CONSERVATOIRE NATIONAL DES ARTS ET METIERS

PARIS

MEMOIRE

présenté en vue d'obtenir

le DIPLOME d'INGENIEUR C.N.A.M.

en

INFORMATIQUE

par

Frank SINGHOFF

Mise en oeuvre du modèle Saturne sur CHORUS/COOL

Soutenu le 26 septembre 1996

JURY

PRESIDENT : M. Gérard FLORIN

MEMBRES : Mme. Claire CAMPAN
Mme. Isabelle DEMEURE
Mme. Laurence DUCHIEN
M. Stéphane CARREZ
M. Eric GRESSIER
M. Yann LE BIANNIC

CONSERVATOIRE NATIONAL DES ARTS ET METIERS

PARIS

MEMOIRE

présenté en vue d'obtenir

le DIPLOME d'INGENIEUR C.N.A.M.

en

INFORMATIQUE

par

Frank SINGHOFF

Mise en oeuvre du modèle Saturne sur CHORUS/COOL

Les travaux relatifs au présent mémoire ont été effectués chez Dassault Aviation , à Saint Cloud, sous la direction de Mme. Claire CAMPAN et M. Eric GRESSIER.

Mise en oeuvre du modèle Saturne sur CHORUS/COOL

Mémoire d'ingénieur C.N.A.M., PARIS, 1996

Résumé

Devant la croissante complexité des systèmes embarqués, les concepteurs et réalisateurs de logiciels ont besoin d'outils adaptés. Ces outils doivent leur permettre de spécifier, réaliser et certifier efficacement leurs applications embarquées. En dehors de leur réalisation, l'exécution de ces logiciels est également très difficile à maîtriser car ils ont de très fortes contraintes temps réel et de sûreté de fonctionnement. Parallèlement, il devient aujourd'hui de plus en plus nécessaire d'utiliser des systèmes distribués, et ce pour des problèmes de performances, de localisations des équipements physiques (tel que des capteurs), de tolérance aux pannes ou à des fins de réduction de coût du système. Malheureusement, l'exécution d'applications temps réel dans un environnement distribué reste encore mal dominée. Sur cette architecture distribuée, il est nécessaire de construire un environnement d'exécution permettant au logiciel applicatif de respecter ses contraintes. L'environnement d'exécution devra en particulier garantir le déterminisme des temps d'exécution et de communication. Saturne est un modèle d'exécution élaboré par le **C**entre d'**E**tudes et de **R**echerches de **T**oulouse, pour les applications temps réel embarquées dans un environnement distribué. Ce document propose d'étudier les mécanismes à mettre en oeuvre pour pouvoir implanter Saturne sur un système d'exploitation distribué temps réel : le système CHORUS.

Mots clefs

Temps réel, avionique modulaire, Saturne, Esterel, CHORUS, COOL, tolérance aux pannes, systèmes distribués, CORBA

Key words

real time, modular aircraft design, Saturne, Esterel, CHORUS, COOL, fault tolerance, distributed systems, CORBA

Remerciements

Je tiens à remercier toutes les personnes qui m'ont aidé à réaliser ce travail. Je remercie tout particulièrement Eric Gressier, Maître de Conférence au Conservatoire National des Arts et Métiers pour son suivi et ses conseils durant ce travail. Je le remercie aussi pour son soutien durant ces trois dernières années d'études au Conservatoire National des Arts et Métiers. Je remercie aussi tout le personnel du département études logiciels de Dassault Aviation qui m'ont apporté leur aide et spécialement Claire Campan, ingénieur, qui reçoit toute ma gratitude et ma sympathie pour ses conseils permanents, pour m'avoir fourni l'aide technique lors de la phase de développement et pour son encadrement. Je remercie aussi Christine Ledey pour ses patientes explications et son aide à l'intégration de l'application de suivi de terrain dans le deuxième prototype ainsi que Yann Le Biannic pour son soutien et toutes les démarches qu'il a entreprises pour que cette étude s'effectue dans de bonnes conditions. Enfin, je salue toutes les autres personnes, du Conservatoire National des Arts et métiers, de Dassault Aviation et d'ailleurs qui m'ont offert leur soutien durant ces quatre dernières années.

Table des matières

1 Introduction **11**

2 Le modèle synchrone fort **13**
2.1 Introduction aux langages synchrones 13
2.2 Exemples de langages synchrones 15
2.2.1 Le modèle flot de données 15
2.2.2 Le langage Lustre 16
2.2.3 Le langage Signal 19
2.3 Description du langage Esterel 22
2.3.1 Syntaxe et sémantique du langage Esterel 22
2.3.2 Exécution des programmes Esterel 29
2.3.3 L'exemple de l'émetteur-récepteur 31
2.3.4 L'exemple de l'additionneur de matrices 34
2.4 Validation de programmes synchrones 38
2.5 Conclusion 39

3 Le modèle Saturne **41**
3.1 Le modèle synchrone faible 41
3.2 Le modèle Saturne 42
3.2.1 L'architecture de Saturne 42
3.3 Extensions du modèle Saturne 46
3.3.1 Comportements non modélisation avec Saturne 46
3.3.2 Les extensions proposées :Saturne multi-synchrone 46
3.4 Le modèle Saturne Objet 48
3.4.1 Description du modèle Saturne Objet 48
3.5 Contraintes apportées par le modèle Saturne sur l'environnement d'exécution 49

4 Le sous-système à objets de CHORUS :COOL V3 **51**
4.1 Le système d'exploitation CHORUS 51

4.1.1 Les abstractions de CHORUS 52
4.1.2 L'architecture des systèmes CHORUS 52
4.2 Le standard CORBA 55
4.2.1 Le modèle de programmation 56
4.2.2 L'architecture OMA 57
4.2.3 La structure d'un bus à objets 58
4.2.4 Le langage IDL 60
4.2.5 Conclusion sur CORBA 61
4.3 L' ORB de CHORUS:COOL V3 62
4.3.1 Les abstractions de COOL 62
4.3.2 Le compilateur CHIC 64
4.3.3 Les services supplémentaires offerts par COOL 65
4.3.4 Conclusion sur COOL 68

5 Le sous-système Saturne sur CHORUS/COOL 71
5.1 Introduction 71
5.2 Architecture du sous-système Saturne sur CHORUS/COOL 72
5.3 Simulation du top 73
5.3.1 Présentation du problème 73
5.3.2 Réalisation sous CHORUS 74
5.4 Résolution des problèmes d'ordonnancement 74
5.4.1 L'ordonnancement sur CHORUS 74
5.4.2 Mise en oeuvre avec Saturne 76
5.5 Les noyaux synchrones 78
5.5.1 Architecture des acteurs SM 78
5.5.2 Communications entre les noyaux synchrones 79
5.6 Description du protocole entre le noyau et l'AM 80
5.6.1 Description d'une tâche transformationnelle 80
5.6.2 Protocole entre l'AM et le noyau synchrone 82
5.7 Implantation du modèle Saturne sur COOL 83
5.7.1 Les objets COOL de Saturne 83
5.7.2 Modes de communication utilisés pour les invocations de méthodes 85
5.8 Conclusion 86

6 Performances et résultats des prototypes réalisés 87
6.1 Le premier prototype 87
6.1.1 Architecture du prototype 87
6.1.2 Objectifs visés 89
6.2 L'application de suivi de terrain 90

6.2.1 Architecture du prototype 90
6.2.2 Génération automatique du code 91
6.2.3 Objectifs visés 95
6.3 Le troisième prototype 95
6.4 Le dernier prototype 97
6.5 Mesure des performances des prototypes 100
6.5.1 Performances des prototypes 100
6.5.2 Techniques permettant d'améliorer les performances des prototypes 113
6.6 Adaptation de COOL et de CHORUS au modèle Saturne 114

7 Conclusion 117

A Sources de l'exemple de l'émetteur/récepteur du chapitre deux 125
A.1 Fichier d'entête (esterel.h): 125
A.2 Fichier principal (main.h): 125
A.3 Fichiers concernant les signaux: 129
A.3.1 Signal de sortie ETAT du noyau EMETTEUR: 129
A.3.2 Signal de sortie DATA du noyau EMETTEUR: 129
A.3.3 Signal de sortie O du noyau RECEPTEUR: 130

B Sources Esterel des prototypes un, trois et quatre 131
B.1 Les sources Esterel de Pepin 131
B.2 Les sources Esterel de Graines 132
B.3 Les sources Esterel de Noyau 133

C L'interface homme machine du deuxième prototype 139

Liste des figures

2.1 Exemple d'un graphe flot de données 15
2.2 Un noeud en Lustre 18
2.3 Le graphe flot de données correspondant au noeud Lustre 18
2.4 Compteur pour pair=COMPTEUR(0,2,FALSE) 19
2.5 L'instruction de retard de Signal 20
2.6 L'instruction *when* de Signal 21
2.7 L'instruction *default* de Signal 21
2.8 Déclaration d'une variable en Esterel 22
2.9 Boucle Esterel illégale 24
2.10 Emission sur le signal foo de la valeur 100 25
2.11 Réception de signaux grâce à une instruction *case/await* 26
2.12 Réception sur le signal foo 26
2.13 Test de présence d'un signal 26
2.14 Déclaration de tâches Esterel 27
2.15 L'instruction *watching* 28
2.16 L'instruction *watching* avec une clause *timeout* 28
2.17 L'instruction *every* 28
2.18 L'instruction *upto* simulée par un *watching* 29
2.19 L'instruction *upto* 29
2.20 Construction d'un exécutable Esterel 31
2.21 Exemple de l'émetteur et du récepteur 32
2.22 Source Esterel de l'émetteur 33
2.23 Source Esterel du récepteur 34
2.24 Exemple de l'additionneur de matrices 34
2.25 Source Esterel du générateur de matrices 36
2.26 Source Esterel de l'additionneur de matrices 37

3.1 L'architecture Saturne 42
3.2 Utilisation des primitives de Saturne 44
3.3 Le problème "50Hz-80Hz" 45
3.4 Le problème du "court-circuit" 45

3.5 Le modèle Saturne Objet 49

4.1 Les abstractions CHORUS 51
4.2 L'architecture des systèmes CHORUS 53
4.3 Le micro noyau CHORUS 53
4.4 Le sous système ClassiX R2 54
4.5 Le modèle de programmation de CORBA 56
4.6 L'architecture OMA 57
4.7 La structure d'un ORB 58
4.8 Exemple de code en IDL 61
4.9 Architecture de COOL V3 63
4.10 Le compilateur CHIC 65
4.11 Notifications sur chiens de garde 65
4.12 Notifications sur entrées/sorties 66
4.13 Objets Lecteurs/rédacteurs de COOL 67
4.14 Sémaphores et mutex de COOL 67

5.1 Architecture du sous système Saturne 72
5.2 Les priorités des acteurs Saturne 76
5.3 Description de l'acteur SM 79
5.4 La classe des tâches transformationnelles 81
5.5 L'interface IDL d'un acteur AM 84
5.6 L'interface IDL d'un noyau synchrone 84
5.7 Communications du sous-système Saturne réalisées en COOL 85

6.1 Le premier prototype 88
6.2 L'application de suivi de terrain 90
6.3 La grammaire en BNF utilisée par le générateur de code 91
6.4 Une application Saturne 92
6.5 Description d'une application Saturne 93
6.6 L'application Esterel utilisée pour la mise en oeuvre du vote 96
6.7 Le mécanisme de double vote 97
6.8 Exemple d'un fichier de configuration 98
6.9 Temps d'exécution d'une méthode COOL 101
6.10 Temps d'exécution d'une méthode COOL 101
6.11 Temps d'exécution d'une méthode COOL 102
6.12 Temps de calcul des tâches T1 et T2 104
6.13 Temps de calcul des tâches T3 104
6.14 Communications entre les AM et les SM 106
6.15 Communications entre les SM 106
6.16 Nombre de tops pour exécuter l'application 107

6.17 Nombre d'exécution des noyaux Esterel 108
6.18 Temps d'exécution des noyaux Esterel 109
6.19 Temps de calcul des tâches en fonction de l'intervalle 110
6.20 Nombre d'exécution des noyaux en fonction du quantum 111
6.21 Temps d'exécution des noyaux en fonction du quantum 111
6.22 Temps de calcul des tâches en fonction du quantum 112

Liste des tableaux

2.1 Exemple d'instructions Lustre 16
2.2 Instructions impératives d'Esterel 23
2.3 Exemples de déclarations de signaux en Esterel 25

6.1 Configurations utilisées pour la répartition du prototype 1 105

Chapitre 1

Introduction

Aujourd'hui, la société Dassault Aviation spécifie les systèmes de ses avions et ce sont des équipementiers (Thomson, Sagem, Sextant Avionique, etc) qui réalisent ces équipements. Le choix des composants matériels et logiciels est propre à chaque équipementier. Cette hétérogénéité entraîne un surcoût de fabrication et de maintenance.

L'avionique modulaire a pour objectif de réduire ce surcoût en définissant une architecture matérielle et logicielle permettant l'utilisation de logiciels dits "sur étagère". Par logiciels "sur étagère", on entend des composants logiciels standards pouvant être achetés dans l'industrie. On évoque souvent ce concept par le terme de COTS[1]. L'utilisation de logiciels sur étagère dans les systèmes embarqués fait actuellement l'objet d'une étude par un groupement d'industriels européens et américains : l' OSAF[2][43]. L'avionique modulaire doit aussi permettre de réduire le coût d'une mission en augmentant le nombre d'heures de vol sans maintenance. Pour cela, les systèmes des avions devront intégrer des mécanismes de tolérance aux pannes pour compenser d'éventuelles défaillances en vol.

Parallèlement à l'avionique modulaire, Dassault Aviation étudie des méthodes de spécification de logiciels temps réel embarqués où les applications s'exécutent selon un modèle synchrone faible[8] élaboré par le CERT[3] : le modèle Saturne[17, 1, 46]. CHORUS[52] étant le système d'exploitation évalué dans le cadre des études d'avionique modulaire, il est également choisi pour cette étude dont l'objectif est de déterminer dans quelles conditions le modèle Saturne peut être mis en oeuvre sur un système d'exploitation distribué temps-réel. Une étude des mécanismes de tolérance

[1]COTS pour **C**ommercial **O**n **T**he **S**helf products.

[2]OSAF pour **O**MI **S**oftware **A**rchitecture **F**orum. Ce groupement comprend entre autres Chorus Systèmes, IONA Technologies, l'Object Management Group, Interglossa, etc.

[3]CERT pour **C**entre d'**E**tudes et de **R**echerches de **T**oulouse.

aux pannes ainsi que l'analyse de l'apport des technologies CORBA[4][41, 38] dans le développement d'applications temps réel complète ces objectifs.

Le modèle Saturne s'appuie sur le modèle synchrone fort. Nous présenterons donc tout d'abord le modèle synchrone fort dans le chapitre deux. Puis nous étudierons le modèle Saturne dans le chapitre trois. Nous définirons ensuite dans le chapitre quatre la plate-forme qui nous a servi à réaliser nos expérimentations : le système d'exploitation CHORUS et le sous système COOL[62][5]. COOL est le bus à objets au standard CORBA 2 développé par la société Chorus Systèmes. Dans le chapitre cinq, nous spécifierons l'architecture du sous-système Saturne ainsi que sa conception avec COOL. Les mécanismes de tolérance aux pannes ne seront pas abordés dans ce document. Pour obtenir une description détaillée des mécanismes de tolérance aux pannes utilisés, nous renvoyons le lecteur à [53]. Enfin, la conclusion sur cette étude de faisabilité sera précédée par le chapitre six qui analyse les résultats obtenus sur les différents prototypes.

[4] CORBA pour **C**ommon **O**bject **R**equest **B**roker **A**rchitecture.

[5] COOL pour **C**HORUS **O**bject **O**riented **L**ayer.

Chapitre 2

Le modèle synchrone fort

Dans ce chapitre, nous présenterons le modèle synchrone fort. Après cette définition, nous étudierons trois langages synchrones qui sont les plus connus et nous nous intéresserons plus particulièrement au langage Esterel. Nous présenterons ensuite les preuves qui peuvent être effectuées sur les programmes écrits en Esterel.

2.1 Introduction aux langages synchrones

* Notion de systèmes réactifs

La notion de systèmes réactifs fut introduite par D. Harel et A. Pnueli[24]. Par systèmes réactifs, on entend généralement un système qui réagit immédiatement à des entrées en provenance d'un environnement extérieur en fournissant des sorties instantanément. Par exemple, on peut citer les processus de contrôle industriel en temps réel, les automatismes (distributeurs de boissons, billets de banque), etc. Ces systèmes réactifs ont été opposés aux systèmes transformationnels par A. Pnueli. Les systèmes transformationnels possèdent leurs entrées lors du démarrage de leur exécution, et fournissent un résultat à la fin de leur exécution. On considère souvent un troisième type de système qui est le système interactif. Ce dernier réagit aussi à des entrées événementielles, mais à son rythme.

* Le modèle synchrone fort

Dans le domaine d'application des systèmes réactifs, les langages procéduraux ne sont pas adaptés. Généralement, pour développer ce type de système, on utilisait plutôt :

- Des langages asynchrones comme CSP[1]/OCCAM ou ADA : malheureusement, le type de parallélisme introduit dans ces langages apporte aussi de l'indéterminisme, ce qui est à proscrire dans les systèmes temps réel critiques comme les systèmes embarqués,
- Des automates d'états finis : ceux-ci deviennent très vite compliqués à concevoir quand leur taille est importante,
- Des primitives système de bas niveau : elles ne permettent pas d'effectuer des certifications de logiciel.

De plus, toutes ces méthodes ne permettent pas d'exprimer les contraintes de temps d'exécution (bien qu'il existe parfois des extensions comme c'est le cas pour CSP[13]). Pour développer ce type de système, des langages spécifiques ont donc été proposés : les langages synchrones. Les langages synchrones les plus connus sont Lustre[23], Signal[21, 22] et le plus ancien des trois : Esterel[2, 4, 5]. L'avantage essentiel de ces langages est qu'ils font cohabiter, grâce au modèle synchrone fort, à la fois le parallélisme d'expression, et une exécution déterministe. De plus, ils permettent de mettre en oeuvre des preuves logiques et temporelles des applications.

Le modèle synchrone fort repose sur un certain nombre de concepts :

- L'hypothèse fondamentale du synchronisme : ce modèle suppose que les temps de réaction des noyaux[2] doivent être nuls. En d'autres termes, on suppose dans ces langages que la vitesse de calcul des machines est infiniment rapide. Cette hypothèse qui semble absurde dans la réalité est en fait tout à fait réaliste. En effet, dans ce type de langage on ne considère pas le temps physique mais plutôt "l'instant d'activation", c'est à dire le moment où les signaux d'entrées arrivent au système réactif. **Le temps physique est discrétisé et on utilise un temps logique**. L'intérêt essentiel de cette hypothèse est qu'elle simplifie les preuves de programmes. Les contraintes temporelles sont alors garanties par l'environnement d'exécution du programme synchrone qui doit s'assurer qu'à l'instant t, les traitements de l'instant t-1 sont terminés. Dans la réalité, pour valider l'hypothèse du modèle synchrone fort, il faut que l'intervalle entre deux instants d'activation soit supérieur au temps de calcul du système réactif : un jeu de signaux ne doit pas arriver au système avant que le jeu précédent ne soit complètement traité. En ne mélangeant pas les différents jeux de signaux, on fournit aux systèmes réactifs une propriété importante qui est nécessaire aux systèmes temps réels : leur déterminisme. Un programme déterministe est

[1] CSP pour **C**ommunicating **S**equential **P**rocesses.
[2] Par noyau, nous entendons système réactif.

un programme qui délivre toujours les mêmes sorties quand il reçoit les mêmes entrées.

- Diffusion des signaux : Le modèle synchrone fort offre une diffusion des données entre les différentes tâches qui est **instantanée**.
- Atomicité de l'exécution du système réactif : quand le système réactif reçoit ses données d'entrée, il s'exécute instantanément et surtout de manière **atomique**.

2.2 Exemples de langages synchrones

Avant d'étudier les langages Lustre et Signal, nous allons d'abord faire un bref rappel sur le modèle de flot de données qui est la base de ces deux langages.

2.2.1 Le modèle flot de données

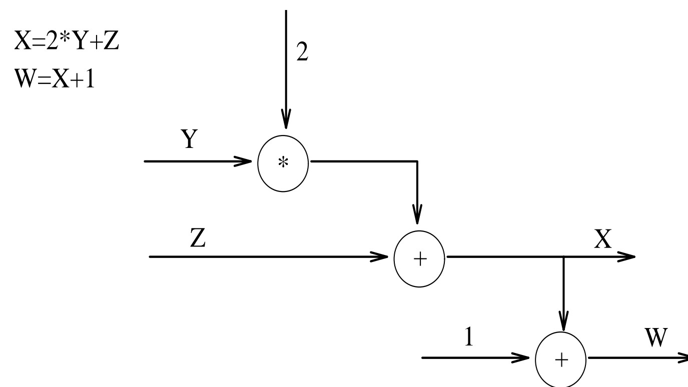

Figure 2.1: Exemple d'un graphe flot de données

Les modèles flots de données sont représentés par un graphe orienté, qui peut être cyclique, dont les sommets sont des opérations et où les arcs contiennent des données. Le graphe possède des arcs sans sommet prédécesseur, ce qui correspond aux données en entrée du graphe, et des arcs sans successeur, ce qui correspond aux résultats fournis par le graphe. Le déclenchement d'un calcul intervient lorsque toutes les entrées de l'opérateur sont présentes. Ainsi, dans notre exemple en figure 2.1, la multiplication entre Y et 2 ne pourra être déclenchée que lorsque ces deux opérandes seront présents à l'entrée de l'opérateur (ce mécanisme permet de mettre en évidence les contraintes de précédence dans un graphe flot de données). Il existe de nombreuses variantes[6] de flot de données. Ainsi, bien qu'en général les calculs soient déclenchés de manière asynchrone et que l'on ne fasse aucune supposition

Instructions	Flots de données				
k	Vrai	Vrai	Faux	Faux	Vrai
I	10	5	2	9	12
V=I *when* k	10	5			12
current V	10	5	5	5	12

Tableau 2.1: Exemple d'instructions Lustre

sur la vitesse de calcul des noeuds, il existe des modèles où ils sont tous déclenchés simultanément[3]. Les arcs, quant à eux, peuvent soit contenir une seule donnée, soit être constitués d'une file d'attente. L'utilisation de ce type de modèle pour les langages synchrones apporte de nombreux avantages :

- Les langages synchrones sont très souvent parallèles. Le modèle flot de données est adapté aux programmes parallèles. En effet, tous les opérateurs dont les entrées sont disponibles peuvent être exécutés en parallèle,
- Parmi les domaines d'application des systèmes réactifs figurent l'automatique et l'électronique, où les spécifications et les modélisations sous forme de flot de données sont courantes. La traduction directe d'un graphe flot de données vers un langage synchrone permet d'utiliser les spécifications éditées sous forme graphique par les utilisateurs,
- Le modèle permet une certaine modularité car il offre la possibilité de créer des nouveaux opérateurs, constitués de sous-graphes,
- Enfin, le modèle repose sur des fondements mathématiques permettant des preuves formelles.

