthodologie) 22
Figure I.05 (décomposition d'un modèle fermé) 23
Figure I.06 (les composantes d'un modèle) 24
Figure I.07 (exemple de planche S.A.O.) 31
Figure I.08 (objets d'un diagramme PROTO) 34
Figure I.09 (exemple de diagramme PROTO) 34
Figure I.10 (représentation d'une architecture dans PROTO) 35
Figure I.11 (exemple d'activation d'une transition) 41
Figure I.12 (les trains, exemple de réseau de Petri Bien Formé) 44
Figure I.13 (interblocage entre deux transitions) 46
Figure I.14 (les trains, exemple de réseau de Petri Ordinaire) 49
Figure I.15 (interprétation centralisée) 49
Figure I.16 (exécuteur dans l'interprétation centralisée) 50
Figure I.17 (interprétation complètement décentralisée) 51
Figure I.18 (les trains, ensembles séquentiels) 54
Figure I.19 (transformation d'un modèle [Colom 86]) 56
Figure I.20 (problème lié à la décomposition [Colom 86]) 56
Figure I.21 (synchroniseur local [Colom 86]) 57
Figure I.22 (les trains, organisation du prototype Ada [Kordon 90]) 59
Figure I.23 (les trains, organisation du prototype OCCAM [Bréant 90]) 60

Chapitre II

Figure II.01 (les trains, décomposition en Processus) 70
Figure II.02 (les trains, exemple de Processus) 71
Figure II.03 (les trains, processus instanciés) 72
Figure II.04 (les trains, classification des Actions) 73
Figure II.05 (classification des Etats_Processus) 74
Figure II.06 (les trains, les Ressources) 75
Figure II.07 (exemple de modèle structurel) 78
Figure II.08 (les trains, différentes décompositions) 80
Figure II.09 (modèles ne vérifiant pas la propriété 1) 82
Figure II.10 (modèles ne vérifiant pas la propriété 2) 82
Figure II.11 (modèles ne vérifiant pas la propriété 3) 83
Figure II.12 (les trains, modèle structurel associé) 86
Figure II.13 (les trains, décomposition en G-objets) 87
Figure II.14 (transformation en vue de vérifier la propriété 1) 88
Figure II.15 (transformation en vue de vérifier la propriété 2) 90

Figure II.16 (transformation en vue de vérifier la propriété 3) 91
Figure II.17 (modèle ne vérifiant pas la propriété 3 sur lequel T2 est inapplicable) 91
Figure II.18 (applications successives de la transformation T2) 92
Figure II.19 (modèles ne vérifiant pas les propriétés 1 et 2, interprétation) 93
Figure II.20 (exemples de Processus pilotes) 94
Figure II.21 (décomposition d'un modèle ne vérifiant pas les propriétés 1 et 2) 95
Figure II.22 (modèle interprétable trivialement pour lequel l'algorithme étendu échoue) 97
Figure II.23 (modèles ne vérifiant pas la propriété 3, interprétation) 98
Figure II.24 (nouvelle classification des Actions) 99
Figure II.25 (classification des réseaux de Petri par rapport à la décomposition) 100

Chapitre III

Figure III.01 (passage d'un modèle au prototype correspondant) 103
Figure III.02 (conception modulaire d'un prototype) 104
Figure III.03 (utilisations de la notion de composant dans le cycle de vie) 106
Figure III.04 (exemple de composant externe, le clavier) 109
Figure III.05 (exemple de modèle, le producteur-consommateur relié à un clavier) 110
Figure III.06 (unité de modélisation et composant externe, équivalence) 111
Figure III.07 (places d'interfaces et services synchrones) 112
Figure III.08 (exemple d'accès à plusieurs composants externes) 113
Figure III.09 (étapes du prototypage) 115
Figure III.10 (étapes de décomposition d'un réseau de Petri) 116
Figure III.11 (décomposition en G-objets, le producteur-consommateur avec clavier) 117
Figure III.12 (étapes de production d'un prototype) 119
Figure III.13 (exemple d'architecture cible) 120
Figure III.14 (exemple d'organisation d'un prototype, le producteur-consommateur avec clavier) 121
Figure III.15 (enchaînement des étapes du prototypage) 123