2.2.2 Le langage Lustre

Le langage Lustre[23] est basé sur un modèle de flot de données synchrone. Par flot de données synchrone, on entend un flot où chaque donnée est indicée par le temps. Lustre définit la notion de variable : une variable est une suite éventuellement infinie de valeurs d'un type donné et d'une horloge. Chaque valeur de la variable est datée par l'horloge. Le langage autorise les types entiers, booléens, réels, ainsi que tous types importés depuis un langage hôte[4]. Sur ces flots, ou variables, Lustre fournit des opérateurs temporels :

[3] C'est le cas des architectures systoliques[64].

[4] Par langage hôte, on entend le langage qui est utilisé pour exécuter un programme synchrone : en effet, on a besoin d'un langage qui permet de faire le lien entre le système d'exploitation et le programme synchrone.

- L'opérateur "*pre*" : pour précédent. Cet opérateur permet d'atteindre une valeur du flot à l'instant précédent l'horloge courante,

- L'opérateur "$\longrightarrow$" : cet opérateur sert à introduire une valeur initiale dans le flot de données,

- L'opérateur "*when*" : celui-ci permet de conditionner la délivrance d'une valeur d'une variable grâce un flot booléen. Pour une instruction I *when* k, les valeurs de I sont renvoyées lorsque les valeurs du flot booléen k sont vraies. Le tableau 2.1 fournit un exemple de cet opérateur.

- L'opérateur "*current*" : ce dernier opérateur permet de récupérer la valeur courante quand la variable est présente. Il renvoie la dernière valeur obtenue dans le cas contraire. La variable est dite présente à l'instant t si elle possède une valeur à cet instant. Ainsi dans notre exemple du tableau 2.1, cet opérateur renvoie deux fois la valeur cinq car sur deux instants, la variable V n'est pas présente.

Tous les opérateurs de Lustre supportent l'hypothèse du synchronisme fort : le temps de calcul est nul. Bien sûr, Lustre permet la définition d'opérateurs plus complexes. La syntaxe de Lustre est déclarative. Un programme Lustre est un ensemble d'équations mathématiques et d'assertions. Lustre permet d'écrire des programmes très modulaires : la modularité de Lustre est héritée du modèle flot de données. Chaque module est appelé un "noeud" et correspond à un nouvel opérateur du graphe flot de données. Le passage d'un modèle flot de données à un programme Lustre est donc très facile.

```
node COMPTEUR (Val-init, Val-incr : int ; RAZ : bool)
returns (n : int);
let
      n=Val-init ⟶ if RAZ then Val-init
            else pre(n) + Val-incr;
tel
```

Figure 2.2: Un noeud en Lustre

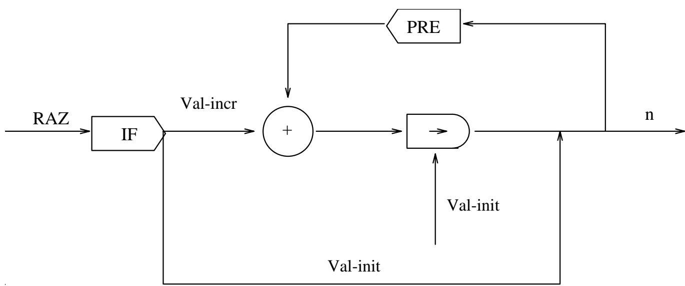

Figure 2.3: Le graphe flot de données correspondant au noeud Lustre

L'exemple de la figure 2.2 montre aisément la correspondance entre noeud et graphe de flot de données. Cet exemple représente un compteur. La variable n est incrémenté de *Val-incr* à chaque top de l'horloge qui lui est associée. L'instruction *pre(n)* permet de récupérer la valeur de n au top précédent. Le noeud pourrait être instancié par "pair=COMPTEUR(0,2,false)" afin de compter les nombres pairs. De ces deux noeuds Lustre (le compteur général et le compteur instancié), on en déduit les graphes de flots de données 2.3 et 2.4.

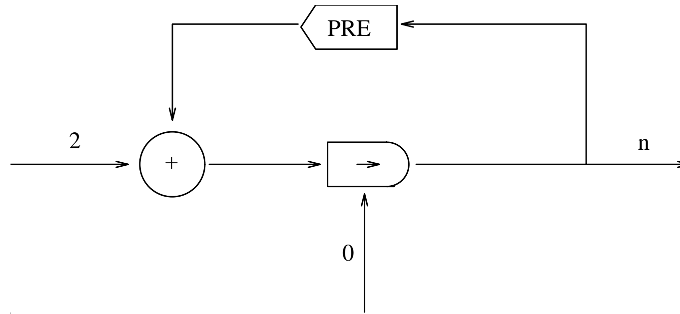

Figure 2.4: Compteur pour pair=COMPTEUR(0,2,FALSE)

Le dernier point que nous aborderons sur Lustre concerne ses capacités de preuves formelles. Comme pour tous les langages synchrones que nous allons voir ici, Lustre offre des possibilités de certification logique et temporelle des programmes. Les propriétés à vérifier sont exprimées grâce aux assertions qui font partie du programme source. A la compilation, on peut choisir d'obtenir un fichier constitué d'automates finis : le format OC[5]. Ce format est utilisé par plusieurs outils de preuves et est commun à d'autres langages synchrones tel qu'Esterel. Nous détaillerons cet aspect dans la fin de ce chapitre.

2.2.3 Le langage Signal

Signal[21, 22] est aussi un langage basé sur les flots de données. Il est assez proche de Lustre. Comme Lustre, Signal introduit la notion de variable (appelée ici "signal") qui est constitué d'une suite de valeurs d'un type donné indexées par le temps. Un signal peut être présent ou absent (l'absence de valeur à un instant t est notée $\perp$ et peut être utilisée comme outil de synchronisation entre deux signaux).

Signal introduit aussi la notion de processus. En fait, un processus Signal est un ensemble d'équations. Toute équation est un processus. Signal fournit une instruction permettant d'effectuer la composition de plusieurs processus en un nouveau. Les programmes Signal sont donc des ensembles d'équations regroupées en processus auxquels on ajoute des contraintes. Celles-ci servent à exprimer les propriétés que l'on souhaite vérifier. Toutes ces équations respectent bien sûr l'hypothèse du synchronisme fort (temps de calcul infiniment court). Signal classifie les processus en deux classes :

- Les processus monochrones : c'est le cas des processus qui ne contiennent

[5] OC pour **O**bject **C**ode.

qu'une seule horloge[6].

1. Statique : tous les signaux d'entrées doivent être présents simultanément,
2. Dynamique : à un instant t, on prend en considération des valeurs du passé (grâce à l'instruction "*delay*" que nous verrons ci-dessous).

- Les processus polychrones : processus avec plusieurs horloges.

Flots de données						
I	10	5	2	9	12	14
K	0	10	5	2	9	12

Figure 2.5: L'instruction de retard de Signal

Signal fournit cinq instructions de bases qui peuvent être combinées pour créer d'autres instructions plus complexes :

- L'instruction "*delay*" : elle permet de spécifier que l'on souhaite obtenir une valeur du passé. L'exemple 2.5 montre les valeurs obtenues à chaque instant. Ici un retard d'un instant est spécifié, mais on peut bien sûr obtenir un retard plus important. L'instruction correspondante est K=I$1,
- La composition de processus : son symbole est ‖. Ex : (P1 ‖ P2) exécute en parallèle P1 et P2,

$$R_n = \begin{pmatrix} I_{n-1} \\ I_{n-2} \\ I_{n-3} \\ I_{n-4} \end{pmatrix} \tag{2.1}$$

- La fenêtre glissante : c'est l'instruction "*window*". Cette instruction permet de définir un vecteur colonne comprenant n valeurs avec un retard croissant. Ainsi l'instruction R:=I *window* 4 créera le vecteur R_n de l'équation 2.1. Ce vecteur contient quatre valeurs qui sont les valeurs de I avec un retard de 1, 2, 3 et 4 instants.

[6]Deux signaux possèdent la même horloge si les valeurs des deux signaux sont présentes au mêmes instants et ce dans tout leur historique.

Flots de données						
Z	10	5	2	9	12	14
Bool	Vrai	Faux	Vrai	$\perp$	Vrai	Vrai
X	10	$\perp$	2	$\perp$	12	14

Figure 2.6: L'instruction *when* de Signal

Flots de données						
Z	10	5	$\perp$	$\perp$	12	$\perp$
Y	5	8	2	$\perp$	1	14
X	10	5	2	$\perp$	12	14

Figure 2.7: L'instruction *default* de Signal

- L'instruction *when* : X:=Z *when* Bool où X et Z sont des signaux et Bool est un signal booléen. X vaut Z quand Bool est présent et vrai, et vaut $\perp$ sinon. Un exemple est donné dans la figure 2.6.

- L'instruction *default* : X:=Z *default* Y. Cette instruction permet d'effectuer un choix entre deux signaux. X vaudra Z quand Z sera présent et Y, sinon. Un exemple est donné dans la figure 2.7.

- L'instruction *cell* : X:=Y *cell* B où B est un signal booléen. Ici, X vaut Y quand Y est présent. X vaut la dernière valeur de Y quand Y est absent et B est présent et vrai.

- L'instruction *event*. T:= *event* X permet d'obtenir l'horloge de X du signal T.

Les applications en Signal sont souvent réalisées à l'aide d'outils graphiques : le programmeur dessine son graphe flot de données et l'outil génère le code Signal. Un exemple de ce type d'outil est SynDEx[54, 55]. Dans le cas de SynDEx, l'utilisateur dessine en plus un graphe représentant les machines sur lesquelles le programme va s'exécuter. SynDEx calcule alors la répartition optimale du code sur les différentes machines du réseau et permet de donner une approximation des temps de réponse. Les temps de réponse réels peuvent être obtenus facilement car SynDEx permet aussi l'instrumentation du code généré.

2.3 Description du langage Esterel

2.3.1 Syntaxe et sémantique du langage Esterel

* Présentation d'Esterel

Esterel[4, 3] fut conçu par G. Berry et L. Cosserat. Contrairement à Lustre et Signal, Esterel est un langage impératif. Il permet comme les deux langages précédents une gestion aisée de la concurrence. En effet, il contient les constructions nécessaires pour exprimer l'exécution de tâches en parallèle. Le parallélisme d'Esterel est un parallélisme d'expression. Le code généré est un code séquentiel où les actions parallèles sont sérialisées. Cette caractéristique est un avantage dans les environnements temps réel car **il n'y a aucun indéterminisme quant à l'ordonnancement des tâches Esterel, ce qui simplifie la mise au point des programmes.** Ce parallélisme d'expression est un point commun avec Lustre et Signal. Comme les langages synchrones précédents, Esterel peut faire l'objet de preuves formelles car sa compilation permet d'obtenir des automates d'états finis.

Dans ce paragraphe, nous verrons les instructions impératives du langage, puis nous décrirons comment les modules Esterel communiquent entre eux. Enfin nous regarderons comment sont gérés le temps et les tâches dans Esterel.

* Les instructions impératives

Une application Esterel est découpée en modules. Chaque module est constitué d'une partie déclarative (l'interface du module) et d'une partie où sont décrites les instructions impératives et temporelles (implantation du module).

* La partie déclarative d'un programme Esterel

```
var foo in
      suite d'instructions;
end var;
```

Figure 2.8: Déclaration d'une variable en Esterel

Cette partie déclare :

- Les signaux d'entrée et de sortie du module qui sont décrits dans le paragraphe suivant,

Instructions	Commentaires
nothing	ne fait rien (ne prend aucun temps d'exécution)
a;b	composition séquentielle : a et b sont exécutés séquentiellement et dans cet ordre
halt	prend un temps infini : la tâche ne sort jamais de cette instruction
a ‖ b	exécute l'instruction (ou la tâche) a en parallèle avec b
var:=expression	instruction d'affectation
[*a*‖*b*];*c*	les caractères [et] permettent d'imposer des priorités entre les opérateurs et les instructions : ici on exécute a et b en parallèle, puis quand a et b sont terminés, on exécute c.
loop instructions *endloop*	effectue une boucle infinie sur la suite d'instructions
if exp *then* instructions1 *else* instructions2	test conditionnel classique
call p	appel de la procédure p

Tableau 2.2: Instructions impératives d'Esterel

- Les prototypes des tâches du module,
- Les prototypes des procédures et des fonctions,
- Les signaux de fin de tâche,
- Les relations,
- Les types construits par l'utilisateur,
- Les constantes.

REMARQUE :

En Esterel, la déclaration des variables est toujours locale à une suite d'instructions [7](voir exemple de la figure 2.8).

*** Liste des instructions impératives d'un programme Esterel**

[7]Ce mécanisme est identique aux déclarations de variables locales dans CAML[30].

Esterel possède des instructions de base telles que l'affectation, et des instructions de contrôle de flots (bien qu'elles soient moins nombreuses que pour des langages structurés comme Pascal ou C). Les principales instructions sont décrites dans le tableau 2.2.

```
loop
      x:=x+1:
end loop;
```

Figure 2.9: Boucle Esterel illégale

Dans les langages synchrones, un programme ne doit pas "prendre de temps" pour s'exécuter. En d'autres termes, toute instruction doit avoir un temps d'exécution fini . Une boucle décrite comme dans la figure 2.9 est illégale : en effet, cette boucle est une boucle infinie qui ne peut donc pas se terminer pendant une transition de l'automate et sera rejetée lors de la compilation. Une seule instruction peut transgresser cette règle : c'est l'instruction "*halt*" qui prend un temps infini et qui est utilisée pour stopper une application.

* **Les signaux et les capteurs Esterel**

Comme nous l'avons précisé dans notre présentation, les modules Esterel communiquent entre eux par des signaux. Ces signaux leurs permettent aussi de communiquer avec l'environnement extérieur, le système d'exploitation par exemple. Un signal est à la fois un outil de synchronisation et un outil de communication. L'arrivée d'un signal "débloque" l'automate et démarre l'exécution de celui-ci. Un signal peut transporter optionnellement une information. Cette information peut être soit d'un type fourni par Esterel, soit d'un type construit par l'utilisateur avec le langage hôte. Esterel ne possède que peu de types : *integer, string* et *boolean.* Les types construits sont donc souvent utilisés. Chaque signal reçu de l'extérieur est diffusé à toutes les tâches dans le module, toutefois, un signal peut être déclaré localement à une partie du code (voir le tableau 2.3) : il est alors émis et reçu à l'intérieur du module.

Il existe plusieurs types de signaux :

- Les signaux en entrée,
- Les signaux en sortie,
- Les signaux en entrée et en sortie,

Déclarations	Commentaires
input foo(*integer*);	signal en entrée du module véhiculant un entier
input foo;	signal en entrée sans donnée (dit signal pur)
output foo(COMPLEX);	signal en sortie du module transportant une donnée de type défini par l'utilisateur
inoutput foo;	signal en entrée et en sortie
signal foo *in* suite d'instructions; end;	signal local
return foo;	le signal foo est envoyé au module quand la tâche associée à ce signal est terminée
sensor foo(*integer*);	déclaration d'un capteur qui permet la lecture d'un entier

Tableau 2.3: Exemples de déclarations de signaux en Esterel

- Les signaux locaux,
- Les signaux locaux qui permettent de détecter la fin d'exécution d'une tâche,
- Et enfin les capteurs qui sont des signaux particuliers. En effet, on peut consulter leur valeur à n'importe quel moment et ils ne permettent pas de synchronisations.

emit foo(100);

Figure 2.10: Emission sur le signal foo de la valeur 100

```
case
        await SIG1 do instructions:
        await SIG2 do instructions;
        await ...
        await SIGn do instructions;
end case;
```

Figure 2.11: Réception de signaux grâce à une instruction *case/await*

```
wait foo;
```

Figure 2.12: Réception sur le signal foo

```
present S then instructions1 else instructions2;
```

Figure 2.13: Test de présence d'un signal

Des exemples de déclaration de tous ces signaux peuvent être consultés dans le tableau 2.3. On peut émettre un signal grâce à l'instruction "*emit*" (voir l'exemple 2.10), et recevoir grâce à "*await*". La réception de signaux peut prendre deux formes, une forme simple avec un seul "*await*", une forme plus complexe avec l'instruction "*case*". Le "*case*" possède une sémantique identique au PRI ALT d'OCCAM[31] (un exemple de "*case*" est donné dans la figure 2.11, un exemple de "*wait*" dans la figure 2.12). Enfin, Esterel permet de tester la présence d'un signal donné grâce à l'instruction "*present*" et d'exécuter des instructions selon le résultat du test (voir l'exemple 2.13).

* Les instructions temporelles et les tâches

```
task tacheUne (integer) (integer, integer, string)
      % cette tâche envoie comme réponse un entier et a besoin
      % de deux entiers et d'une chaîne comme paramètres d'appel
task tacheDeux () (string, string)
      % cette tâche ne renvoie pas de résultat mais
      % a besoin de deux chaînes pour commencer à s'exécuter
```

Figure 2.14: Déclaration de tâches Esterel

* La gestion des tâches

Pour les communications comme pour la gestion des tâches, Esterel fait abstraction des services que peuvent offrir les couches inférieures (notamment, Esterel ne fait aucune supposition sur les services offerts par le système d'exploitation). Ainsi, quand un développeur veut utiliser une tâche Esterel, il doit réaliser une partie du développement dans le langage hôte. C'est en particulier le cas des fonctions qui permettent de créer, suspendre, réactiver et supprimer une tâche, mais c'est aussi le cas du corps des tâches. L'écriture du corps de la tâche dans le langage hôte permet de faire fonctionner une application avec des tâches écrites dans plusieurs langages différents. Un module central en Esterel prouvé effectue les changements de mode de fonctionnement de l'application, et un ensemble de tâches effectue des calculs de manière asynchrone par rapport au noyau Esterel (voir le chapitre trois traitant du modèle Saturne). Cette approche permet d'obtenir une application dont la partie critique (qui est la partie temps réel dur) est prouvée, et une partie moins critique non prouvée formellement.

Au niveau déclaration, il suffit de donner le prototype de la tâche. Comme

pour les procédures, les tâches reçoivent des paramètres en entrée et en sortie. Il faut spécifier le type des paramètres dans deux ensembles entre parenthèses. Le premier ensemble cite les informations qui seront recopiées au retour de la tâche, le deuxième ensemble, les informations d'appel de la tâche. L'appel d'une tâche est faite par l'instruction "*exec*" (voir l'exemple de déclaration figure 2.14).

*** Les instructions temporelles**

```
do
    inst;
watching SIG1;
```

Figure 2.15: L'instruction *watching*

```
do
    inst;
watching SIG1;
timeout inst2;
```

Figure 2.16: L'instruction *watching* avec une clause *timeout*

```
every 60 SECONDE do
    emit MINUTE;
end every
```

Figure 2.17: L'instruction *every*

```
do
        inst;
        halt;
watching SIG
```

Figure 2.18: L'instruction *upto* simulée par un *watching*

```
do
        inst;
upto SIG
```

Figure 2.19: L'instruction *upto*

Les instructions temporelles d'Esterel sont un des points les plus importants du langage. Elles permettent d'implanter facilement dans les applications des mécanismes de chien de garde et d'exception. Le nombre de ces instructions et leurs utilisations étant très nombreux, nous n'en décrirons que quelques unes. Nous renvoyons le lecteur à [14] pour une description plus détaillée.

Le "*watching*" permet de mettre en place des mécanismes de type chien de garde : dans notre exemple 2.15, l'instruction "inst" est exécutée tant que le signal SIG1 n'est pas arrivé. Si l'instruction "inst" finit avant l'occurrence de SIG1 ou si le signal SIG1 arrive avant la fin de "inst", alors le programme sort du "*watching*". Le mot clef "*timeout*" permet d'exécuter une instruction si SIG1 arrive avant la fin de "inst". Dans notre exemple 2.16, si l'instruction "inst" n'est pas terminée lors de l'arrivée du signal SIG1, alors on exécute l'instruction "inst2".

L'instruction "*every*" permet d'exécuter un bloc d'instructions à chaque réception d'un signal. Dans notre exemple 2.17, à chaque réception de soixante signaux SECONDE, on envoie un signal MINUTE.

L'instruction "*upto*" est similaire à l'instruction "*watching*" mais le bloc instructions ne se termine pas : on pourrait simuler une instruction "*upto*" par l'exemple 2.18. La syntaxe de l'instruction "*upto*" est décrite dans l'exemple 2.19.

2.3.2 Exécution des programmes Esterel

Jusqu'à présent, nous avons décrit les principes d'Esterel, puis la syntaxe et la sémantique de ses instructions les plus simples (pour obtenir une définition de la sémantique d'Esterel plus formelle, on peut consulter [2, 5]). Mais nous n'avons pas

parlé de la manière dont est exécuté un code Esterel.

Comme nous l'avons déjà dit auparavant, Esterel fait une totale abstraction du système d'exploitation sur lequel il va fonctionner. Dans la pratique, une partie du code de l'application devra être écrite au moment de l'implantation. Le compilateur Esterel actuel génère un source (en C, ADA ou en LISP) constituant l'automate Esterel ainsi que les prototypes des fonctions, des procédures et des fonctions d'entrée/sortie des signaux de l'automate. Le programmeur est chargé de compléter le corps de ces fonctions.

Le code qui reste à développer concerne donc :

- La définition des types construits par l'utilisateur et les opérations sur ces types (voir l'exemple de l'additionneur de matrices à la fin de ce chapitre),
- Les primitives utilisées pour créer, suspendre, supprimer des tâches Esterel,
- L'implantation des fonctions, des procédures et des tâches Esterel.
- La communication entre automates. Esterel ne prend pas en compte les techniques qui sont utilisées par deux automates pour s'échanger les signaux, et laisse le choix au programmeur (en utilisant des sockets BSD, l'interface TLI[8], des appels de procédure à distance, des appels de méthode à travers un bus objet, etc),

Pour interfacer la partie générée et la partie à écrire dans le langage hôte, le compilateur Esterel attribue le nom des fonctions avec les règles suivantes : quand un automate de nom EMETTEUR[9] veut émettre un signal de nom INFOS, l'automate Esterel appellera la fonction C correspondante : EMETTEUR_O_INFOS(donnée à envoyer). Le développeur devra donc écrire cette fonction, puis effectuer l'édition des liens avec le source généré par le compilateur Esterel. De la même manière, l'arrivée des signaux est constituée par une fonction C. Si l'automate reçoit un signal INF, l'utilisateur devra activer la fonction EMETTEUR_I_INF() pour signaler l'arrivée de la donnée. Un exemple de production d'un exécutable à base d'un automate Esterel est donné dans la figure 2.20.

Dans le source C que génère le compilateur Esterel, il existe une fonction[10] qui démarre l'exécution d'une transition de l'automate. Ainsi, quand l'utilisateur désire activer un automate, il doit d'abord appeler les fonctions qui permettent de

[8]TLI pour **T**ransport **L**evel **I**nterface.

[9]Le nom de l'automate est donné après le mot clef "*module*", voir exemple dans la partie suivante.

[10]Cette fonction a le nom du module Esterel.

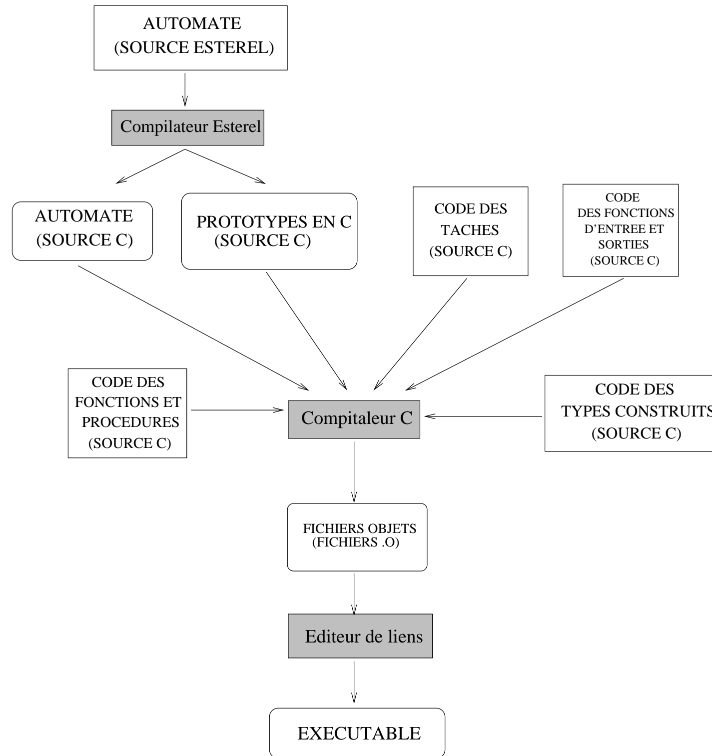

Figure 2.20: Construction d'un exécutable Esterel

signaler l'arrivée des données, puis appeler cette fonction d'activation. Les fonctions d'émission de signaux sont automatiquement exécutées suivant les besoins de l'automate durant la réalisation de sa transition.

Il existe des outils qui permettent de simuler l'exécution de l'automate sans écrire la partie communication (comme l'outil XES[36]). Ces simulations obligent toutefois l'utilisateur à écrire le code concernant les types qu'il a construits et qui sont manipulés par l'automate. Ainsi, les développeurs peuvent tester le code Esterel sans se soucier des problèmes de communication.

2.3.3 L'exemple de l'émetteur-récepteur

Dans cette partie et la suivante, nous allons présenter deux exemples de programmes Esterel qui constituent des "cas d'école" mais qui permettent d'illustrer les outils qui ont été utilisés dans cette étude. Ce premier exemple met en oeuvre deux modules : le module émetteur et le module récepteur. Ces modules manipulent des signaux transportant des entiers. Rappelons que les signaux Esterel peuvent

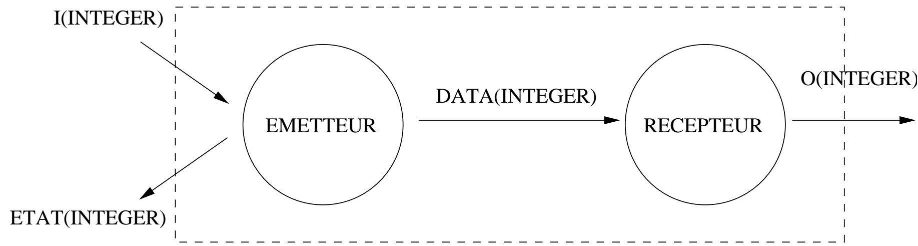

Figure 2.21: Exemple de l'émetteur et du récepteur

véhiculer n'importe quel type d'information (les types offerts par Esterel ou définis par le programmeur) mais peuvent aussi ne transporter aucune information. Dans ce cas, ces signaux réalisent uniquement une synchronisation, ils sont alors appelés "signaux purs". La figure 2.21 nous montre la représentation graphique que nous adopterons dans ce document pour les applications Esterel. Chaque cercle constitue un module. Les arcs orientés définissent les signaux échangés. Le nom du signal est stipulé sur l'arc correspondant. Si le signal véhicule une information, alors son type est donné entre parenthèses après son nom. Le cadre en pointillé permet de délimiter l'application synchrone (qui est constituée de tous les modules Esterel) de l'environnement extérieur.

* Le module émetteur

A chaque activation, ce module reçoit un entier par le signal I. Il réémet cette valeur sur le signal de sortie DATA en direction du module récepteur. La construction syntaxique exprimant qu'à chaque activation, le signal DATA doit être réémis est l'instruction "*every...do*". L'émission du signal ETAT est gérée différemment : l'instruction "*await*" suspend l'automate jusqu'à la prochaine transition. Ainsi, dans le module émetteur, la première transition émet un "1" sur le signal ETAT, puis un "2" sur la transition suivante et ainsi de suite. L'instruction "*loop*" exprime que cette suite d'émission des valeurs 1 et 2 sur le signal ETAT est faite dans une boucle infinie. Le symbole "?I" permet la lecture de la valeur reçue sur le signal I. La figure 2.22 donne le source Esterel de ce module. On remarquera que ce module est constitué de deux entités s'exécutant en parallèle. C'est le signe || qui exprime que les deux instructions *loop* et *every..do* sont parallèles.

* Le module récepteur

Le module récepteur lit un signal d'entrée DATA à chaque transition, puis, émet le signal O. La valeur émise sur O est la valeur reçue sur le signal d'entrée DATA plus la valeur du signal DATA à la transition précédente. La figure 2.23 donne le source Esterel de ce module.

```
module emetteur:

input I (integer);
output ETAT (integer);
output DATA (integer);

every I do emit DATA(?I);
end every
||
loop
	await I;
	emit ETAT(1);
	await I;
	emit ETAT(2);
end loop
end module
```

Figure 2.22: Source Esterel de l'émetteur

* Comment exécuter cet exemple

Nous avons dit dans les parties précédentes que le code généré par le compilateur Esterel n'était pas directement exécutable. Nous allons présenter ici les programmes qui peuvent être utilisés pour faire fonctionner l'exemple ci-dessus. Le langage hôte est le C. L'ensemble des sources C nécessaires se trouve en annexe A. Les modules émetteur et récepteur sont implantés par deux processus Unix. Ils utilisent des files de messages pour communiquer le signal DATA. L'annexe A comprend les trois fonctions qui émettent les signaux de sorties ETAT, O et DATA. Pour le module émetteur, le compilateur Esterel génère les prototypes : EMETTEUR_*O*_ETAT(*integer*), EMETTEUR_*O*_DATA(*integer*), EMETTEUR_*I*_I(*integer*), et pour le module récepteur : RECEPTEUR_*O*_O(*integer*), RECEPTEUR_*I*_DATA(*integer*). Les trois fonctions de sortie prennent en argument un entier qui correspond à la valeur du signal. Elles sont automatiquement appelées par l'automate lorsqu'il a besoin de transmettre un signal en sortie. Le corps de la fonction EMETTEUR_*O*_DATA(*integer*) est une écriture dans une file d'attente Unix afin de pouvoir communiquer l'entier fourni par le module émetteur au module récepteur. Les deux autres fonctions de sortie sont de simples affichages à l'écran. Le programme principal est le fichier *main.c*. Après les initialisations, il appelle une fois chaque automate grâce aux fonctions *emetteur*() et *recepteur*(). Ce premier appel permet de les initialiser. Dans le fichier *main.c*, on teste le contenu de la file d'attente, si elle contient une donnée,

```
module recepteur:

input DATA(integer);
output O (integer);

var LAST:= 0: integer in
every DATA do
       emit O (?DATA+LAST);
       LAST:=?DATA;
end every
end var
end module
```

Figure 2.23: Source Esterel du récepteur

on appelle la fonction RECEPTEUR_I_DATA(*integer*) en lui passant comme argument la valeur de la donnée. Les fonctions d'entrée sont générées par le compilateur Esterel. Une fois les fonctions d'entrée exécutées, l'automate peut être invoqué de nouveau grâce à *emetteur*() ou *recepteur*().

2.3.4 L'exemple de l'additionneur de matrices

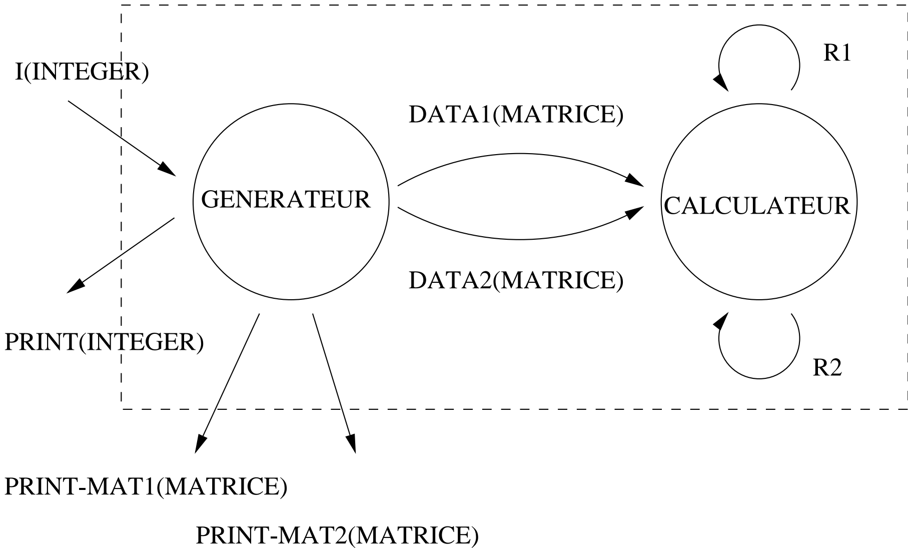

Figure 2.24: Exemple de l'additionneur de matrices

Nous décrivons ici notre deuxième exemple d'application Esterel. Cet exemple nous permet d'illustrer le fonctionnement des tâches sous Esterel. Ce système est constitué de deux modules qui sont "générateur" et "calculateur". Le premier des deux tire aléatoirement deux matrices d'entiers qu'il affiche grâce aux deux signaux PRINT-MAT1 et PRINT-MAT2, le second fait la somme de celles-ci, puis affiche le résultat. On voit ici comment on peut écrire en Esterel une application qui manipule un type construit par l'utilisateur sans se préoccuper, dans un premier temps, de l'implantation du type et des opérations que l'on pourra effectuer sur celui-ci. Dans cet exemple, c'est le cas du type MATRICE où dans le source Esterel on ne spécifie pas le contenu des tâches startComputeMatrice et startAffichMatrice.

* **Le module générateur**

Dans ce module, on utilise les mêmes constructions que l'on avait utilisées dans l'exemple de l'émetteur/récepteur. On introduit ici deux notions supplémentaires importantes en Esterel : l'expression de la concurrence et l'utilisation de procédures et fonctions :

- L'instruction || permet d'exprimer le parallélisme entre des blocs d'instructions séquentielles délimités par les caractères [et]. C'est un parallélisme d'expression. Les différents blocs sont sérialisés à la compilation. L'exécution d'un automate Esterel est séquentielle,
- Esterel permet de définir des procédures et des fonctions. Les prototypes de ces procédures et de ces fonctions sont déclarés dans l'interface du module Esterel et c'est au programmeur de réaliser l'implantation dans le langage hôte (ici le langage C).

* **Le module calculateur**

Ce dernier module s'occupe de l'addition des deux matrices. Celui-ci attend l'arrivée des signaux DATA1 et DATA2 (qu'il reçoit en parallèle), puis démarre une tâche startComputeMatrice qui additionne le contenu de DATA1 et DATA2 pour mettre le résultat dans "result". Ce résultat est alors affiché par une autre tâche qui est startAffichMatrice. Dans ce module, on a déclaré deux signaux internes un peu particuliers : ces signaux sont déclarés par l'instruction "*return*". Ils doivent être envoyés par la tâche pour signaler à l'automate que celle-ci vient de se terminer. Bien que la déclaration de ces signaux soit différente, ceux-ci sont gérés de la même manière que des signaux normaux, et en particulier, la technique décrite dans le paragraphe "Exécution de programme Esterel" est aussi celle utilisée pour les émettre depuis le langage hôte.

```
module generateur:

type MATRICE;

input I(integer);
output DATA1 (MATRICE);
output DATA2 (MATRICE);
output PRINT (integer);
output PRINT-MAT1(MATRICE);
output PRINT-MAT2(MATRICE);

function matriceAleatoire() : MATRICE;

loop
      await I;
      emit PRINT(?I);
end loop
||
every I do
      [
      var matrice : MATRICE in
            matrice:=matriceAleatoire();
            emit DATA1(matrice);
            emit PRINT-MAT1(matrice);
      end var
      ]
      ||
      [
      var matrice : MATRICE in
            matrice:=matriceAleatoire();
            emit DATA2(matrice);
            emit PRINT-MAT2(matrice);
      end var
      ]
end every
end module
```

Figure 2.25: Source Esterel du générateur de matrices

```
module calculateur:

type MATRICE;

return R1;
return R2;

task startComputeMatrice (MATRICE) (MATRICE,MATRICE);
task startAffichMatrice () (MATRICE);

input DATA2(MATRICE);
input DATA1(MATRICE);

loop
    var result : MATRICE in
        [
        await DATA1;
        ||
        await DATA2;
        ]
        exec startComputeMatrice (result) (?DATA1,?DATA2) return R1;
        exec startAffichMatrice () (result) return R2;
    end var
end loop
end module
```

Figure 2.26: Source Esterel de l'additionneur de matrices

2.4 Validation de programmes synchrones

Bien que la validation de programmes Esterel ne fasse pas partie de cette étude, nous présenterons succinctement dans ce paragraphe la manière dont les preuves de programmes Esterel peuvent être conduites. Nous donnerons deux exemples d'outils utilisant chacun un fichier différent obtenu après compilation d'un programme Esterel : les fichiers OC[11] et les fichiers BLIF[12].

Un fichier OC contient les états d'un automate d'états finis[65, 44] correspondant à un programme Esterel. Lorsqu'un utilisateur souhaite vérifier un certain nombre de propriétés, il doit d'abord utiliser le compilateur Esterel pour obtenir le fichier OC de son application, puis faire appel à un outil dit "model-checker". TempEst[39] est un "model-checker" qui utilise un fichier OC en entrée. Il est basé sur une logique temporelle linéaire[13]. L'utilisateur spécifie en logique temporelle les propriétés de sécurité[45] qu'il souhaite vérifier (TempEst ne gère que les propriétés de sécurité et non celles de vivacité. Une propriété de vivacité ou "liveness" exprime intuitivement *"qu'une bonne chose arrivera nécessairement"*. Les propriétés de sécurité ou "safety" expriment pour leur part *"que de mauvaises choses ne seront jamais vérifiées"*, et ce quelles que soient les exécutions du programme). Cette spécification est traduite automatiquement par TempEst en un programme Esterel qui émet un signal quand une des propriétés de la spécification est violée. Le programme de l'utilisateur et celui généré par TempEst sont ensuite mis en parallèle. TempEst applique ensuite sur ce nouveau programme une technique de "model-checking" qui consiste à parcourir tous les états de l'automate afin de déterminer si l'un d'eux conduit à l'émission d'un signal détectant la violation d'une propriété. Cette technique de "model checking" est souvent appelée technique de "l'observateur".

Un fichier BLIF décrit l'automate d'une manière implicite (c'est à dire sans énumérer explicitement les états) à l'aide d'un système d'équations booléennes. Ce format est utilisé par l'outil CHECKBLIF[10]. Le fonctionnement de cet outil est proche de celui de TempEst. L'utilisateur écrit un programme observateur en Esterel qui émettra un signal lorsque la propriété sera violée. Il compile son programme Esterel avec l'observateur afin d'obtenir le fichier BLIF. Le fichier BLIF ainsi que le nom du signal détectant la violation de la propriété sont donnés en entrée de CHECKBLIF. CHECKBLIF recherche alors si le signal est émis dans l'espace des états de l'automate et, si c'est le cas, donne la suite des transitions pour arriver sur l'état fautif. Ce chemin peut alors être visionné par un outil comme XES[36]. Sur de grands programmes Esterel, le nombre d'états est très important. L'utilisation des

[11]OC Pour **O**bject **C**ode.

[12]BLIF pour **B**erkeley **L**ogical **I**nterchange **F**ormat.

fichiers OC est alors impossible. Le format BLIF à l'inverse, autorise les traitements sur de gros programmes. En effet, le format BLIF permet l'utilisation d'une représentation de l'espace des états qui économise de la place mémoire : les BDD[13][12]). Les BDD décrivent implicitement les états plutôt que de les énumérer. D'autres outils de preuves utilisent les fichiers BLIF et les BDD. Une partie du projet MEIJE[25] de l'INRIA[14] a pour objectif l'étude des environnements de vérification et d'analyse de systèmes communicants. Une équipe de ce projet travaille sur un outil de preuves nommé FCTOOLS[18] qui utilisent les BDD et les fichiers BLIF.

2.5 Conclusion

Esterel n'est pas seulement un langage de programmation. C'est un langage permettant d'effectuer des spécifications, de les prouver, et d'obtenir un prototype permettant de faire une première évaluation de l'application finale. Ce langage synchrone peut être associé à des méthodes de conception s'appuyant sur des représentations sous formes d'automates d'états finis. Nous avons vu que plusieurs outils de preuves existaient pour Esterel. Un certain nombre de ces outils ont été récemment associés pour construire un environnement complet de développement d'applications synchrones : la plate-forme SPORTS[15][36]. SPORTS rassemble, en plus des outils de preuves, des éditeurs graphiques ou textuels, des générateurs de code et des outils de mise au point et de simulation. Néanmoins, dans le cadre d'applications embarquées, l'utilisation directe du code généré par le compilateur Esterel reste aujourd'hui encore difficile. Pour des programmes complexes où le nombre d'états des automates est important, la taille et les performances du code généré ne sont pas compatibles avec les contraintes des logiciels embarqués. Cependant, les travaux en cours sur l'optimisation d'Esterel permettent d'espérer que ces difficultés seront bientôt levées.

Dans ce chapitre, nous avons présenté le modèle synchrone fort, puis nous avons décrit le langage Esterel. Le modèle synchrone fort a été conçu pour un environnement centralisé, dans le chapitre suivant, nous allons exposer le modèle Saturne qui permet une exécution répartie de noyaux synchrones.

[13]BDD pour **B**inary **D**ecision **D**iagram.

[14]INRIA pour **I**nstitut **N**ational de **R**echerche en **I**nformatique et en **A**utomatique.

[15]SPORTS pour **S**ynchronous **P**rogramming **Of** **R**eal **T**imes **S**ystems.