Chapitre IV

Figure IV.01 (processus de prototypage) 128
Figure IV.02 (résultats de la décomposition d'un réseau de Petri) 128
Figure IV.03 (structure du prototype) 130
Figure IV.04 (les différentes parties du prototype) 131
Figure IV.05 (les différents modules du système) 133
Figure IV.06 (relations des modules-processus avec les autres modules) 134
Figure IV.07 (exemple, requête dans une base de données) 136
Figure IV.08 (les Processus, exemple du producteur-consommateur avec clavier) 143
Figure IV.09 (exemple, graphes d'états actifs du producteur consommateur avec clavier) 143
Figure IV.10 (types de services requis par un module-processus) 144
Figure IV.11 (modèle de protocole, évaluation en parallèle des Actions postcondition d'un Etat_Processus alternatif) 145
Figure IV.12 (G.M.A. du modèle de protocole pour l'évaluation en parallèle des Actions postcondition d'un Etat_Processus alternatif) 146

Figure IV.13 (modèle de protocole, choix d'une Action dans la postcondition d'un Etat_Processus alternatif) 147
Figure IV.14 (G.M.A. du modèle de protocole pour le choix d'une Action dans la postcondition d'un Etat_Processus alternatif) 148
Figure IV.15 (types de services offerts par un module-processus) 150
Figure IV.16 (synthèse des services d'un module-processus) 151
Figure IV.17 (schéma d'association de structure de contrôle aux G-objets) 152
Figure IV.18 (relations du gestionnaire de Ressources avec les autres modules) 154
Figure IV.19 (production dans une Ressource externe) 156
Figure IV.20 (consommation dans une Ressource externe liée à un composant externe non-réversible) 157
Figure IV.21 (consommation dans une Ressource externe liée à un composant externe réversible) 158
Figure IV.22 (types de services offerts par le gestionnaire des Ressources) 160
Figure IV.23 (exemple, préconditions ressources et requêtes bases de données) 162
Figure IV.24 (exemple, partie contrainte et libre d'une précondition ressource) 162
Figure IV.25 (exemple, Action avec pré et post conditions complexes) 164
Figure IV.26 (exemple, représentation d'une précondition associée à une transition) 165
Figure IV.27 (exemple, représentation d'une postcondition) 165
Figure IV.28 (types de services requis par le gestionnaire des Ressources) 166
Figure IV.29 (synthèse des services du gestionnaire de Ressources) 167
Figure IV.30 (organisation du gestionnaire de Ressources, un seul serveur) 169
Figure IV.31 (organisation du gestionnaire de Ressources, un serveur par Ressource) 169
Figure IV.32 (organisation du gestionnaire de Ressources, un serveur pour plusieurs Ressources) 170
Figure IV.33 (relations du gestionnaire des synchronisations avec les autres modules) 172
Figure IV.34 (exemple, précondition statique et synchrone) 174
Figure IV.35 (types de services offerts par le gestionnaire des synchronisations) 176
Figure IV.36 (types de services requis par le gestionnaire des synchronisations) 177
Figure IV.37 (synthèse des services du gestionnaire des synchronisations) 179
Figure IV.38 (architecture du gestionnaire des synchronisations, un seul serveur) 180
Figure IV.39 (organisation du gestionnaire des synchronisations, un serveur par Action synchronisée) 180
Figure IV.40 (relation sentre le module de contrôle et les autres modules) 182
Figure IV.41 (types de services offerts par le module de contrôle) 183
Figure IV.42 (types de services requis par le module de contrôle) 185
Figure IV.43 (synthèse des services du module de contrôle) 186
Figure IV.44 (organisation du module de contrôle, un seul gardien) 187
Figure IV.45 (organisation du module de contrôle, plusieurs gardiens) 187
Figure IV.46 (synthèse des interfaces de tous les modules du prototype) 189