Chapitre 3

Le modèle Saturne

3.1 Le modèle synchrone faible

Dans le chapitre deux, nous avons décrit le modèle synchrone fort qui permet de spécifier des applications temps réel en bénéficiant à la fois d'un parallélisme d'expression, d'un comportement déterministe durant l'exécution et de possibilité de preuve logique et temporelle. Toutefois, ce modèle ne prend pas en compte les temps de communication. En effet, il a été conçu pour des systèmes centralisés, et non pour des systèmes distribués. Dans un environnement distribué, les temps de communication ne sont pas négligeables et ne permettent plus de supposer que les temps d'exécution et de communication d'un noyau soient nuls. Un autre modèle doit donc être utilisé. Il doit permettre de prendre en compte la répartition d'une application tout en conservant les propriétés prouvées dans le modèle synchrone fort. Il doit fournir une réactivité suffisante pour les applications temps réel, et rester aisément adaptable à une évolution de l'environnement d'exécution. Le modèle qui peut être utilisé ici est le modèle synchrone faible[1].

*** Définition du modèle synchrone faible:**

Dans ce modèle, il existe une horloge commune à tous les noyaux. Les noyaux ne réagissent plus par réflexe, mais périodiquement. L'horloge est une variable globale du système et les noyaux sont tous activés en même temps sur les tops de cette horloge globale. La notion d'instant d'activation s'apparente alors à cette activation périodique et non plus au moment où les signaux sont présents à l'entrée du noyau comme c'était le cas dans le modèle synchrone fort. Cet instant logique permet de limiter les temps de communication des signaux. **En d'autre termes, le réseau de communication doit être capable de garantir qu'une communication soit d'une durée bornée.** Cette borne fait alors partie de l'intervalle entre deux

tops de l'horloge globale : un noyau synchrone prend un intervalle de temps pour ses calculs et ses communications. On échange la notion de temps de calcul/temps de communication nul qui était spécifié dans le modèle synchrone fort par un temps de calcul/communication égal à **un instant logique de durée constante.** Cet intervalle de temps définit la période de l'horloge globale du système.

3.2 Le modèle Saturne

3.2.1 L'architecture de Saturne

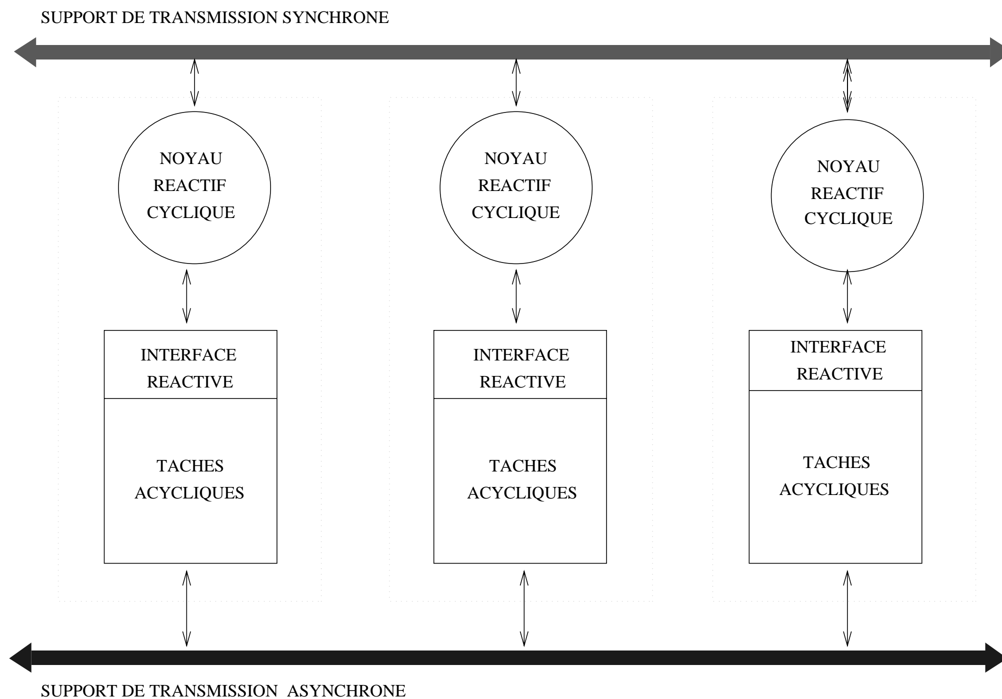

Figure 3.1: L'architecture Saturne

L'architecture du modèle Saturne[9, 8] correspond à un couplage entre modèle synchrone faible et modèle synchrone fort. Ce couplage permet de bénéficier de tous les avantages des deux modèles. Il confère à chaque noyau réactif un comportement externe périodique, et un temps de réaction de durée constante (donc compatible avec le modèle synchrone faible), tandis que son comportement interne est du type réflexe instantané (donc synchrone fort). Saturne est constitué de grappes connec-

tées entre elles par deux réseaux comme le montre la figure 3.1. Chaque grappe est constituée par :

- Un noyau synchrone : c'est la partie réactive de l'application. Elle est constituée d'un automate Esterel. Le noyau synchrone suit le modèle synchrone fort et effectue les changements de mode de fonctionnement de l'application,
- Une partie constituée de l'interface réactive et des tâches transformationnelles. Les tâches transformationnelles sont pilotées par le noyau synchrone. Toutefois, les noyaux synchrones ne commandent pas directement celles-ci, c'est l'interface réactive qui exécute les commandes du noyau synchrone. Les tâches transformationnelle constituent la partie calcul du système.

* **Notion de tâche interruptible**

Le modèle Saturne distingue deux types de tâches transformationnelles : des tâches conventionnelles et celles qui sont dites "interruptibles". Les tâches conventionnelles reçoivent des données en entrée et ne peuvent fournir un résultat **qu'à la fin de leur exécution.** Par notion de tâches interruptibles, on entend une tâche qui a la possibilité de fournir un résultat d'une qualité donnée à n'importe quel moment de son exécution. Ces tâches sont aussi appelée "*anytime*"[48]. En général, elles sont constituées de traitements itératifs qui fournissent un résultat de qualité croissante au fur et à mesure de leur exécution. C'est le cas des algorithmes d'optimisation que l'on peut trouver en recherche opérationnelle ou en intelligence artificielle. Une des difficultés de ce concept est de spécifier la qualité du résultat[1]. Par contre, l'avantage de ce type de tâche dans les applications temps réel est qu'elles fournissent une solution permettant de respecter les échéances de celles-ci. On peut ainsi concevoir l'utilisation dans des systèmes temps réel d'algorithmes dont la terminaison n'est pas bornée. **Toutefois, il faut noter que dans les applications d'avionique, l'utilisation des tâches transformationnelles est peu fréquente et que les tâches conventionnelles prédominent.**

* **Les primitives de Saturne**

Les tâches transformationnelles étant contrôlées par le noyau synchrone du modèle Saturne, les commandes Saturne sont donc une extension du langage Esterel. En effet, les noyaux synchrones sont écrits en Esterel pour respecter le modèle synchrone fort. Les primitives Saturne sont au nombre de six :

- START_Ti [j] (paramètres en entrée) : activation de la tâche Ti avec les

[1]Quelle qualité de résultat doit on absolument obtenir une fois la tâche interrompue ? Cette qualité sera t'elle suffisante ?

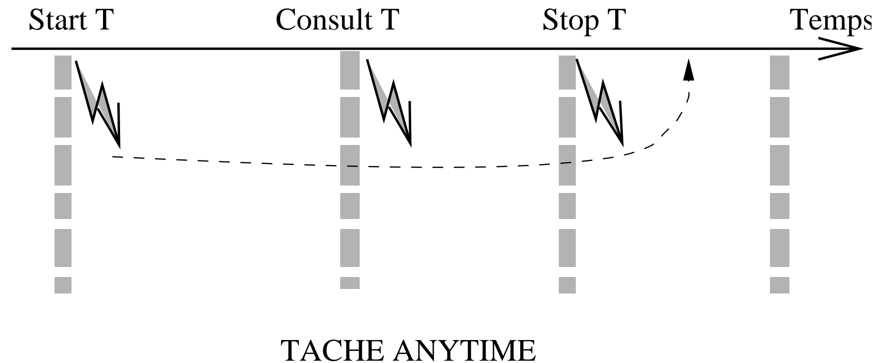

Figure 3.2: Utilisation des primitives de Saturne

paramètres d'entrée. L'indice j est un numéro d'exemplaire : en effet, on peut imaginer que le noyau réactif ait besoin de démarrer plusieurs fois une même tâche avec des données différentes,

- KILL_Ti [j] : arrêt de l'exemplaire j de la tâche Ti sans récupération des résultats,

- CONSULT_Ti [j] : consultation des résultats courants de l'exemplaire j de la tâche Ti,

- STOP_Ti [j] : consultation des résultats courants de l'exemplaire j de la tâche Ti, puis arrêt de celle-ci,

- SUSPEND_Ti [j] : suspension de l'exemplaire j de la tâche Ti,

- RESUME_Ti [j] : réactivation de l'exemplaire j de la tâche Ti.

Sur la figure 3.2, on peut voir un exemple d'utilisation des primitives Saturne. Cette figure nous donne un chronogramme de l'exécution d'une tâche. Les tops d'horloge sont représentés par les barres verticales et l'envoi des commandes du noyau synchrone vers la tâche par les éclairs. La courbe horizontale en pointillé montre l'exécution de la tâche. Celle-ci est lancée par une commande START au premier top puis, à chaque activation du noyau synchrone, le noyau consulte le résultat intermédiaire produit par la tâche transformationnelle. Au troisième top, le noyau décide d'arrêter la tâche et de récupérer le dernier résultat calculé. Cette action est réalisée par une commande STOP. L'arrêt de la tâche intervient avant le quatrième top.

* **Les réseaux de communication**

Le modèle Saturne génère deux modes de communication différents représentés sur la figure 3.1 par les deux flèches horizontales :

- Un mode synchrone pour les échanges entre noyaux réactifs. Dans ce mode, la taille des signaux est faible mais leur délai d'acheminement est très contraint car tous les signaux émis dans l'instant t doivent être reçus à l'instant $t+1$,

- Un mode asynchrone pour les échanges entre tâches transformationnelles. Dans ce mode les messages échangés peuvent être de grande taille, mais les temps de communication sont moins critiques.

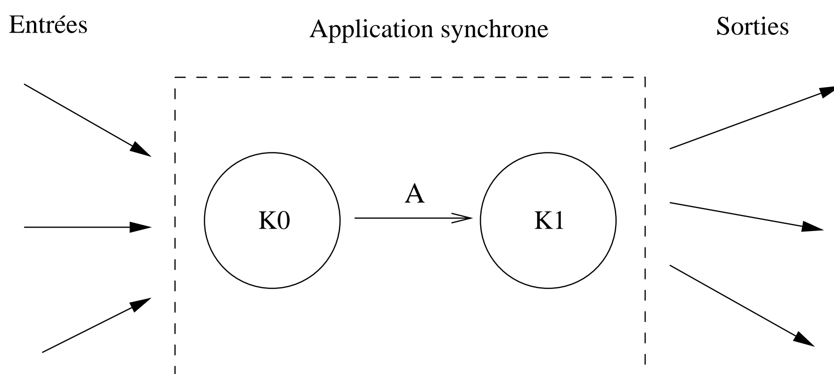

Figure 3.3: Le problème "50Hz-80Hz"

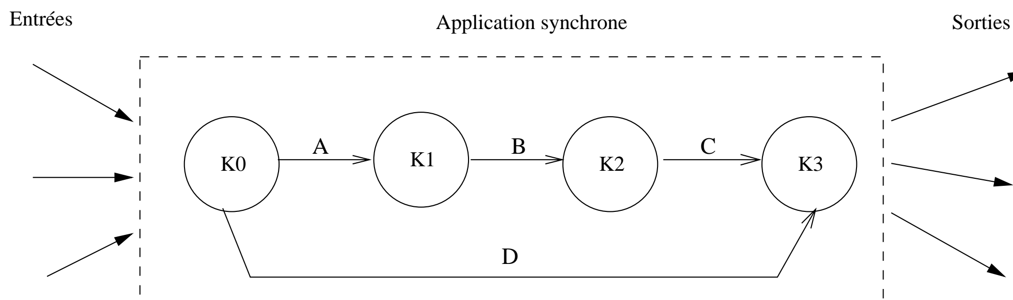

Figure 3.4: Le problème du "court-circuit"

3.3 Extensions du modèle Saturne

Le modèle Saturne décrit ci-dessus a été conçu entre 1992 et 1994. Depuis, plusieurs études ont été réalisées sur ce sujet. En collaboration avec Dassault Aviation, le CERT a fait évoluer le modèle Saturne pour l'adapter aux problèmes spécifiques des applications avioniques embarquées. Nous présentons dans cette partie les problèmes posés par le modèle Saturne initial pour la modélisation d'applications embarquées, puis nous décrivons les extensions proposées par le CERT.

3.3.1 Comportements non modélisation avec Saturne

L'objectif initial de Saturne est la spécification d'applications temps réel embarquées. Or, un certain nombre de comportements courants dans ces systèmes ne sont pas directement modélisables avec la version initiale de Saturne. Voici deux exemples qui ont été détaillés dans un récent rapport du CERT[47] : le premier concerne la synchronisation de deux équipements qui sont activés à des fréquences différentes. Ce problème a été baptisé le problème "50Hz-80Hz". Dans les systèmes embarqués des avions, il existe de nombreux cas où deux équipements ont besoin de coopérer alors qu'ils ont des fréquences d'activations très différentes. La figure 3.3 nous montre deux noyaux synchrones. Le noyau K0 activé à 50Hz, envoie le signal A au noyau K1 qui est activé à 80Hz. Une telle application n'est pas modélisable en Saturne puisque Saturne active tous les noyaux en même temps grâce à son horloge globale. Le deuxième exemple est le problème du "court-circuit" signalé par E. Nassor et Y. Le Biannic, ingénieurs chez Dassault Aviation. La figure 3.4 nous montre quatre noyaux synchrones. Le noyau K0 émet deux signaux : A et D. Le noyau K3 reçoit les signaux C et D. La difficulté ici est de synchroniser l'arrivée des signaux C et D sur K3. Avec la première version de Saturne, on voit bien que si K0 émet A et D à l'instant t, D arrivera sur K3 en t+1 et C en t+3. Il faut donc pouvoir retarder suffisamment le signal D pour qu'il arrive sur K3 en même temps que C. Ce que le modèle Saturne initial n'est pas capable de faire.

3.3.2 Les extensions proposées : Saturne multi-synchrone

Pour résoudre les problèmes ci-dessus, le CERT a donc conçu un nouveau modèle : le modèle Saturne multi-synchrone[47] (nous parlerons de Saturne mono-synchrone pour le modèle Saturne sans les extensions décrites dans cette partie. **Il faut aussi préciser que les chapitres suivants de ce document concernent la version mono-synchrone de Saturne**).

Ce nouveau modèle reste assez proche de Saturne mono-synchrone. On y retrouve les grappes, les noyaux synchrones , les tâches transformationnelles, les interfaces

réactives ainsi que les deux supports de communication. Les innovations concernent l'horloge globale de Saturne, les activations des noyaux ainsi que la communication des signaux Esterel. On y introduit de nouvelles abstractions :

- A chaque noyau synchrone on associe une horloge locale d'activation,
- On définit une horloge de référence. Cette horloge correspond à l'horloge globale de Saturne mono-synchrone. La fréquence de l'horloge globale est égale au plus petit commun multiple des fréquences de toutes les horloges locales des noyaux. Dans le modèle multi-synchrone, l'horloge de référence est toujours diffusée à tous les noyaux. Mais ceux-ci s'activent seulement quand le top de l'horloge de référence reçu correspond à un top de leur horloge locale. Cette correspondance peut se réaliser facilement par une opération de modulo entre l'horloge de référence et l'horloge locale. Ce nouveau mécanisme permet de modéliser des comportements de type "50Hz-80Hz",

$$\begin{cases} L_{recep,K1,A} = L_{recep,K2,B} = L_{recep,K3,C} = L_{recep,K3,D} = 0 \\ L_{emis,K0,A} = 0 \\ L_{emis,K0,D} = 2 \\ L_{emis,K1,B} = 0 \\ L_{emis,K2,C} = 0 \end{cases} \tag{3.1}$$

- On introduit enfin la notion de latence. Les temps de latence sont des retards qui sont appliqués à l'émission ou à la réception des signaux entre les noyaux. On définit donc une latence par signal pour chaque noyau synchrone. Nous noterons ici L_{recep} (respectivement L_{emis}) le temps de latence pour la réception (respectivement pour l'émission) d'un signal. Ainsi, dans notre figure 3.4, le temps de latence en émission du signal A par le noyau K0 est noté $L_{emis,K0,A}$. Les temps de latence vont nous permettre de résoudre des problèmes de type "court-circuit". Le système d'équation 3.1 nous donne les valeurs des différentes latences pour que l'arrivée de C et D sur le noyau K3 soient synchronisées. Ici, on a choisi de positionner toutes les latences de réception à zéro. La synchronisation de C et D s'effectue sur les émissions : on positionne le temps d'émission de D à deux et tous les autres à zéro. En dehors du coupe-circuit, les temps de latence ont de nombreuses autres applications possibles, mais ils ont aussi quelques inconvénients. L'utilisation de plusieurs horloges et des latences complique singulièrement l'exécution d'une application Saturne. Il devient difficile d'en comprendre le fonctionnement. De plus, sur de très grosses applications, la détermination des latences devient

vite compliquée. L'utilisation des latences nécessitera très certainement la réalisation d'outils permettant de positionner automatiquement des latences, ou au moins, de vérifier que leurs valeurs soient correctes par rapport à une spécification de la synchronisation voulue.

3.4 Le modèle Saturne Objet

Les systèmes embarqués sont compliqués et de taille importante. Il est donc nécessaire de pouvoir y associer des outils et des méthodologies. Le monde objet est riche dans ce domaine et un certain nombre de travaux ont tenté d'associer le monde synchrone au monde objet. F. Boulanger dans sa thèse[11] a intégré des noyaux synchrones dans des classes C++. Ainsi, il est possible d'appliquer des opérations issues du monde des objets sur des "objets synchrones", comme l'héritage, et surtout l'instanciation dynamique. Les travaux de Boulanger ont donné lieu, entre autres, à une mise en oeuvre d'objets synchrones sur une plate-forme temps réel VxWorks. Les objets synchrones furent par la suite utilisés pour la définition d'un modèle Saturne objet par Faure[19], que nous allons maintenant rapidement décrire.

3.4.1 Description du modèle Saturne Objet

Le modèle Saturne Objet diffère du modèle Saturne initial principalement par deux aspects :

- La notion de grappe disparaît. Avec le modèle Saturne initial, une tâche était pilotée par un noyau et un seul. Toutes les tâches qui appartenaient à un noyau étaient regroupées ensemble. Ici, toutes les tâches utilisant des données communes sont regroupées sur un même site. Les contrôleurs de site assurent la cohérence des données communes. Enfin, l'interface réactive est remplacée par un contrôleur de tâche. Chaque tâche est pilotée par un contrôleur de tâche (voir notre figure 3.5). Ce nouveau regroupement des tâches implique que les noyaux peuvent piloter des tâches de n'importe quel site. Toutefois, une tâche est toujours contrôlée par un noyau et un seul,
- Le système devient dynamique. Les composants de Saturne Objet sont pour certains, réalisés sous formes d'objets synchrones. Ils peuvent donc être instanciés durant l'exécution. C'est le cas des noyaux synchrones, des contrôleurs de site, des tâches et de leur contrôleur.

Il est difficile de dire si l'aspect dynamique de Saturne Objet est compatible avec des applications temps réel dur : ce modèle a été implanté dans le monde Unix, mais pas sur une plate-forme temps réel.

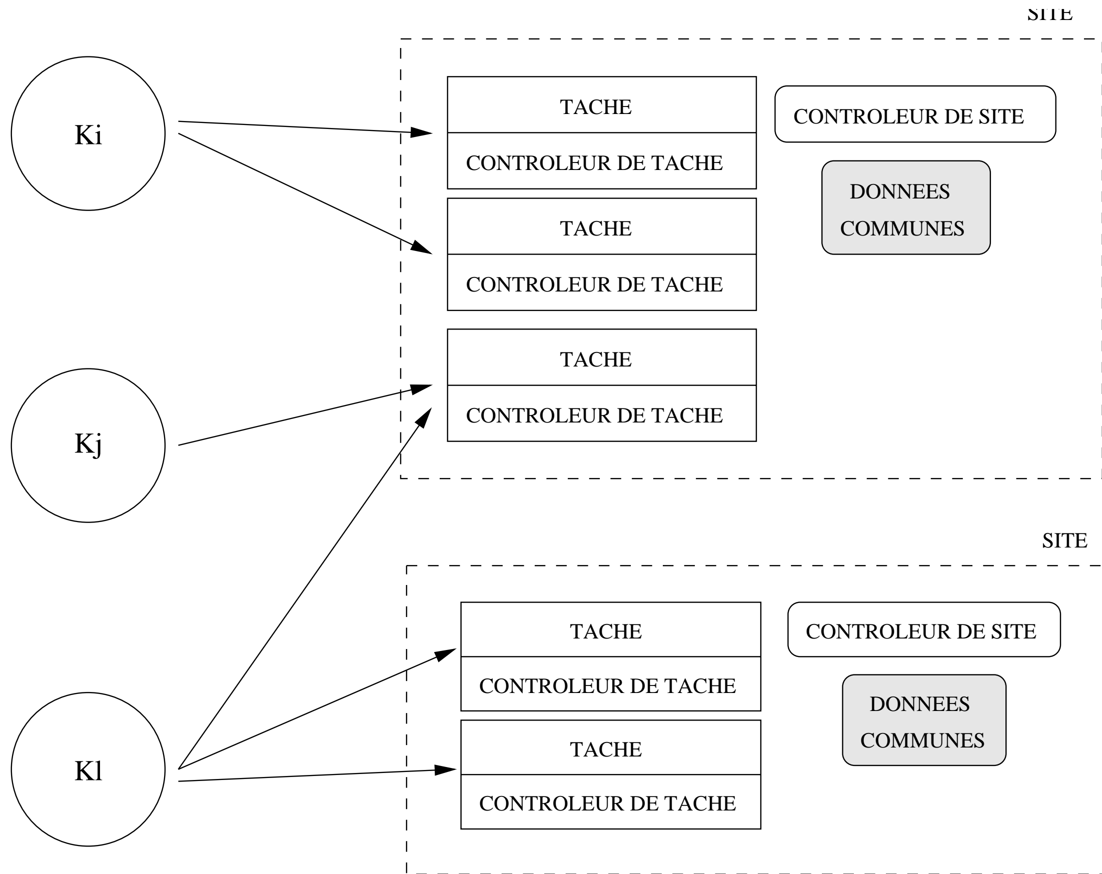

Figure 3.5: Le modèle Saturne Objet

3.5 Contraintes apportées par le modèle Saturne sur l'environnement d'exécution

A partir des paragraphes précédents, on voit rapidement les contraintes qu'apporte le modèle Saturne mono-synchrone sur l'environnement d'exécution. Le système d'exploitation sur lequel Saturne peut fonctionner doit permettre :

- De supporter deux modes de communication par processeur **dont un qui soit synchrone**. Ceci est nécessaire pour respecter les contraintes du modèle synchrone faible. Bien que ceci ne soit pas standard dans CHORUS et que cette étude ne fasse pas partie de ce mémoire, certains travaux, comme ceux réalisés au CNET[2][29, 56], ont démontrés qu'il était possible de réaliser ce travail.

[2] CNET pour **C**entre **N**ational d'**E**tudes en **T**élécommunications.

Dans ce projet, une interface FDDI[3] cohabitait avec une interface Ethernet. Le coupleur FDDI supportait le trafic synchrone tandis que le coupleur Ethernet permettait le partage de fichiers,

- De garantir des temps d'exécution bornés et prédictibles. CHORUS étant un système d'exploitation temps réel, cette contrainte devrait être satisfaite,

- Bien que ca ne fasse pas partie de la spécification du modèle Saturne, il est impossible d'ignorer les contraintes de tolérance aux pannes : en effet, cette propriété est primordiale dans les applications temps réel qui seront développées sur Saturne. CHORUS permet la mise en place de ce type de mécanisme grâce, entre autre, à sa transparence à la localisation et à sa notion de groupe[4]. Cet aspect sera traité dans [53].

- Enfin, de résoudre les problèmes d'ordonnancement et de fournir des mécanismes de préemption afin d'assurer un niveau de réactivité suffisant. Nous verrons dans le chapitre cinq, comment ceci peut être obtenu avec CHORUS. Toutefois, avant de voir de quelles manières nous concevons le modèle Saturne sur CHORUS/COOL, nous allons dans le prochain chapitre décrire la plate-forme d'exécution choisie.

[3] FDDI pour **F**iber **D**ata **D**itributed **I**nterface.

[4] Bien que celle-ci soit relativement pauvre comparé à des outils comme Isis[15, 16] ou Horus[66, 50].

Chapitre 4

Le sous-système à objets de CHORUS : COOL V3

Dans ce chapitre, nous allons décrire COOL V3[63, 62]. Toutefois, avant de rappeler le standard CORBA 2 sur lequel COOL est basé, nous allons rapidement décrire les aspects principaux de l'environnement sous lequel nous avons utilisé celui-ci : le micro-noyau CHORUS[59, 60] ainsi que le sous-système ClassiX[61].

4.1 Le système d'exploitation CHORUS

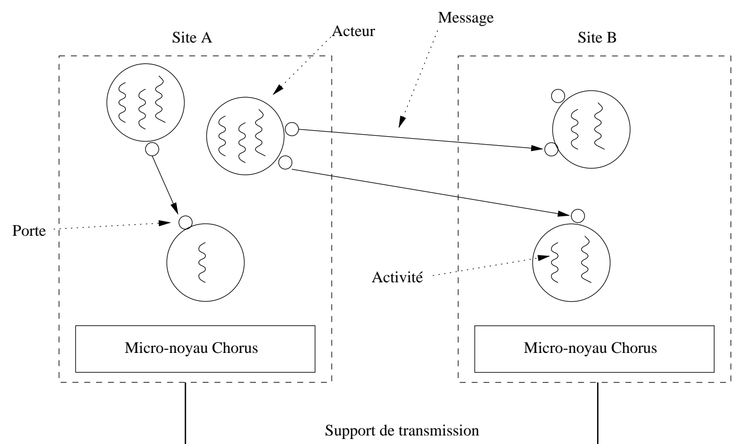

Figure 4.1: Les abstractions CHORUS

CHORUS est un système distribué. Il intègre une approche micro-noyau et a pour objectif de fournir des outils pour concevoir des applications temps réel distribuées sur un ensemble de machines connectées par un réseau ou un bus. Une de ses grandes

caractéristiques est qu'il offre la possibilité de faire cohabiter des applications temps réel avec d'autres applications non critiques. Dans cette partie, nous donnerons une rapide description de CHORUS et du sous-système nécessaire au fonctionnement de COOL.

4.1.1 Les abstractions de CHORUS

Le système CHORUS[52, 59, 60] introduit un certain nombre d'abstractions qui décrivent les entités présentes dans le système. Celles-ci sont représentées sur la figure 4.1. On trouve ainsi :

- La notion de site CHORUS. Un site CHORUS est une machine où s'exécute un micro-noyau CHORUS. Celui-ci est relié à d'autres sites CHORUS par un support de communication qui peut être un bus ou un réseau,
- La notion d'acteur : un acteur est une machine virtuelle. C'est un espace d'allocation de ressources au sens large. Il est constitué d'un espace d'adressage mémoire ainsi que de toutes les ressources systèmes qui sont allouées durant l'existence de l'acteur. Ces ressources peuvent être des sémaphores, des portes, des activités, etc.
- La notion d'activité. L'activité, ou processus léger[1] est l'unité d'allocation du processeur. Il peut y avoir plusieurs activités dans un même acteur. Celles-ci partagent alors le même espace d'adressage et c'est à l'utilisateur de contrôler les problèmes de concurrence qu'introduit ce partage.
- La notion de porte. C'est l'entité de communication. Bien que deux activités issues du même acteur puissent communiquer par mémoire partagée, ce n'est pas le cas de deux activités appartenant à deux acteurs distincts. Celles-ci communiquent alors par échange de messages grâce aux portes. Les portes possèdent un nom unique dans le temps et dans l'espace. Elles sont allouées par rapport à un acteur mais elles peuvent migrer d'un acteur à un autre. C'est par ce nom unique que les activités adressent ces portes. Les portes peuvent être groupées.

4.1.2 L'architecture des systèmes CHORUS

[1] En anglais, on parle de "thread".

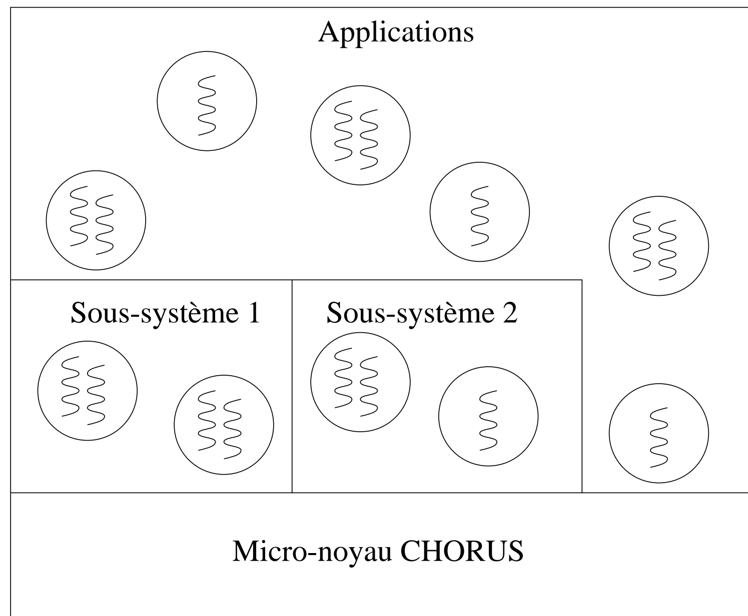

Figure 4.2: L'architecture des systèmes CHORUS

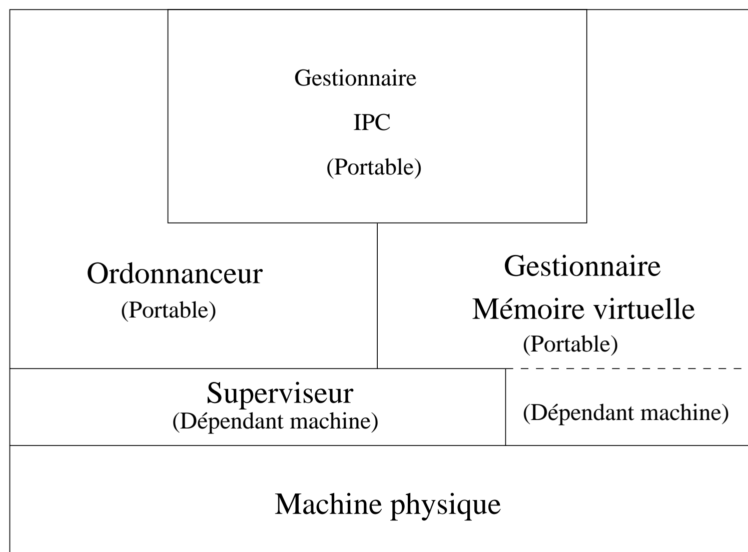

Figure 4.3: Le micro noyau CHORUS

En introduction de ce chapitre, nous avons dit que CHORUS est un "micro-noyau". La notion de micro-noyau implique que le noyau formant le coeur du système d'exploitation soit minimal. L'architecture d'un système CHORUS est représenté sur la figure 4.2. La couche la plus basse est le micro-noyau, celui-ci ne fournit que des fonctions de base indispensables à toutes les tâches. Il s'occupe de trois grandes fonctions (voir 4.3):

- La gestion de la mémoire virtuelle,
- Les IPC[2],
- Et enfin de l'ordonnancement.

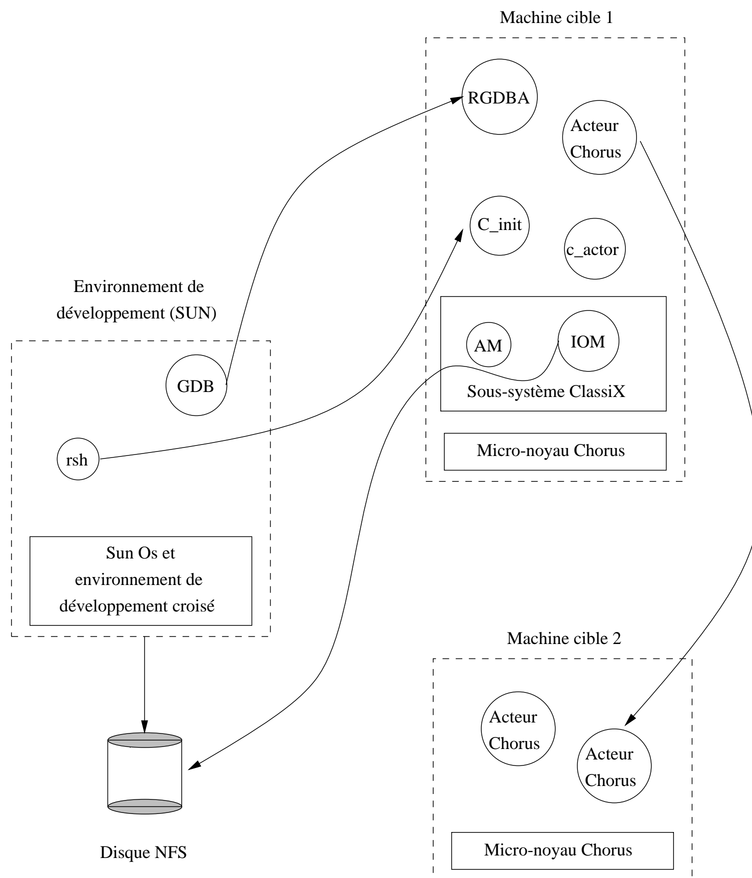

Figure 4.4: Le sous système ClassiX R2

[2]IPC pour **I**nter**P**rocess **C**ommunication.

Un certain nombre d'applications peuvent déjà fonctionner en n'utilisant que ces services. C'est le cas des sous-systèmes. Un sous-système est un ensemble d'acteurs offrant aux entités utilisatrices de niveau plus élevé un ensemble de services plus complexe que les services de base du micro-noyau. Les fonctions non nécessaires à tous les acteurs du système sont implantées dans ces sous-systèmes. Certains de ces sous-systèmes offrent ce que l'on nomme parfois des "personnalités". Une personnalité est un ensemble de composants logiciels capables de supporter une compatibilité binaire avec des applications fonctionnant avec un système d'exploitation différent. Ainsi, dans CHORUS, il existe un sous-système capable d'exécuter des applications System V Release 4 : c'est le sous-système MiX. D'autres sous-systèmes n'ont pas cette vocation de compatibilité binaire. C'est la cas de ClassiX[61] que nous utilisons dans le cadre de notre expérimentation.

ClassiX (voir la figure 4.4) a pour vocation le développement d'applications CHORUS. Un système ClassiX est composé d'une station de travail SUN et d'un ensemble de sites CHORUS sans disque. Ces sites CHORUS peuvent être des compatibles PC à base de processeur Intel. Sur ces machines, que l'on nomme "machines cibles", aucun outil de développement n'est présent. Tous les outils de développement sont sur la station SUN, les machines cibles ne servant qu'à l'exécution. Les exécutables sont accessibles par NFS[3] sur les machines cibles et celles-ci utilisent TFTP[4] pour démarrer. Cette architecture permet de profiter au mieux du confort des stations de travail SUN tout en déchargeant les machines cibles de la charge due à un sous-système tel que MiX. En dehors de la compilation croisée, ClassiX offre aussi un certain nombre de fonctionnalités supplémentaires comme des bibliothèques permettant de manipuler des sockets ou des fichiers, créer et manipuler des acteurs à distance, utiliser des fonctions "*thread-safe*", des outils de mise au point, etc. COOL V3, qui est lui même un sous-système, utilise les services de ClassiX.

4.2 Le standard CORBA

CORBA est un standard qui a été défini par l'OMG[5]. L'OMG est un consortium d'environ 500 membres (essentiellement des industriels). Ce standard fut défini en deux étapes :

- Corba 1.2 d'abord [41, 40] en 1993
- Corba 2.0 ensuite [38, 42] depuis décembre 1994

[3]NFS pour **N**etwork **F**ile **S**ystem.
[4]TFTP pour **T**rivial **F**ile **T**ransfert **P**rotocol.
[5]OMG pour **O**bject **M**anagement **G**roup.

Il décrit l'accès à des objets avec deux objectifs :

- L'interopérabilité entre objets. Les interfaces des objets sont définies dans un langage commun mais leurs implantations peuvent être faites dans des langages différents. CORBA permet aussi de gérer les problèmes d'hétérogénéité des machines dans un réseau. Ainsi, les objets peuvent être invoqués à partir de machines d'architectures et de systèmes différents, c'est CORBA qui gère les problèmes de présentation par le biais de ce que l'on appelle l'ORB[6].
- De transparence à la localisation. En effet, une fois l'application initialisée, les objets sont accédés par les clients d'une manière identique qu'ils soient sur une machine distante ou sur la machine locale.

4.2.1 Le modèle de programmation

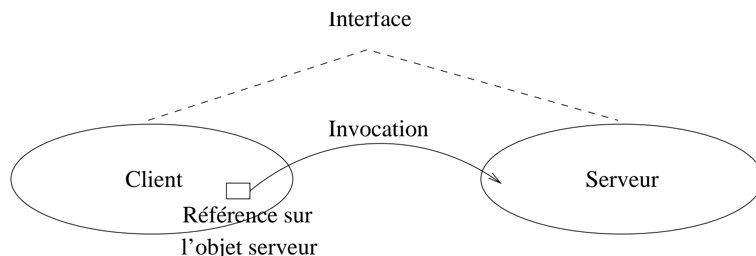

Figure 4.5: Le modèle de programmation de CORBA

Nous allons préciser maintenant ce que nous avons évoqué dans la partie précédente, c'est à dire le mode de fonctionnement des applications CORBA.

Les applications CORBA sont baties sur le modèle clients-serveurs. En effet, ces applications contiennent des objets clients et des objets serveurs. Les objets serveurs sont constitués d'une interface et d'une implantation. Ces deux parties sont dissociées : on peut très bien avoir plusieurs implantations pour une même interface. Ces interfaces sont décrites dans un langage spécial : le langage IDL[7]. Le client possède une référence sur l'objet serveur qu'il veut invoquer. Cette référence est obtenue pendant la phase qui est appelée "phase de liaison".

Chronologiquement, on commence par initialiser le serveur. Celui-ci se fait connaître, lie l'interface de son service avec une implantation, puis attend les requêtes des clients. Le client quant à lui, avant de pouvoir invoquer une méthode de l'objet

[6] ORB pour **O**bject **R**equest **B**roker, nous définirons ce qu'est un ORB dans les pages suivantes.
[7] IDL pour **I**nterface **D**efinition **L**angage.

serveur, doit récupérer une référence sur cet objet. Pour ce qui est des invocations, il existe deux modes d'invocations :

- Les invocations synchrones : le client est bloqué jusqu'à ce que le serveur ait terminé son traitement. C'est le mode utilisé quand le client doit récupérer des informations en retour. Ce mode assure la fiabilité des communications. Il offre aussi des mécanismes d'exceptions qui permettent de traiter les erreurs de communications,

- Les invocations "*oneway*" : le client n'est pas bloqué lors de l'invocation de la méthode. C'est une invocation asynchrone. On ne peut donc récupérer des arguments en retour. De plus, généralement, il n'y a pas de contrôle de la communication. Il n'y a donc pas de contrôle de flux sur le serveur, de contrôle en cas de perte ou de déséquencement des messages. C'est ce qu'on appelle dans un contexte qui n'est pas objet le mode message.

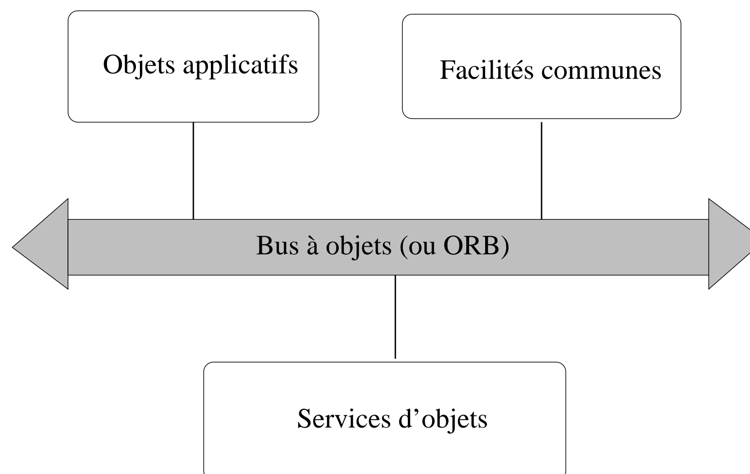

Figure 4.6: L'architecture OMA

Remarque : Un objet peut à la fois être client et serveur. Le modèle de programmation qui vient d'être décrit est représenté par la figure 4.5.

4.2.2 L'architecture OMA

Pour mettre en oeuvre le modèle précédent, l'OMG a défini une architecture : l'OMA[8].

Cette architecture, décrite par la figure 4.6 est constituée de quatre parties :

[8]OMA pour **O**bject **M**anagement **A**rchitecture.

- Le bus à objets. Celui-ci fournit les services de communication entre les objets. C'est lui qui met en oeuvre la transparence à la localisation.

- Les services d'objets. Ce sont les services systèmes définis dans les COSS (**C**ommon **O**bject **S**ervice **S**pecification). Ces services sont des services de bas niveaux, ils sont classés en cinq groupes (COSS1 à COSS 5). Ces services peuvent être des services de persistance, de nommage, de concurrence, etc.

- Les facilités communes: Ce sont des services de plus haut niveau. On les appelle aussi des "*framework*". On peut considérer ces services comme des progiciels. On trouve ainsi des *framework* pour écrire des applications tolérantes aux pannes, des applications de communications, des modèles cohorte/coordinateur, etc.

- Les objets applicatifs. Ce sont les objets clients et les objets serveurs écrits par les utilisateurs.

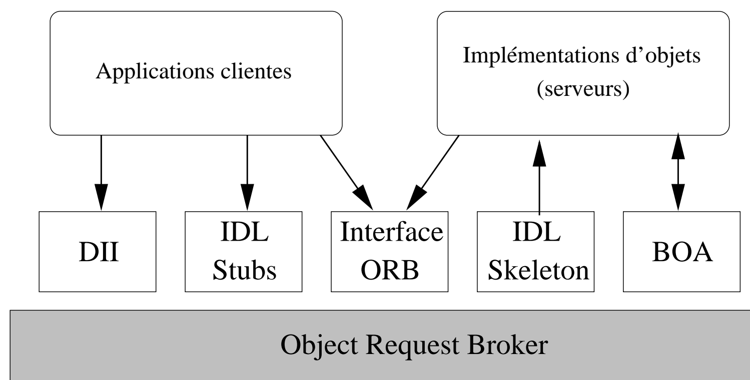

Figure 4.7: La structure d'un ORB

Remarque: Les services d'objets et les facilités communes représenteraient un énorme marché si l'utilisation de CORBA venait à s'étendre, et en particulier en informatique de gestion[9]. Ces services vont nécessiter un très gros effort de développement.