Introduction à la partie II

Figure d (services réalisés par les différents modules dans le prototype Ada) 194

Chapitre V

Figure V.01 (composants d'un module fonctionnel en Ada) 201
Figure V.02 (paquetage écran) 202

Figure V.03 (sémantique du rendez-vous Ada et protocole client-serveur) 208
Figure V.04 (fonctionnement du dialogue serveur client) 210
Figure V.05 (hiérarchie dans les classes de couleurs) 214

Chapitre VI

Figure VI.01 (décomposition du gestionnaire des Ressources) 222
Figure VI.02 (description du résultat de l'évaluation d'une précondition) 224
Figure VI.03 (paquetages décrivant le serveur de Ressources) 225
Figure VI.04 (unités décrivant un Processus) 229
Figure VI.05 (paquetages décrivant un Processus) 230
Figure VI.06 (organisation du gestionnaire de synchronisations) 232

Chapitre VII

Figure VII.01 (principes généraux de répartition) 240
Figure VII.02 (principe de décomposition d'une application Ada) 242
Figure VII.03 (exemple de transformation, un exécutable par tâche Ada) 244
Figure VII.04 (exemple de transformation, un exécutable par machine hôte) 244
Figure VII.05 (messages réalisant un rendez-vous distant) 246
Figure VII.06 (messages réalisant une création de tâche distante) 247
Figure VII.07 (transformation des tâches en TTR) 248
Figure VII.08 (exemple, répartition d'une application) 249
Figure VII.09 (modélisation du protocole de réalisation d'un rendez-vous distant) 250
Figure VII.10 (G.M.A. de la modélisation du protocole de rendez-vous distant) 251
Figure VII.11 (exemple, réalisation d'un rendez-vous distant) 251
Figure VII.12 (services de la boîte à outils de répartition) 252
Figure VII.13 (exemple, architecture centralisée) 255
Figure VII.14 (exemple, architecture décentralisée) 256
Figure VII.15 (exemple, réalisation d'un rendez-vous distant en mode centralisé) 257
Figure VII.16 (contrôle de l'application en mode centralisé) 258
Figure VII.17 (modules composant une application répartie, architecture centralisée) 260
Figure VII.18 (exemple, réalisation d'un rendez-vous distant en mode décentralisé) 262
Figure VII.19 (construction d'une topologie décentralisée) 262
Figure VII.20 (les modules composant une application répartie, architecture décentralisé) 264
Figure VII.21 (services offerts par le module de contrôle réparti) 267
Figure VII.22 (exemple, déroulement d'un service dans le prototype réparti) 269
Figure VII.23 (exemple, fonctionnement incohérent du dialogue client-serveur) 271
Figure VII.24 (nouveau protocole de transmission d'un résultat) 272
Figure VII.25 (exemple, perte d'une requête liée à un jeton R) 274
Figure VII.26 (protocole de transmission d'un jeton) 275
Figure VII.27 (exemple, transmission d'une Ressource) 276
Figure VII.28 (exemple, interblocage par échec d'évaluation) 277
Figure VII.29 (exemple, interblocage indirect par échec d'évaluation) 277
Figure VII.30 (exemple, consommation d'une marque dans une Ressource externe) 280
Figure VII.31 (services offerts par le gestionnaire des Ressources délocalisé) 281
Figure VII.32 (exemple, application de la règle de placement N°1) 283

Figure VII.33 (exemple, application de la règle de placement N°2) 284
Figure VII.34 (exemple, application de la règle de placement N°3) 285
Figure VII.35 (schéma d'un l'atelier d'assemblage de carrosseries automobile) 287
Figure VII.36 (exemple, modélisation de l'atelier d'assemblage d'une carrosserie automobile) 288
Figure VII.37 (courbe, évolution des durées d'exécution) 290
Figure VII.38 (courbe, évolution du gain obtenu) 291
Figure VII.39 (courbe, évolution du nombre de messages échangés entre exécutables) 291