4.2.3 La structure d'un bus à objets

La structure de l'ORB déjà définie dans CORBA 1, a été précisée avec CORBA 2.0. Cette dernière version du standard avait comme objectif principal de rendre possible

[9]La liaison avec le langage COBOL est en cours d'élaboration.

l'interopérabilité entre des ORB de sociétés différentes, ce qui n'était pas le cas avec CORBA 1. En effet, avec le premier standard, la structure interne de l'ORB n'était pas standardisée : chaque constructeur était libre d'en choisir son implantation. La structure d'un ORB CORBA est donnée par la figure 4.7. Cet ORB est constitué de cinq grandes parties :

- Les "*repositories*" (ou dépôts d'objets) : il en existe deux, un pour stocker les interfaces, un autre pour stocker les implantations des objets,
- Le BOA[10] : cette partie est présente du coté serveur. Elle contient les mécanismes nécessaires pour démarrer les requêtes des clients. Par exemple, il s'occupe de créer les processus qui traitent les requêtes, d'instancier les objets, de gérer les références d'objets, etc. Il existe plusieurs sortes de BOA (concurrent, non concurrent, persistant, etc) mais CORBA a standardisé un BOA minimal,
- L'interface de l'ORB : cette interface contient un certain nombre de services permettant par exemple de transformer des références d'objets en chaîne de caractères et vice versa, de savoir si une référence d'objet existe, d'obtenir son interface ou son implantation, etc,
- Le DII (pour "*Dynamic invocation interface*", ou "interface d'invocation dynamique") : les invocations CORBA, un peu comme pour SQL[11], peuvent être réalisées de deux manières différentes :
    1. Par une invocation statique : celle-ci est faite par compilation d'un code IDL. Le compilateur génère à partir de cet IDL une souche et un squelette. Cette souche est ajoutée pendant l'édition des liens au code du client. Du coté serveur, on ajoute la souche et le squelette. Le fonctionnement des souches et des squelettes est expliqué dans les lignes suivantes,
    2. Par une invocation dynamique : l'utilisateur peut choisir de constituer son invocation dynamiquement (c'est à dire durant l'exécution). Le DII est le composant de l'ORB permettant cette technique. Le client consulte alors l'interface *repository* pour rechercher les méthodes qu'il souhaite invoquer, constitue ses arguments, puis fait son invocation. L'intérêt de cette méthode est qu'elle est souple : au cours de l'exécution, de nouvelles interfaces d'objets peuvent être ajoutées dans l'interface *repository*, celles-ci devenant immédiatement invoquables.

[10]BOA pour **B**asic **O**bject **A**dapter.

[11]SQL pour **S**tructured **Q**uery **L**anguage.

- L'IDL stub et l'IDL skeleton : littéralement les souches et les squelettes des interfaces. Ces deux dernières composantes sont obtenues à la compilation. Du coté client, la souche a deux rôles :

  1. Permettre au client de faire l'appel localement, la souche faisant elle même l'appel distant. On parle alors assez souvent d'objet ou serveur "proxy". C'est en partie ce proxy qui permet de rendre les invocations transparentes à la localisation,
  2. Codage/décodage : le codage est l'opération de transformation des types du langage hôte[12] dans un format indépendant du type de machine, de manière à pouvoir transmettre les données sur un réseau de machines éventuellement hétérogènes. Le décodage constitue l'opération inverse. Ce codage permet de s'affranchir de l'éventuelle hétérogénéité des machines mais aussi de l'invocation par un client écrit dans un langage hôte x alors que le serveur est écrit dans un langage y[13]. Cette approche peut être comparée à l'approche syntaxe abstraite de réseau offerte par ASN1-BER[14] dans l'environnement OSI[15].

  Du coté serveur, le squelette permet de connecter l'implantation d'un objet à son interface.

En plus de la structure de l'ORB, CORBA 2.0 a proposé un protocole permettant l'interopérabilité entre applications d'ORB différents. Ce protocole appelé IIOP[16], spécifie comment les messages doivent être transmis sur une pile TCP/IP[17]. Ces messages sont standardisés par le protocole GIOP[18].

4.2.4 Le langage IDL

Jusqu'ici, nous n'avons rien dit sur le langage utilisé pour décrire les interfaces des objets serveurs. Ce langage ainsi que les liaisons avec les langages hôtes constituent une bonne partie du standard CORBA. Le langage de définition des interfaces est l'IDL. L'utilisateur spécifie ses interfaces en IDL, puis, un compilateur, génère le code de la souche et du squelette.

[12] Le langage hôte est le langage utilisé pour l'écriture des implantations des objets.

[13] A titre d'exemple, chez Dassault Aviation, un compilateur IDL a été écrit afin d'interfacer des objets écrits en Le-Lisp avec d'autres en C++.

[14] ASN1 pour **A**bstract **N**otation **S**yntax, **B**asic **E**ncoding **R**ules.

[15] OSI pour **O**pen **S**ystems **I**nterConnection.

[16] IIOP pour **I**nternet **I**nter-**O**RB **P**rotocol.

[17] TCP/IP pour **T**ransmission **C**ontrol **P**rotocol/**I**nternet **P**rotocol.

[18] GIOP pour **G**eneral **I**nter-**O**RB **P**rotocol.

Ce langage IDL décrit les invocations que peuvent faire les clients : nom des méthodes, prototypes, modes d'invocations. Mais il permet aussi la description d'exceptions, de constantes et d'attributs. Il est orienté objet, il intègre donc les notions d'héritage. Les langages hôtes pour lesquels une liaison avec l'IDL a été définie par l'OMG sont le C, C++, Smalltalk, et demain le COBOL.

```
module legume
    {
    interface tomate : radis
        {
        const float prix = 100.0;
        typedef sequence any liste;
        readonly long nombre;
        void achete-tomates(out long
            resultat, in liste lst);
        oneway mange-tomate(in long nb);
        }
    }
```

Figure 4.8: Exemple de code en IDL

Un exemple de code IDL est donné par la figure 4.8. Dans cet exemple, on trouve la définition de deux méthodes :

- Une méthode synchrone qui récupère la variable "resultat". C'est la méthode "achete-tomates",
- Une méthode asynchrone (dite *oneway*) dont le nom est "mange-tomate".

Dans la méthode "achete-tomates", on spécifie les résultats renvoyés par le serveur grâce au mot clef *out*. Les paramètres passés au serveur sont spécifiés par le mot clef *in*. Les constantes sont définies par le mot clef *const*. On peut aussi construire des types comme avec le type liste. La syntaxe d'IDL est proche de celle du langage C.

4.2.5 Conclusion sur CORBA

Nous venons de décrire brièvement le standard CORBA. Celui-ci permet de développer des applications réparties très facilement. Bien que ce standard soit récent, il

existe aujourd'hui de nombreux ORB CORBA universitaires ou industriels. Toutefois, il est difficile de dire si CORBA continuera à se développer car il a un concurrent sérieux : OLE[19] . Celui-ci possède un avantage important : une importante base installée de machines fonctionnant sous les différents systèmes d'exploitation Microsoft.

4.3 L' ORB de CHORUS : COOL V3

COOL[20] V3 est disponible commercialement depuis février 1996 [63, 62]. Il est totalement compatible avec le standard CORBA 2.0. Il offre donc tous les services spécifiés par ce standard. De plus, COOL offre des services supplémentaires qui devraient permettre d'atteindre les objectifs suivants :

1. Permettre un développement facile de sous-systèmes CHORUS et d'applications pour les télécommunications ou la productique,

2. Interopérabilité. Le standard CORBA 2.0 doit rendre possible cette propriété. De plus, COOL V3 a été écrit sur plusieurs plates-formes (Fusion, ClassiX, Windows 95, Windows NT, SCO, Sun OS). Pour ce faire, il peut utiliser pour la communication soit les IPC[21] CHORUS soit les sockets TCP/IP. Toutefois, il n'implante pas le protocole IIOP,

3. Utilisation pour des applications coopératives grâce à la notion de groupe d'objets.

4.3.1 Les abstractions de COOL

Comme pour le micro-noyau CHORUS, COOL introduit un certain nombre d'abstractions. Celles-ci sont d'ailleurs assez proches des abstractions du micro-noyau CHORUS. COOL définit donc :

- La notion de noeud COOL : c'est en fait un site CHORUS. Un noeud COOL ne peut appartenir qu'à un seul domaine,

- La notion de domaine : un domaine est une collection de noeud COOL. Un domaine délimite l'espace de nommage de COOL : nous verrons que COOL permet de donner un nom de service à un objet serveur. Cette notion de service est la même notion que celle dans TCP/IP,

[19]OLE pour **O**bject **L**inking **E**mbedding.
[20]COOL pour **CHORUS** **O**bject **O**riented **L**ayer.
[21]IPC pour **I**nter **P**rocess **C**ommunications.

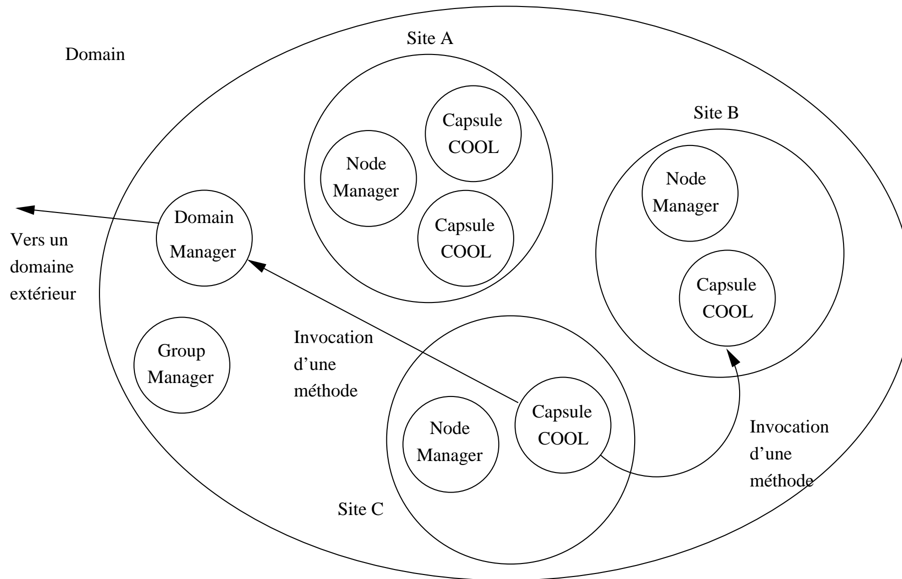

Figure 4.9: Architecture de COOL V3

- La notion de capsule : c'est l'espace d'adressage dans COOL (la notion de capsule COOL est la même que la notion d'acteur CHORUS),

- La notion d'activité COOL : si l'on considère COOL sur ClassiX, alors une activité COOL est un fil d'exécution distribué constitué des différentes activités CHORUS utilisées lors de l'exécution d'une application. Supposons que nous utilisions une application qui se déroule localement en dehors d'un seul appel de méthode à distance. Dans ce cas, l'activité COOL est constituée de l'activité CHORUS locale plus l'activité CHORUS du serveur distant.

La figure 4.9 illustre les notions que nous venons d'introduire. COOL est en fait constitué de trois acteurs CHORUS. Ces acteurs mettent en oeuvre les abstractions ci-dessus. On trouve :

- Le "*node Manager*" qui est présent sur chaque noeud COOL,

- Le "*domain Manager*" qui s'occupe des domaines COOL. Il y a un "*domain Manager*" par domaine COOL,

- Le "*group Manager*" qui gère les groupes d'objets COOL. Il y a un "*group Manager*" par domaine COOL.

4.3.2 Le compilateur CHIC

CHIC[22] est le compilateur IDL de COOL. Il implante toutes les spécifications de l'IDL faite par l'OMG en dehors des modules. Seul le C++ peut être utilisé comme langage hôte.

La construction des programmes sous COOL est constituée des étapes suivantes :

- Tout d'abord, l'utilisateur écrit le source IDL de l'interface qu'il veut mettre en oeuvre,
- Puis il édite un fichier Imakefile : COOL fournit en plus du compilateur Chic un outil permettant de générer le Makefile qui servira à la construction des exécutables [23]. Dans ce fichier Imakefile, l'utilisateur spécifie avec quelle implantation l'interface doit être associée (CORBA permet de lier plusieurs implantations différentes pour une même interface),
- L'utilisateur doit ensuite écrire le code C++ qui va servir d'implantation aux objets décrits par les interfaces IDL,
- Enfin vient la phase de compilation : l'appel des dépendances du Makefile généré par CoolMkMf produit les classes C++ correspondant au code IDL,

[22]CHIC pour **CH**orus **I**dl **C**ompiler.
[23]Cet outil s'appelle CoolMkMf.

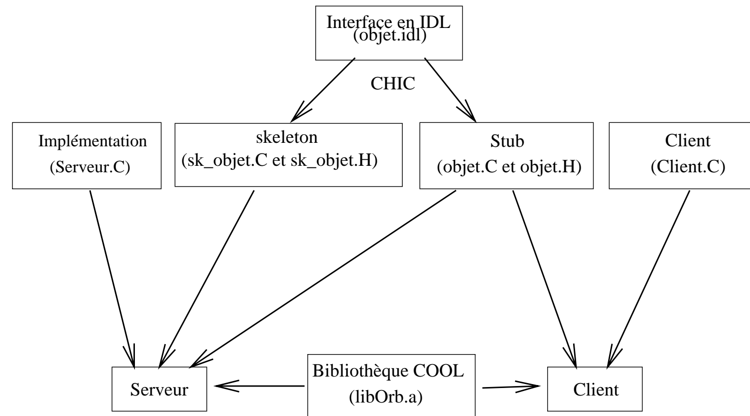

Figure 4.10: Le compilateur CHIC

- La suite est identique à tout autre ORB CORBA : Au code du serveur on ajoute les souches et les squelettes générés par CHIC, puis les bibliothèques de COOL. Au code du client on ajoute les souches et les bibliothèques de COOL.

Ce processus est résumé sur la figure 4.10

4.3.3 Les services supplémentaires offerts par COOL

```
class COOL-TimerCallback {
public:
        COOL-TimerCallback();
        COOL-TimerCallback(unsigned long interval);
        int setTimerOut(unsigned long interval);
        void remove();
        virtual void notifyTimer() = 0;
};
```

Figure 4.11: Notifications sur chiens de garde

En dehors des services standards CORBA, COOL offre d'autres fonctionnalités comme :

- Un service de nommage. Quand un serveur commence son initialisation, il a la possibilité d'enregistrer une chaîne de caractères comme étant le service qu'il fournit. Les clients lorsqu'ils démarreront, pourront obtenir une référence sur ce serveur grâce à cette chaîne de caractères,

```
class COOL-InputCallback {
public:
        COOL-InputCallback();
        COOL-InputCallback(int fd,
            COOL-InputMode mode);
        int setInput(int fd,
           COOL-InputMode mode);
        void remove();
        int file() const;
        virtual void notifyInput() = 0;
};
```

Figure 4.12: Notifications sur entrées/sorties

- Un service de notification d'événements. COOL permet d'appeler une fonction ou une méthode d'une capsule lors de l'occurrence d'un événement. Ces événements peuvent être de deux types, soit des chiens de garde, soit des entrées/sorties. Les figures 4.11 et 4.12 donnent les définitions des classes C++ de ces services. Les utilisateurs font dériver leurs objets avec ses classes, puis surchargent les méthodes virtuelles avec le code qu'ils souhaitent voir être appelé lors de l'occurrence des événements,

- Invocation sur groupe d'objets. COOL intègre la notion de groupe d'objets. Les groupes COOL sont très proches des groupes du micro-noyau CHORUS,

- Des outils de synchronisation distribués.

* Les outils de synchronisation COOL

COOL fournit deux types d'outils de synchronisation : des outils sous formes d'objets et des primitives permettant d'influencer le fonctionnement des activités COOL.

- Sur les activités on peut utiliser trois primitives :

  1. Delay(). Cette primitive permet de suspendre l'activité concernée,
  2. Join(). Cette primitive offre la possibilité de synchroniser deux activités. Une activité effectuant un join() est bloquée jusqu'à l'arrivée de l'activité attendue,

```
interface COOL-DRWLock {
        boolean writerGet(in long delay);
        void writerRel();
        boolean readerGet(in long delay);
        void readerRel();
        void destroy();
};
```

Figure 4.13: Objets Lecteurs/rédacteurs de COOL

```
interface COOL-DMutex {
        void get();
        boolean tryGet();
        void rel();
        void destroy();
};
interface COOL-DSem {
        boolean P(in long delay);
        void V();
        void destroy();
};
```

Figure 4.14: Sémaphores et mutex de COOL

3. Detach(). Cette primitive permet à partir d'une activité d'en créer une autre. Supposons que nous utilisions COOL sur un micro-noyau CHORUS. Sur CHORUS, les activités COOL sont réalisées par des activités CHORUS. Ainsi, une activité COOL s'exécutant sur un site A et effectuant une invocation de méthode sur le site B est donc constituée de deux activités CHORUS. Grâce à detach(), on peut créer une nouvelle activité COOL sur le site B avec la deuxième activité CHORUS. A l'issue de l'invocation, on obtient donc deux activités indépendantes,

- Les objets lecteurs/rédacteurs. Ces objets sont distribués. **Les services sont spécifiés sous forme d'interface IDL** (voir figure 4.13).

- Les objets sémaphores et mutex distribués. Ces objets sont aussi spécifiés en IDL. Ils sont strictement identiques aux mutex et sémaphores du micro-noyau en dehors du fait qu'ils soient distribués (voir figure 4.14).

* **L'invocation sur groupe de COOL**

La notion de groupe COOL est liée à la notion de groupe du micro-noyau CHORUS. Il n'y a donc pas de notion d'atomicité ou de protocole ordonné comme nous l'avons vu dans Electra[34, 35, 28, 32, 33] et dans Isis[15, 57, 16, 27, 58]. L'invocation sur groupe d'objets est faite de manière transparente du coté client. Les objets serveurs peuvent être insérés ou retirés dynamiquement. Lors de la création d'un groupe COOL, l'utilisateur doit spécifier de quelle manière les objets vont être atteints et de quelle manière les résultats vont être collectés. L'appel de méthode sur les différents objets du groupe peut être fait :

- Soit en HALF_RELIABLE : COOL envoie à chaque serveur la requête. Cette option rend la diffusion de la requête fiable vis à vis des erreurs de communication qui pourraient survenir durant l'invocation,
- Soit en BROADCAST : COOL envoie la requête à chaque serveur accessible. Aucun mécanisme de contrôle des erreurs de communication n'est mis en place,
- Soit en FUNCTIONAL : lorsqu'on insère un serveur dans un groupe, l'utilisateur doit fournir à COOL une priorité qui est associée au serveur. Dans ce mode de fonctionnement, COOL envoie la requête au serveur de plus haute priorité,
- Et enfin, soit en FUNCTIONAL_RANDOM : le fonctionnement est très proche du mode FUNCTIONAL. Ici, dans le cas où plusieurs serveurs auraient la même priorité, le serveur est choisi au hasard parmi ceux-ci.

Quand à la manière dont les résultats peuvent être récupérés, on dispose de deux possibilités :

- FIRST_EXCEPTION : dans ce mode, on attend la première exception, toutes les réponses précédentes sont ignorées. Dans le cas où il n'y a pas d'exception , la première réponse est renvoyée,
- ou FIRST_REPLY : COOL renvoie la première réponse sans tenir compte des réponses ou des exceptions qui arriveraient plus tard.

4.3.4 Conclusion sur COOL

Nous venons de présenter l'ORB de CHORUS. Celui-ci permettra de développer facilement des applications objets réparties avec CHORUS, mais aussi avec d'autres systèmes. De plus, puisque COOL fonctionne sur divers environnements, il va simplifier la communication de systèmes CHORUS avec d'autres systèmes d'exploitation.

Nous venons de décrire dans ce chapitre la plate-forme et les outils que nous avons utilisés pour la réalisation de nos expérimentations. Le modèle Saturne et l'approche synchrone forte ont été abordés dans les chapitres précédents. Nous proposons maintenant dans les chapitres cinq et six une implantation de Saturne sur CHORUS/COOL et donnons les résultats que nous en avons obtenu.

Chapitre 5

Le sous-système Saturne sur CHORUS/COOL

5.1 Introduction

Dans ce chapitre, nous décrivons les techniques nécessaires pour implanter le modèle Saturne sur CHORUS/COOL. Les objectifs de cette étude sont de déterminer dans quelles conditions Saturne peut être réalisé sur un système d'exploitation temps réel à l'aide d'un bus à objet au standard CORBA. La version du modèle Saturne utilisé pour nos expérimentations est la version mono-synchrone. Bien que ce modèle existe maintenant depuis plusieurs années, il n'y a pas eu à notre connaissance d'implantation **dans un environnement temps réel distribué**, alors que son objectif est l'exécution d'applications temps réel. Plusieurs implantations sur Unix furent toutefois réalisées. Nous citerons celle du Copilote électronique, menée conjointement par Dassault Aviation et le CERT. Enfin, le modèle formel du synchronisme faible (dont le nom est COREA[7]) fut utilisé par O. Potoniée dans sa thèse[49] pour proposer un modèle d'exécution d'objets synchrones [11] sur CHORUS. L'originalité de notre implantation est donc l'utilisation d'une plate-forme temps réel pour la mise en oeuvre de Saturne.

Dans le paragraphe deux, nous décrirons l'architecture que nous avons choisie en terme d'acteurs et d'activités CHORUS, puis, le paragraphe trois expliquera la simulation des tops d'horloge. Le paragraphe quatre traitera de la résolution des problèmes d'ordonnancement. Nous donnerons ensuite une description des acteurs constituant les noyaux synchrones dans le paragraphe cinq. Le protocole entre le noyau réactif et l'interface réactive sera décrit dans le paragraphe six. Enfin, nous présenterons la manière dont le modèle Saturne peut être conçu en termes d'interfaces IDL et d'objets COOL en sept.

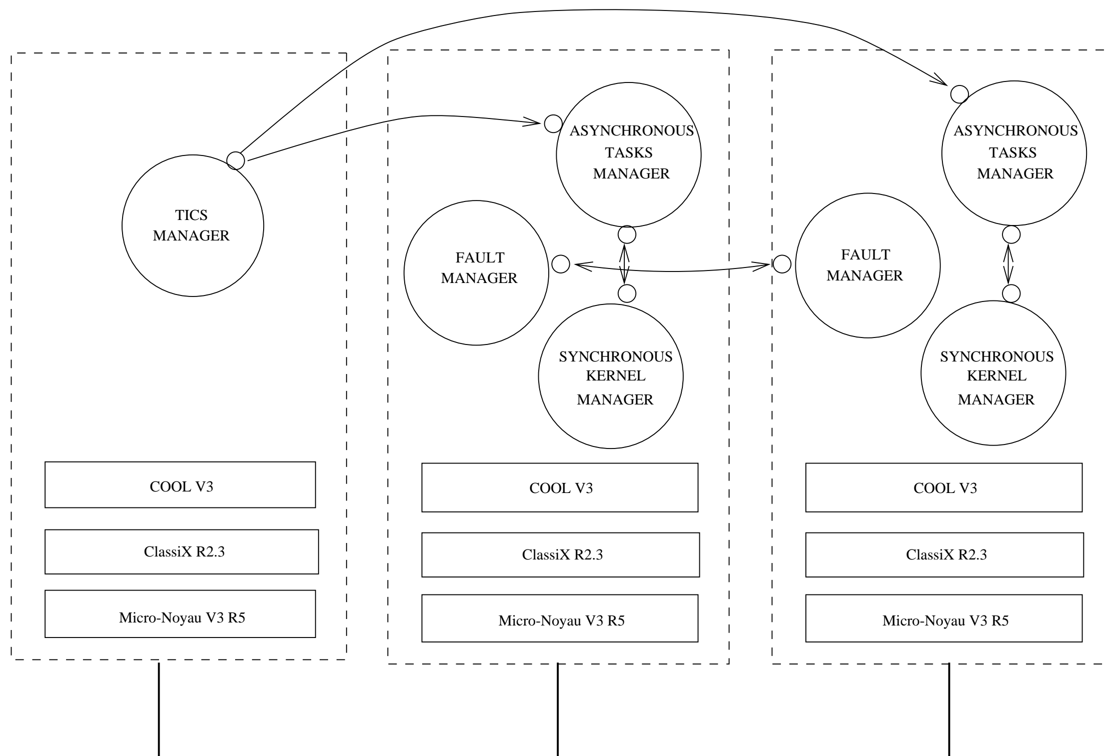

Figure 5.1: Architecture du sous système Saturne

5.2 Architecture du sous-système Saturne sur CHORUS/COOL

Dans le modèle Saturne, les grappes sont connectées par deux supports de communication (un support synchrone, un autre asynchrone). Dans nos simulations, nous n'implanterons pas l'utilisation du support asynchrone (c'est à dire les communications entre tâches transformationnelles). Nous simulerons uniquement les communications synchrones entre les noyaux. Nous ne faisons aucune supposition quant à la répartition des grappes sur le réseau : on peut imaginer qu'il y ait une ou plusieurs grappes par processeur. Ces grappes se composent de deux parties:

- Le noyau réactif,
- Les tâches transformationnelles.

Les abstractions de CHORUS s'adaptent facilement à cette architecture. Les composantes de Saturne sont réalisées sous forme d'acteurs CHORUS qui sont :

- L'AM (Asynchronous tasks Manager) : cet acteur contient toutes les tâches transformationnelles, l'interface réactive de Saturne et un ordonnanceur pour les tâches transformationnelles. Chaque tâche transformationnelle est exécutée par une activité CHORUS. L'interface réactive et l'ordonnanceur sont implantés de manière identique. L'interface réactive est bloquée en attente de messages venant du noyau synchrone. Le protocole entre noyau synchrone et interface réactive est décrit dans le paragraphe six. L'ordonnanceur est décrit dans le paragraphe quatre,

- Le SM (Synchronous kernel Manager) : cet acteur encapsule le code Esterel. Il représente un noyau synchrone du modèle Saturne. Il reçoit régulièrement des tops en provenance du TM qui est décrit ci-dessous. A chaque top reçu, le SM envoie ses commandes vers l'AM et diffuse ses signaux vers les autres noyaux synchrones,

- Le TM (Tics Manager) : c'est l'acteur qui génère à intervalles réguliers des tops en direction des SM et des FM. Il implante l'horloge globale du modèle synchrone faible. Le fonctionnement de cet acteur est décrit dans le paragraphe trois. Il y a un seul TM dans un système CHORUS,

- Le FM (Fault Manager) : cet acteur est responsable de la mise en oeuvre des mécanismes de tolérance aux fautes que nous souhaitons utiliser. Son fonctionnement est décrit dans [53]. Il y a un FM par site CHORUS.

Pour des contraintes techniques et de performances, les SM et FM sont des acteurs superviseurs. Les AM sont des acteurs utilisateurs. L'architecture du sous-système CHORUS/COOL est représentée dans la figure 5.1.

5.3 Simulation du top

5.3.1 Présentation du problème

Dans le modèle Saturne, les noyaux sont activés périodiquement grâce à une horloge logique globale. Ce mode de fonctionnement est lié au modèle synchrone faible décrit dans le chapitre trois. Pour réaliser cette horloge, il existe deux grandes catégories de solution :

- Les solutions distribuées constituées par des synchronisations d'horloge. Elles reposent sur des algorithmes complexes et nécessitant, pour obtenir une bonne précision, de disposer de temps de communication bornée,

- Les solutions centralisées qui consistent à diffuser les tops de l'horloge sur le réseau. Ces solutions sont adaptées aux systèmes fermés où les distances sont faibles, ce qui est le cas des systèmes d'avioniques.

5.3.2 Réalisation sous CHORUS

La solution retenue pour générer les tops à intervalles réguliers est celle d'un acteur qui effectue une diffusion sur l'ensemble des FM et des SM. L'acteur qui génère ces tops a été appelé le TM (Tics Manager).

Pour émettre des messages régulièrement vers tous les SM et les FM, on utilise un groupe de diffusion CHORUS. Le TM est constitué d'une seule activité. Celle-ci est réveillée toutes les n milli-secondes et diffuse alors le signal de top. Le signal ne peut pas être généré par une communication synchrone : en effet, on souhaite obtenir une simulation la plus proche possible d'une communication cyclique, or les appels successifs de tous les SM et FM en synchrone introduiraient trop de décalage entre le premier signal envoyé et le dernier.

REMARQUE IMPORTANTE:
Le génération du top d'horloge **n'est qu'une simulation.** C'est pour cette raison que l'on se contente d'une diffusion CHORUS non fiable et non atomique. Dans un système réel, l'horloge de base devra être parfaitement fiable et précise et il est fort possible qu'une solution matérielle soit la plus adaptée à ce problème.

5.4 Résolution des problèmes d'ordonnancement

Avant de décrire les solutions envisagées pour l'ordonnancement d'une application Saturne sur CHORUS, nous allons brièvement rappeler les outils disponibles sur CHORUS.

5.4.1 L'ordonnancement sur CHORUS

L'ordonnancement de CHORUS est de type préemptif. Il se base sur une priorité unique. L'ordonnanceur CHORUS garantit que, sur une machine monoprocesseur, c'est l'activité de plus haute priorité qui s'exécute.

En fait, cette unique priorité manipulée par l'ordonnanceur CHORUS limite les modèles d'ordonnancement possibles. Ainsi, comment faire du temps partagé ? Dans le micro noyau, Chorus système a introduit la notion de classe d'ordonnancement. Chaque activité est placée dans une classe et une seule. On peut déplacer une activité

d'une classe à une autre. Les classes d'ordonnancement peuvent être vues comme des fonctions qui modifient la valeur de la priorité dans l'ordonnanceur CHORUS. On parle alors de priorité relative (priorité dans la classe d'ordonnancement) et de priorité absolue (priorité dans l'ordonnanceur CHORUS). Pour obtenir un modèle d'ordonnancement différent, tout le travail de la classe d'ordonnancement est de transformer cette priorité relative en une priorité absolue adaptée.

Dans le noyau actuel, on trouve ainsi un certain nombre de classes d'ordonnancement :

- K_SCHED_DEFAULT qui est la classe par défaut : la priorité de l'activité est ajoutée à celle de l'acteur pour obtenir la priorité absolue. Toute valeur de la priorité absolue inférieure à K_PRIOMIN est initialisée à K_PRIOMIN,
- Trois autres classes d'ordonnancement qui ont été conçues pour la réalisation du sous-système MiX[1] et qui sont :
    1. K_SCHED_SVR4_SYS,
    2. K_SCHED_SVR4_RT,
    3. et K_SCHED_SVR4_TS qui est la classe permettant de faire du temps partagé sous Unix.

Remarque : il est tout à fait possible pour les utilisateurs de CHORUS, bien que ce ne se soit pas trivial, de redéfinir de nouvelles classes d'ordonnancements. Ainsi au CNAM, deux élèves[20] ont développé une classe d'ordonnancement permettant de faire de l'EDF[2]. Ce travail a été repris par C. Santellani[26].

Grâce à ces nombreuses classes d'ordonnancement, **le micro noyau CHORUS est capable de faire fonctionner des applications temps réel avec des applications temps partagé sur un même site**. Cette fonctionnalité est exploitée par les PABX A4400[51]. Les autocommutateurs A4400 sont constitués d'un sous-système MiX ainsi qu'une application temps réel. Sur le sous-système MiX s'exécutent des applications d'administrations du commutateur sans contrainte critique. L'application temps réel réalise l'ouverture des connexions X25 et des circuits virtuels téléphoniques. L'objectif est de permettre à l'application temps réel de s'exécuter dès qu'un signal arrive et ce même si l'application MiX exécute un travail nécessitant beaucoup de ressource processeur. L'application MiX est alors préemptée.

[1] MiX est le sous-système CHORUS qui permet d'exécuter des applications Système V Release 4.

[2] EDF pour **E**arliest **D**eadline **F**irst.

On voit qu'il existe certaines convergences d'architecture entre le modèle d'exécution des commutateurs ALCATEL 4400 et le modèle Saturne. En effet, l'équivalent de l'application CHORUS pour le commutateur est le noyau synchrone de Saturne. Les applications MiX se substituent quant à elles aux tâches transformationnelles. Les tâches transformationnelles peuvent consommer le temps processeur entre deux tops, par contre, lors de la réception d'un top, les noyaux synchrones doivent prendre l'ensemble des ressources processeur pour s'exécuter le plus rapidement possible. **On doit garantir aux noyaux synchrones suffisamment de réactivité.**

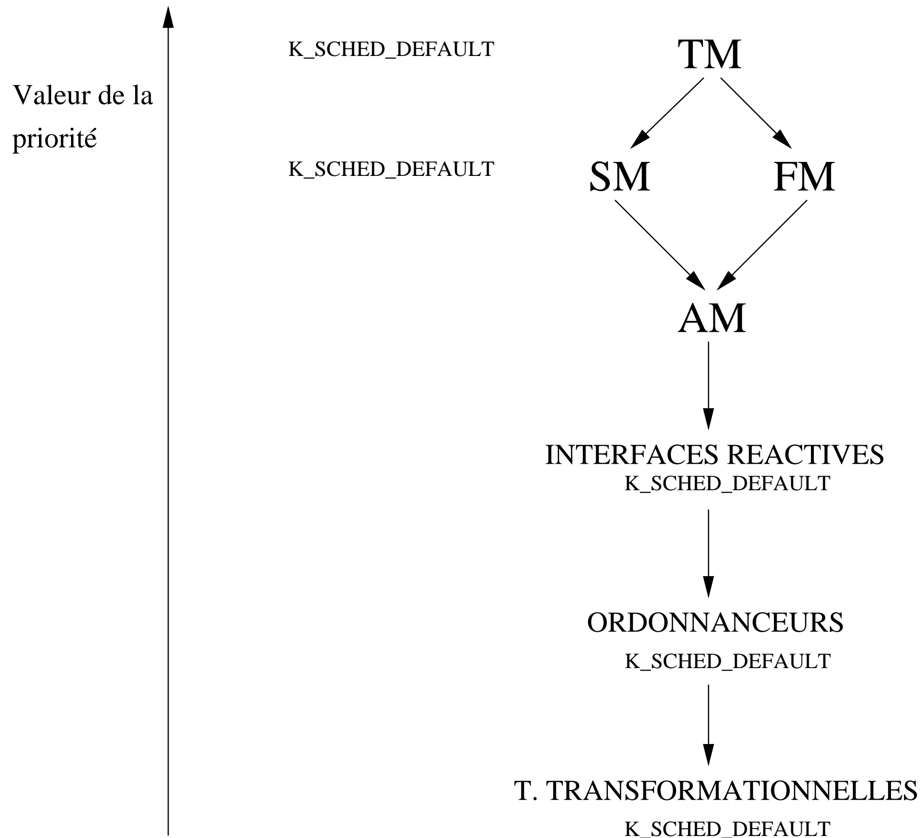

Figure 5.2: Les priorités des acteurs Saturne

5.4.2 Mise en oeuvre avec Saturne

L'ordonnancement des activités CHORUS constituant le sous-système Saturne est effectué de la manière suivante :

1. Le TM possède la priorité maximale et est ordonnancé dans la classe par défaut,

2. Les SM possèdent une priorité juste inférieure à celle du TM afin de pouvoir prendre la main dès l'envoi des tops. Ils sont ordonnancés dans la classe par défaut,

3. La priorité du FM doit être la même que celle des SM. En effet, les FM sont aussi activés par les tops du TM. Leur temps d'exécution doit être compris dans la durée de l'intervalle entre deux tops d'horloge du TM,

4. Les tâches transformationnelles sont ordonnancées dans la classe par défaut avec une priorité identique pour toutes les tâches transformationnelles. Cette valeur doit être inférieure à celle de l'interface réactive de Saturne [3]. En effet, lorsqu'un noyau synchrone est activé et qu'il envoie des commandes à son interface réactive, on souhaite suspendre les tâches pour que l'interface réactive obtienne le processeur au plus vite et serve le plus rapidement possible le noyau synchrone. Pour la même raison, la priorité de l'interface réactive doit être inférieure à celle des SM afin de garantir au SM suffisamment de réactivité,

5. Pour l'ordonnancement des tâches transformationnelles, on ne souhaite pas utiliser les algorithmes fournis par le micro noyau CHORUS. En effet, dans de nombreuses applications, l'ordonnancement est statique mais il existe plusieurs algorithmes d'ordonnancement dynamique qui peuvent être utilisés dans des applications temps réel (tel que rate monitonic ou EDF). Un système Saturne comprend plusieurs noyaux synchrones. A chaque noyau synchrone est associé un AM. On peut très bien imaginer que tout ces AM n'aient pas forcément besoin du même algorithme d'ordonnancement. La présence de l'ordonnanceur dans l'AM permettra au développeur d'applications Saturne de choisir l'ordonnanceur qui s'adapte le mieux à ses tâches transformationnelles. Dans nos prototypes, les tâches n'ont pas de contraintes d'échéances, nous avons donc opté pour la réalisation d'un ordonnanceur de round-robin afin de simplifier notre mise en oeuvre. Le temps partagé n'est pas adapté aux applications temps réel, mais dans le cadre de cette expérimentation, l'ordonnancement des tâches transformationnelles par un algorithme de ce type nous suffit. Notre algorithme très simple suspend et active à tour de rôle les activités CHORUS qui constituent les tâches transformationnelles. Il est clair que dans une implantation réelle, un algorithme temps réel sera utilisé. La

[3] Rappelons que toutes les tâches transformationnelles d'un noyau synchrone, l'interface réactive ainsi que l'ordonnanceur des tâches transformationnelles se trouvent dans l'acteur AM.

possibilité de connecter facilement à un AM un algorithme d'ordonnancement différent a donc été notre objectif principal dans la réalisation de cet acteur.

La figure 5.2 résume les priorités relatives des activités du sous-système Saturne. Dans cette figure, les différents acteurs composants le sous-système Saturne sont triés dans l'ordre de leur priorité. Si l'on résume l'enchaînement des différents composants de Saturne lors d'un top de l'horloge, le TM est le premier à prendre la main afin de diffuser le top de l'horloge globale. Lorsqu'il ne diffuse pas de top, il est endormi pour libérer le processeur. Une fois ce top émis et reçu, les SM prennent le processeur afin de réaliser une transition de leur automate Esterel. En fonction de leurs besoins, ils émettent des signaux à d'autres noyaux synchrones et envoient des commandes vers leur interface réactive. Puis, les interfaces réactives qui sont sollicitées réagissent et effectuent les commandes des noyaux synchrones. Ceci termine les traitements déclenchés par un top du TM. Les ordonnanceurs de chaque AM et les tâches transformationnelles utilisent le reliquat de processeur entre deux tops. La fréquence du top doit donc être correctement dimensionnée pour garantir aux tâches transformationnelles suffisamment de ressource processeur.

5.5 Les noyaux synchrones

Dans ce paragraphe, nous allons décrire les acteurs SM, puis nous verrons le mode de communication utilisé entre les noyaux synchrones.

5.5.1 Architecture des acteurs SM

Les acteurs SM, comme le montre la figure 5.3, sont composés de :

1. Trois activités :
   - Une première activité qui exécute le noyau Esterel et qui est bloquée sur un sémaphore en attente des signaux à consommer dans la file d'attente de réception,
   - Une deuxième activité chargée de la réception des signaux. Elle reçoit et stocke dans la file de réception les signaux en continu,
   - Une troisième activité qui attend les tops émis par le TM. A chaque top, elle détermine les signaux qui peuvent être délivrés au noyau Esterel grâce aux temps de latence en réception. Elle réveille ensuite le noyau Esterel,
2. Deux files d'attentes qui permettent de stocker les signaux jusqu'à ce que les temps de latence soient écoulés,

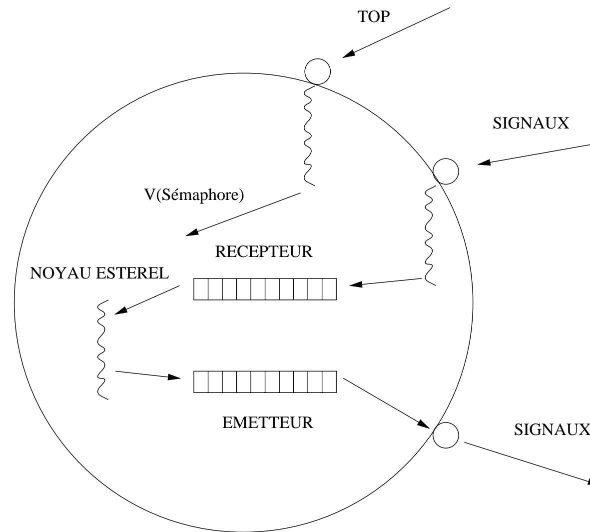

Figure 5.3: Description de l'acteur SM

3. Une horloge H_{tops} qui stocke l'horloge globale courante. Cette horloge est partagée par les trois activités de l'acteur. Avec cette horloge (qui est une horloge logique), ils déterminent si les temps de latence des signaux sont écoulés. Les temps de latence sont définis dans le chapitre trois.

5.5.2 Communications entre les noyaux synchrones

Toutes les communications entre noyaux sont exécutées par des messages asynchrones. Bien que les messages soient envoyés de manière asynchrone, leur transport doit être fiable. Les communications asynchrones permettent d'augmenter le parallélisme des réceptions et des émissions de requêtes et donc de diminuer l'intervalle entre deux tops d'horloge. Les temps de latence sont simulés par un retard en nombre de top. Le fonctionnement du SM pendant un cycle peut se résumer de cette façon :

- L'activité de réception reçoit tous les signaux envoyés par les autres noyaux. Cette réception est continue. Chaque signal reçu est stocké dans la file de réception et est estampillé par l'horloge H_{tops},

- L'activité en attente des tops est réveillée par le TM. A cet instant, elle met à jour H_{tops}. Puis, elle regarde tous les signaux de la file de réception qui peuvent être délivrés (en fonction du temps de latence L_{recep} associé à chaque signal). Enfin, elle débloque le noyau Esterel, puis, se met en attente du prochain top,

- L'activité du noyau Esterel est bloquée sur son sémaphore. Quand elle est débloquée, elle consomme les signaux que l'activité en attente des tops lui a donnés. Elle effectue ensuite une transition de l'automate Esterel. Les signaux de sorties qu'elle émet pendant la transition sont estampillés par H_{tops} et sont stockés dans la file d'attente en émission. A la fin de son exécution, elle consulte la file d'attente en émission pour voir si elle peut envoyer des signaux. Elle tient compte de l'estampille et du temps de latence associé à chaque signal,

- Puis on recommence un autre cycle au prochain top du TM.

5.6 Description du protocole entre le noyau et l'AM

Dans ce paragraphe, nous allons décrire les techniques que nous avons utilisées pour la réalisation de ce protocole d'échange.

L'interface réactive est réalisée par une activité au sein de l'AM. Elle exécute les ordres du noyau synchrone sur les tâches transformationnelles. Les échanges entre l'AM et l'interface réactive sont faits par des communications synchrones. De ce fait, les temps de traitement de l'interface réactive devront aussi être intégrés dans la durée de l'intervalle entre deux tops. Les deux paragraphes suivants décrivent les tâches transformationnelles ainsi que les interactions entre le noyau synchrone, l'interface réactive et les tâches transformationnelles.