Chapitre VIII

Figure VIII.01 (architecture de l'atelier AMI) 302
Figure VIII.02 (exemple d'écran Macao) 304
Figure VIII.03 (exemple de réseau AMI) 307
Figure VIII.04 (architecture du générateur de code) 308
Figure VIII.05 (exemple, modèle ne vérifiant pas l'hypothèse sur la manipulation de marques composées) 310
Figure VIII.06 (exemple, vérification de l'hypothèse sur la désignations des marques précondition d'une Action) 312
Figure VIII.07 (architecture de l'application réalisant Identification + Carctérisation) 313
Figure VIII.08 (étapes de la phase d'Identification) 314
Figure VIII.09 (exemple, système relié à un composant externe) 314
Figure VIII.10 (exemple, diverses utilisations d'un composant externe) 316
Figure VIII.11 (traitements associés à la décomposition d'un modèle) 317
Figure VIII.12 (étapes de la phase de Génération) 318
Figure VIII.13 (arborescence UNIX décrivant un composant) 321
Figure VIII.14 (ajout d'un composant dans une bibliothèque) 322
Figure VIII.15 (état de réalisation du générateur de code) 324

Conclusion Générale

Figure e (composantes d'informations intégrées dans la production d'un logiciel) 328

Annexes & Appendices

Figure B.1 (exemple, modèle prototypé) 365

f

		Introduction générale
Chapitre I	:	*Prototypage de systèmes parallèles*
Chapitre II	:	*Interprétation sémantique d'un réseau de Petri*
Chapitre III	:	*Méthodologie de prototypage*
Chapitre IV	:	*Structure du prototype généré*
Chapitre V	:	*Caractéristiques du prototype Ada*
Chapitre VI	:	*Le prototype centralisé*
Chapitre VII	:	*Le prototype réparti*
Chapitre VIII	:	*CPN/TAGADA : L'outil de prototypage*
		Conclusion générale
		Bibliographie
+		*Annexes & Appendices*

Appendice F :

Index des références bibliographiques

Bases de données

[Bellahsene 82] 136
[Bernstein 78] 171; 268; 269; 273
[Bernstein 80] 273
[Bernstein 81] 46; 171; 268; 269
[Birman 85] 273
[Eswaran 76] 273
[Garcia-Molina 85] 273
[Gardarin 86] 136; 160; 224; 226
[Marzullo 91] 273
[Raynal 91] 273
[Rosenkrantz 77] 46

Formalismes

Réseaux de Petri

[Arnold 92] 29
[Ayache 85] 4; 62
[Berthelot 86] 306
[Bonnaire 92a] 213; 306; 307; 365; 399
[Brams 82] 29; 39; 47; 48; 67; 72; 73; 75; 76
[Bréant 89] 47
[Bréant 90] 47; 48; 306
[Bruno 86] 47; 48; 115
[Chehaibar 90] 4; 62
[Chehaibar 91] 4; 39; 75; 100; 103
[Chiola 87] 76; 79; 306; 309
[Chiola 90] 4; 39; 41; 44; 72; 77; 135; 307; 365
[Chiola 91] 39; 77
[Chvatal 83] 76; 79
[Colom 86] 47; 48; 81; 89; 115
[Cousin 88a] 48
[Couvreur 90a] 4; 39; 77; 100; 306
[Couvreur 90b] 4; 39; 100
[Di Giovanni 90] 29
[Dutheillet 91] 4; 39; 41; 44; 72; 100; 135; 307; 365
[El Fallah 89] 47; 48
[Finkel 90] 305
[Girault 87] 4
[Hack 74] 6; 67; 69
[Haddad 88] 75; 76
[Haddad 91] 4; 39
[Hauschildt 87] 47; 48; 115
[Hubert 90] 4; 29; 39; 62
[Jensen 81] 4; 39; 72; 77
[Kordon 89] 47
[Kordon 90] 47; 48; 306
[Memmi 83] 67; 75
[Moitessier 91] 4; 62
[Murata 86] 47; 48; 115
[Murata 90] 39
[Nelson 83] 47; 48; 115
[Paludetto 90] 29; 47
[Paludetto 91] 29; 47; 115
[Peterson 81] 39
[Petrucci 91] 4; 39; 100
[Shapiro 91] 4; 62
[Silva 79] 47; 48; 115
[Silva 80] 47; 48; 115
[Silva 82] 47; 48; 115
[Souissi 89] 73; 74; 103
[Souissi 90] 75; 103
[Taubner 87] 47; 48; 115
[Thuriot 85] 47; 48; 115
[Trèves 89] 76; 79; 306
[Tu 90] 4; 62
[Valette 82] 47; 48; 115
[Valette 83] 48
[Vautherin 84] 77
[Vinci 90] 4; 62

Autres

[Arbaoui 91] 28
[Arnold 92] 62
[Coad 92] 28
[De Marco 78] 28; 33
[Di Giovanni 90] 29
[Gaudel 91] 29; 62

[HOOD 89] 28; 57
[Loomis 87] 28
[Paludetto 90] 29
[Paludetto 91] 29

Génie Logiciel

[Ada 83] 107; 108
[Arthaud 89] 20
[Bernard 88] 299; 324
[Bernard 89] 79; 299; 324
[Bernard 90] 5; 79; 195; 299; 301; 324
[Boehm 76] 12
[Boehm 81] 12
[Booch 86] 105; 111
[Booch 87] 105; 107; 108
[Brinkkemper 90] 4
[Brunet 91] 4
[Budde 84] 27
[Case 86] 28
[Coad 92] 105; 111
[Estraillier 91] 4; 27; 63
[Floyd 84] 27; 200
[GAO 79] 11
[Hallmann 91] 25; 200
[ISO 7498] 23
[Jaulent 90] 17
[Kurtz 89] 4
[Lientz 78] 11
[Murphy 89] 24; 62
[Pomello 90] 5
[Stroustrup 87] 105; 111
[Van Glabbeck 90] 5
[Vonk 92] 11; 27

Langages

Ada

[Ada 83] 12; 23; 107; 108; 200; 201; 204; 205; 206; 243; 245; 246; 250; 251; 324
[Ada 91a] 216
[Ada 91b] 216
[Atkinson 88] 239
[Bazalgette 91] 239; 242; 243; 244; 245; 247
[Boussand 92] 287; 290; 294
[Heitz 91] 239; 240; 242
[Ichbiah 86] 204; 209
[Kordon 91c] 239; 240; 242; 243; 245; 266; 294
[Millard 91] 239; 242; 245
[Richard Foy 91] 204; 208; 211

Autres

[Jensen 78] 12
[Wirth 83] 12

Modélisation

[Ada 83] 23
[Arbaoui 91] 28
[Arnold 92] 62
[Arthaud 89] 20
[Ayache 85] 4; 62
[Bachatene 92] 23; 111; 130
[Bernard 88] 299
[Bernard 89] 299
[Bernard 90] 5; 195; 299; 301
[Bonnaire 92a] 306; 307
[Booch 86] 12; 28
[Booch 87] 23
[Brams 82] 39
[Brinkkemper 90] 4
[Brunet 91] 4
[Burns 91] 118
[Chehaibar 90] 4; 62
[Chehaibar 91] 39; 75; 103
[Chiola 91] 5; 39
[Coad 92] 13; 23; 28
[Couvreur 90a] 39
[Couvreur 90b] 39
[De Marco 78] 28; 33
[Dutheillet 91] 39
[Estraillier 91] 4; 19; 27; 61; 63; 299

[Estraillier 92] 19; 299
[Girault 87] 4
[Haddad 91] 39
[Kurtz 89] 4
[Loomis 87] 28
[Meyer 88] 28
[Moitessier 91] 4; 13; 62
[Murata 90] 39
[Pomello 90] 5
[Shapiro 91] 4; 62
[Souissi 89] 103
[Souissi 90] 75; 103
[Tu 90] 4; 62
[Valette 82] 75
[Van Glabbeck 90] 5
[Vinci 90] 4; 62
[Vonk 92] 13

Prototypage

[Bréant 89] 47; 60
[Bréant 90] 47; 48; 59; 60; 61; 169; 306
[Bruno 86] 13; 47; 48; 115
[Budde 84] 27
[Burns 90] 13; 30; 33
[Burns 91] 13; 30; 33; 35; 118
[Cassigneul 91] 13; 27; 30; 62; 63
[Colom 86] 13; 47; 48; 50; 51; 55; 56; 57; 58; 59; 61; 74; 115
[Cousin 88a] 48
[Cousin 88b] 59; 63; 139
[Dollas 90] 25
[El Fallah 89] 26; 47; 48; 50
[Estraillier 91] 19; 61; 63; 299
[Estraillier 92] 19; 299
[Floyd 84] 11; 27; 200
[Hallmann 91] 11; 25; 200
[Hauschildt 87] 13; 47; 48; 52; 115; 139; 169
[Kordon 89] 47; 59; 60; 139
[Kordon 90] 47; 48; 59; 60; 61; 139; 306
[Kordon 91a] 168
[Kordon 91b] 63; 67
[Lintulampi 90] 26
[Murata 86] 13; 47; 48; 115
[Nelson 83] 13; 47; 48; 50; 115
[Paludetto 90] 13; 47; 57; 58; 63
[Paludetto 91] 13; 47; 57; 61; 63; 115
[Petersen 90] 26
[Royals 90] 26
[Silva 79] 13; 47; 48; 50; 55; 115
[Silva 80] 13; 47; 48; 50; 55; 115
[Silva 82] 13; 47; 48; 50; 55; 115
[Taubner 87] 13; 47; 48; 115; 139; 169
[Thuriot 85] 13; 47; 48; 115
[Valette 82] 13; 47; 48; 115
[Valette 83] 13; 47; 48; 50; 115
[Vonk 92] 27

Systèmes parallèles et répartis

[Atkinson 88] 239
[Avizienis 87] 184; 243; 259; 263
[Bazalgette 91] 239; 242; 243; 244; 245; 247
[Bernard 91] 247
[Bernstein 78] 171; 268; 269; 273
[Bernstein 80] 46; 171; 273
[Bernstein 81] 46; 268; 269
[Billionnet 89] 119; 282; 317
[Birell 84] 144; 145; 147; 245
[Birman 85] 171; 273
[Birman 89] 171
[Blanc 90] 188; 240
[Boussand 92] 287; 290; 294
[Burns 81] 243; 248
[Cheng 90] 263
[Coffman 71] 46; 243
[Eswaran 76] 46; 273
[Folliot 89] 247; 253
[Garcia-Molina 85] 171; 273
[Garcia-Molina 89] 273
[Goscinski 91] 247; 295
[Heitz 91] 239; 240; 242
[Hoare 74] 45
[Kordon 91c] 239; 240; 242; 243; 245; 294
[Krakowiak 85] 69; 159; 268; 318
[Kumar 91] 273
[Lamport 78] 46; 171; 272; 273; 274

[Lavarenne 89] 317
[Le Lann 77] 171; 273; 274
[Lo 88] 119; 282; 317
[Marzullo 91] 252; 273
[Millard 91] 239; 242; 245
[Mullender 89] 132; 182; 252; 256
[Nelson 81] 144; 145; 147; 245
[Nelson 90] 184; 243; 259; 263
[Pujolle 85] 258
[Raynal 84] 171; 188; 256; 268
[Raynal 87] 133; 171; 182
[Raynal 88] 240; 256
[Raynal 91] 188; 268; 269; 273
[Rosenkrantz 77] 46
[Sens 90] 254
[Shapiro 86] 248
[Sinclair 87] 119; 282; 317
[Stevens 90] 193; 242; 250; 252; 253
[Theimer 88] 247; 253
[Weihl 89] 144; 145; 147; 245

Autres références

[Amarger 88] 305
[Bonnaire 92b] 307
[Dagron 89] 305
[Hein 88] 305
[Lanord 92] 260
[Le Roch 90] 332
[Le Roch 91] 332

f