5.6.1 Description d'une tâche transformationnelle

Une tâche transformationnelle est un objet de la classe *task* dont les membres sont :

- Le constructeur de classe qui est appelé lors de la création de la tâche. C'est lui qui démarre la nouvelle activité CHORUS. La commande Saturne correspondante est START nom_tâche. Les primitives CHORUS utilisées sont celles qui allouent les ressources nécessaires ainsi que les primitives de création d'activités. Dans une implantation réelle, on connaît les tâches qui vont être nécessaires pendant l'exécution de l'application ainsi que leur nombre. Elles seront allouées au démarrage du système ce qui permettra d'améliorer les performances,

- Le destructeur de classe libère les ressources allouées par l'activité. Il est déclenché quand le SM a été averti que la tâche est terminée,

```
class task {
public:
        task();
        ~task();
        suspend();
        resume();
        kill();
        ended();
protected:
        int li;
        int Status;
        int nbExemp;
        char stack[STACKSIZE];
}
```

Figure 5.4: La classe des tâches transformationnelles

- Les méthodes suspend() et resume() correspondent aux primitives Saturne SUSPEND nom_tâche et RESUME nom_tâche. Elles sont directement implantées par les primitives CHORUS,
- La méthode kill() correspond à la destruction de l'activité encapsulant la tâche transformationnelle. La primitive STOP nom_tâche de Saturne est réalisée par un appel à la méthode kill(),
- La méthode end() positionne l'attribut Status pour signaler la fin de la tâche transformationnelle. Cette méthode est appelée par la tâche afin d'avertir l'interface réactive qu'elle se termine,
- Le numéro d'exemplaire Saturne est le numéro que l'on peut passer en paramètre à la commande START (ex: START nom_tâche (numéro d'exemplaire) (paramètres d'entrées et de sorties)). Ce numéro permet de démarrer plusieurs instances de la même tâche,
- Status est l'état de la tâche transformationnelle (ACTIVE, SUSPENDED, FINISHED, KILLED et READY). Ces états sont utilisés par l'ordonnanceur. Mais cette information est aussi disponible à l'interface réactive. En effet, le noyau synchrone peut demander à connaître l'état de la tâche et en particulier, il peut vouloir savoir si une tâche est terminée. Les états correspondent à :
  - ACTIVE : la tâche transformationnelle utilise le processeur,

- READY : la tâche peut s'exécuter mais n'a pas le processeur,
- SUSPENDED : la tâche a été suspendue par la commande Saturne SUSPEND nom_tâche,
- FINISHED : la tâche est terminée,
- KILLED : la tâche a été détruite par une commande Saturne STOP nom_tâche.

- Li et stack sont respectivement le "local identifier" CHORUS ainsi que la pile associée à l'activité exécutant la tâche transformationnelle.

A chaque type de tâche est associé un objet dérivé de l'objet générique *task* qui spécifie les paramètres d'entrée/sortie et le code de la tâche. C'est ce type dérivé qui est instancié lors de la création d'une tâche. Dans cet objet il n'existe pas de méthode consult() ou stop(). En fait, lorsque l'interface réactive désire consulter des résultats, cette consultation est immédiate puisque toutes les activités partagent le même espace d'adressage.

5.6.2 Protocole entre l'AM et le noyau synchrone

Les interactions entre l'interface réactive et le noyau synchrone sont de trois types :

- Le noyau synchrone peut créer, suspendre, réactiver ou détruire une tâche. Dans ce cas, l'interface réactive appelle la méthode de la classe *task* qui correspond. Ces méthodes sont synchrones et ne fournissent pas de résultat au retour,

- Le noyau peut aussi obtenir des informations sur les tâches. Pour faciliter la réalisation de ce type de commande, l'interface réactive et les tâches transformationnelles sont placées dans le même acteur. En effet, les tâches et l'interface réactive partageant le même espace d'adressage, l'interface réactive peut directement lire les données mises à jour par les tâches transformationnelles. C'est ainsi que l'on réalise les commandes Saturne CONSULT et KILL. Un certain nombre d'autres commandes que nous avons mises en place (qui permettent, entre autre, la consultation de l'état d'une tâche) sont réalisées grâce à la même technique,

- Enfin, la dernière interaction entre l'AM et le noyau concerne la terminaison des tâches transformationnelles. Quand une tâche transformationnelle se termine, celle-ci met à jour son attribut Status. Lorsque le noyau synchrone a terminé d'envoyer ses commandes à l'interface réactive, il lui demande si certaines tâches sont terminées. L'interface réactive consulte alors les instances

des tâches, puis renvoie les numéros des tâches dont l'état est FINISHED. L'instance est supprimée lorsque le noyau, une fois averti de la terminaison de la tâche, a consulté les résultats calculés.

5.7 Implantation du modèle Saturne sur COOL

5.7.1 Les objets COOL de Saturne

Jusqu'à présent, nous avons décrit la structure du sous-système Saturne sans pour autant décrire la manière dont nous utilisions COOL pour la mettre en oeuvre. C'est l'objet de ce paragraphe.

Une application COOL est constituée d'objets serveurs et d'objets clients. Nous modélisons donc le sous-système Saturne comme un ensemble d'objets. Dans Saturne, on trouve deux types d'objets COOL :

```
♯include <COMPLEX.idl>
interface callIR {
	void start-T1(out long identificateur, in long p1, in long p2);
	void start-T2(out long identificateur, in COMPLEX p1,
		in COMPLEX p2, in COMPLEX p3);
	void consult-T1(in long identificateur,out long p1);
	void consult-T2(in long identificateur,out COMPLEX p1);
	void kill(in long identificateur);
	void suspend(in long identificateur);
	void resume(in long identificateur);
	boolean isFinished(in long identificateur);
	boolean isSuspended(in long identificateur);
	boolean isActive(in long identificateur);
};
```

Figure 5.5: L'interface IDL d'un acteur AM

```
♯include "HAUTEUR.idl"
interface NoyauInput {
	oneway void receiveMessage-APPUI-HOMING();
	oneway void receiveMessage-MODIFICATION-HS(in HAUTEUR data);
	oneway void receiveMessage-SELECTION-FORCAGEMER-OFF();
	oneway void receiveMessage-SELECTION-FORCAGEMER-ON();
};
```

Figure 5.6: L'interface IDL d'un noyau synchrone

- Les objets de type “interface réactive”. Dans chaque AM on trouve un objet de ce type. Il permet au noyau synchrone d'exécuter les commandes Saturne sur les tâches transformationnelles. Un exemple de code IDL est donné dans la figure 5.5. On y retrouve toutes les commandes Saturne définies dans le chapitre trois,

- Les objets de type “objets synchrones”. Chaque objet de ce type encapsule un automate Esterel. Ces "objets synchrones" sont constitués des éléments que l'on a vu dans la figure 5.3 : c'est à dire des files de réception et d'émission.

Chaque objet synchrone exporte une interface IDL qui comprend tous les signaux que les autres noyaux peuvent lui envoyer. Chaque méthode IDL correspond au dépot d'un signal. L'invocation d'une de ces méthodes place un signal dans la file de réception du noyau destinataire. Un noyau a qui souhaite envoyer un signal *SIG* au noyau b invoquera la méthode *receiveMessage-SIG*() du noyau b. La figure 5.6 nous donne un exemple d'IDL d'un objet synchrone.

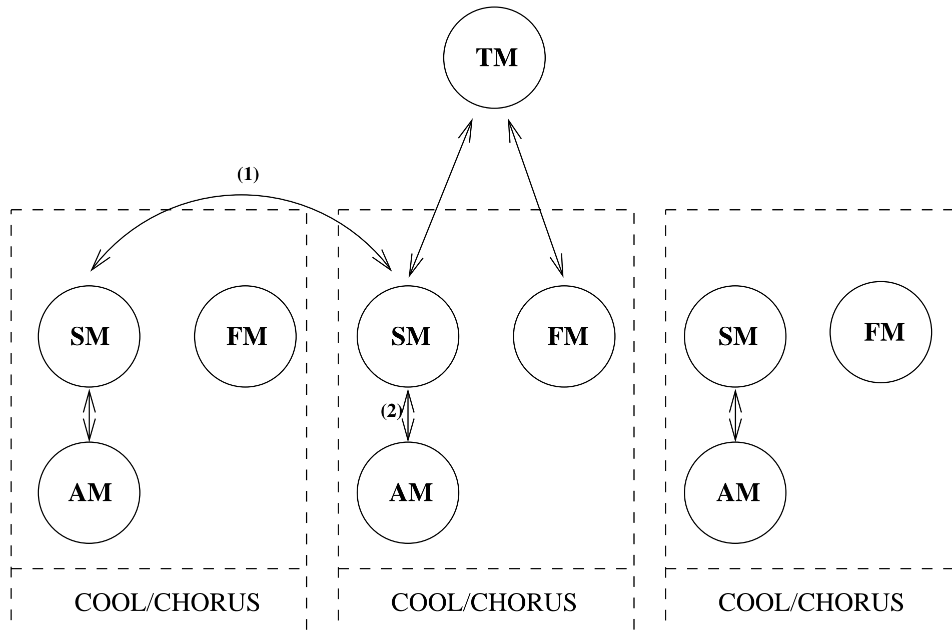

Figure 5.7: Communications du sous-système Saturne réalisées en COOL

5.7.2 Modes de communication utilisés pour les invocations de méthodes

La figure 5.7 représente l'architecture Saturne. Dans cette architecture, seules les communications issues de l'applicatif sont faites en COOL. Toutes les communications concernant les acteurs systèmes du sous-système Saturne sont développées en IPC CHORUS. Les communications entre applicatifs (flèches (1) et (2)) réalisées par COOL sont :

1. Les échanges de signaux Esterel entre les noyaux synchrones : c'est une communication point à point. Elle est effectuée de manière asynchrone. On utilise un appel de méthode avec l'attribut IDL "*oneway*". L'appel de méthode

"*oneway*" est justifié par l'augmentation du parallélisme entre les différentes communications qu'il permet,

2. Les échanges entre interface réactive et le noyau synchrone : c'est une communication point à point synchrone. On utilise donc un appel de méthode CORBA synchrone. Cette communication est synchrone car l'exécution des commandes Saturne doit être faite dans l'instant d'exécution de la grappe Saturne. De plus, certaines méthodes renvoient des résultats (c'est le cas des commandes Saturne CONSULT).

5.8 Conclusion

Nous avons décrit l'architecture du sous système Saturne sur CHORUS/COOL. Le chapitre suivant présente les prototypes qui ont servis à tester cette architecture. La réalisation des différents prototypes Saturne sur CHORUS a été faite en trois étapes :

- Développement d'un premier prototype incluant le modèle d'exécution Saturne avec les temps de latence mais sans la partie tolérance aux pannes. Ce premier prototype est validé par un deuxième prototype qui utilise une application Saturne réelle fournie par la société Dassault Aviation,
- Ajout au premier prototype de mécanismes de redondance active,
- Ajout de mécanismes de points de reprise.

Cette réalisation en trois étapes est accompagnée par un bilan et des performances sur chaque prototype.

Chapitre 6

Performances et résultats des prototypes réalisés

Dans ce dernier chapitre, nous allons décrire les différents prototypes réalisés. Nous détaillerons les objectifs de chacun et dirons si nous les avons atteints. Puis, nous donnerons quelques chiffres issus d'une étude de performances de trois de ces prototypes. Nous tenterons de les analyser pour en dégager le comportement de nos implantations et voir en quoi CHORUS et COOL sont adaptés ou non à une mise en oeuvre de Saturne. Cette partie conclura cette étude.

6.1 Le premier prototype

6.1.1 Architecture du prototype

L'architecture du premier prototype est simple. Celui-ci est constitué de trois noyaux : *Pepin*, *Noyau* et *Graines*. Les noyaux de cette application ainsi que les signaux qu'ils s'échangent sont représentés dans la figure 6.1. Chaque cercle correspond à un noyau synchrone. Les arcs représentent les signaux. Ils portent un nom et sont orientés. L'orientation des arcs correspond aux sens des communications entre les noyaux. Si un signal transporte une valeur, le type de cette valeur est spécifié entre parenthèses après le nom du signal.

* **Le noyau *Pepin***

Ce noyau manipule un type utilisateur qui est la représentation des nombres complexes. Une seule tâche transformationnelle est contrôlée par ce noyau synchrone.

* **Le noyau *Noyau***

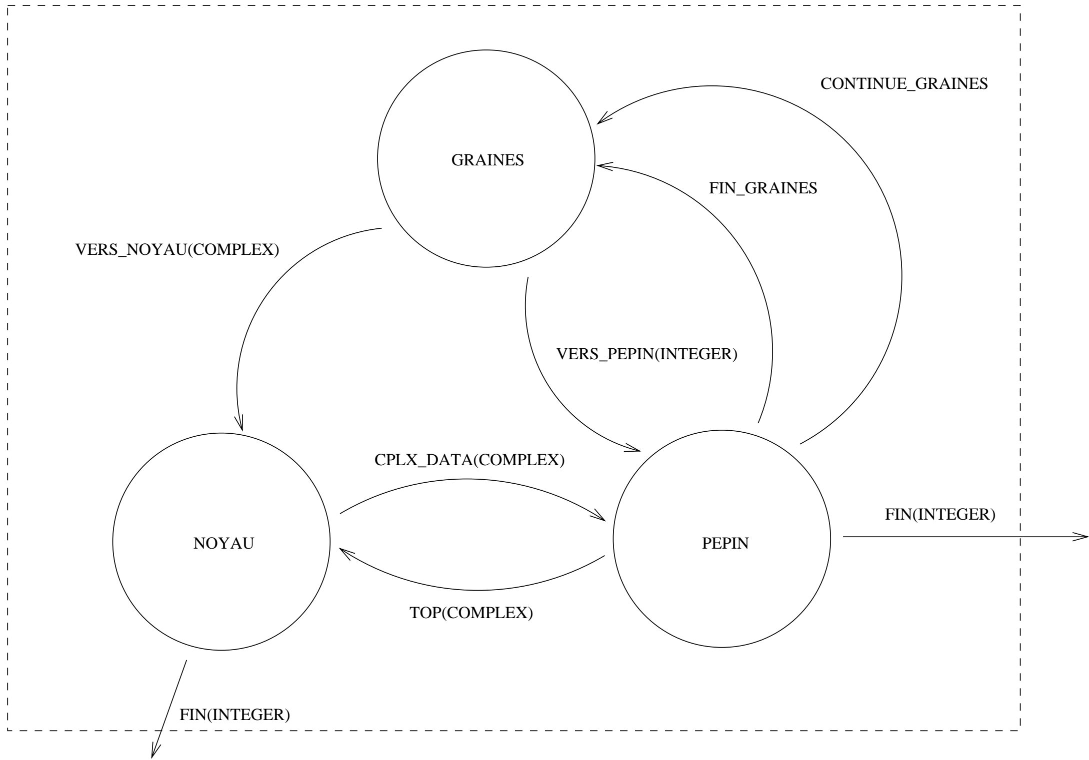

Figure 6.1: Le premier prototype

Ce noyau est constitué de six tâches Esterel s'exécutant en parallèle. Il contrôle trois tâches transformationnelles.

*** Le noyau *Graines***

Ce dernier noyau est le plus simple des trois. Il ne gère aucune tâche transformationnelle.

Les sources Esterel de ces noyaux sont consultables dans l'annexe B.

*** Description du fonctionnement de l'application Esterel**

Les deux noyaux synchrones *Noyau* et *Pepin* s'échangent des signaux transportant un nombre complexe. Ces signaux sont TOP et CPLX_DATA. Ils affichent tous les deux la fin de leurs tâches transformationnelles grâce à l'envoi d'un signal FIN vers l'environnement extérieur. Ce signal FIN comporte l'identifiant de la tâche terminée. *Noyau* démarre quatre tâches dès le début de son exécution (Deux tâches T1, puis une tâche T2 et enfin un tâche T3). A la terminaison de sa tâche T2, il envoie le résultat de calcul de cette tâche vers *Pepin* grâce au signal CPLX_DATA. *Pepin* est inactif jusqu'à la réception de ce signal. Lorsqu'il reçoit le signal CPLX_DATA, il démarre lui aussi une tâche T2, attend que celle-ci se termine, puis renvoie le résultat à *Noyau*. *Graines*, quant à lui, diffuse à chaque top un entier vers *Pepin* et un nombre complexe vers *Noyau*. Il cesse ses émissions quand *Pepin* reçoit le signal CPLX_DATA.

6.1.2 Objectifs visés

L'objectif de ce premier prototype est de fournir un modèle d'exécution de Saturne simplifié. Il ne comprend pas de mécanisme de tolérance aux pannes et doit permettre la communication entre les différents noyaux Esterel ainsi que la communication entre les noyaux synchrones et les grappes de tâches transformationnelles. Le prototype doit aussi fournir les mécanismes permettant la gestion des tâches transformationnelles. L'ordonnancement de ces tâches a été choisi arbitrairement. Les tâches ne présentant pas de contraintes temporelles, un algorithme de type round-robin suffit pour ces démonstrations. Toutefois, l'acteur AM est capable d'intégrer n'importe quel algorithme d'ordonnancement. Enfin, dans ce prototype, les temps de latence sont pris en compte bien qu'ils ne soient pas utilisés. Ce premier prototype a donc permis de tester tous les mécanismes de base de Saturne. Les tests ont montrés que le comportement du prototype correspondait avec ceux prévus dans le modèle Saturne. Il n'a pas posé de réel problème technique lors de sa réalisation mais c'est le prototype qui demanda le plus d'effort de développement car il

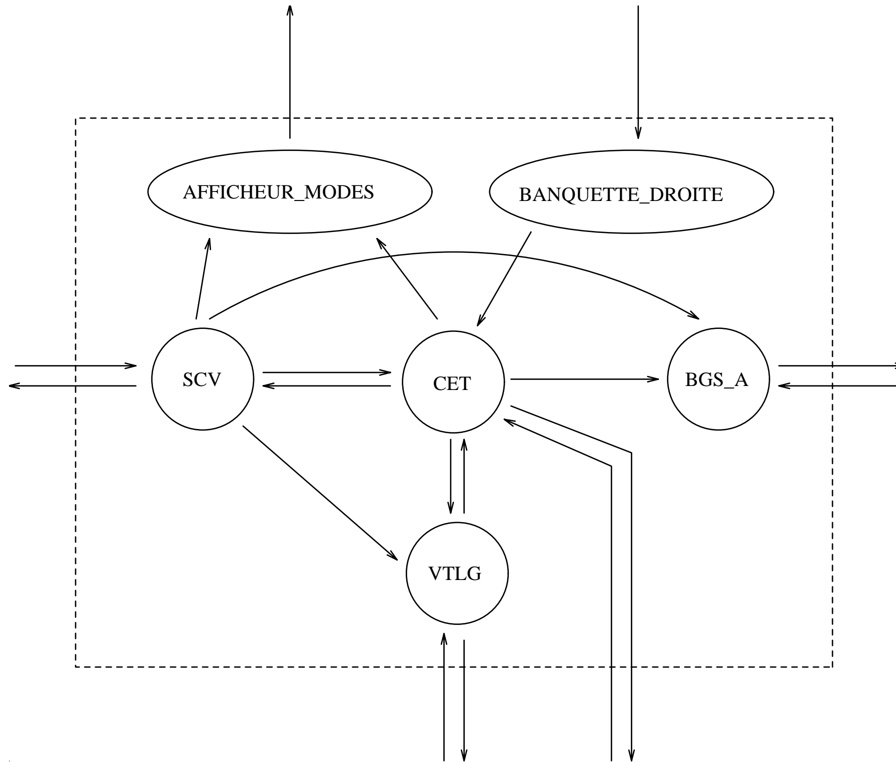

Figure 6.2: L'application de suivi de terrain

constitue l'ossature qui est réutilisée par les autres prototypes. Toutefois, pour une validation plus sérieuse, l'utilisation d'une application existante a été décidée : ceci constitue le deuxième prototype.

6.2 L'application de suivi de terrain

6.2.1 Architecture du prototype

Cette nouvelle application est constituée de six noyaux Esterel qui communiquent entre eux par le biais de trente sept signaux. Sa représentation graphique est donnée par la figure 6.2. Cette application interagit avec l'environnement extérieur grâce à soixante cinq signaux de sortie et trente huit signaux d'entrée. Les signaux entre l'application Esterel et l'environnement extérieur permettent de faire fonctionner une interface homme machine[1] réalisée par Eric Nassor, ingénieur chez Dassault Aviation (voir en annexe C). Les interactions entre l'ihm et les noyaux Esterel sont

[1]Nous parlerons dans la suite d'*ihm* pour désigner une interface homme machine.

constituées :

- Par des ordres partant de l'ihm vers les noyaux Esterel,
- Par l'envoi d'informations des noyaux Esterel vers l'ihm. Ces informations décrivent l'état des noyaux synchrones.

Il faut préciser que cette application avait déjà fonctionné dans un environnement centralisé : les noyaux Esterel étaient alors compilés en "synchrone fort". La compilation en synchrone fort consiste à combiner tous les noyaux Esterel en un seul. Le compilateur Esterel ne génère alors qu'un seul source C. Un premier découpage de cette application a ensuite été réalisé dans un environnement Unix par Christine Ledey, ingénieur chez Dassault Aviation. Cette nouvelle implantation consiste en une mise en oeuvre dans un véritable environnement temps réel distribué.

6.2.2 Génération automatique du code

La mise en oeuvre d'une application réelle, contrairement à une application comme celle du premier prototype, ajoute un certain nombre de contraintes. La contrainte la plus importante est une contrainte d'échelle. En effet, dans le premier prototype il n'y a que huit signaux. Une application comme celle du suivi de terrain en comporte cent quarante. Il n'est donc pas envisageable d'écrire à la main le code C++ effectuant les communications entre les noyaux et l'environnement extérieur. Dans le cas du premier prototype, le code spécifique aux noyaux et aux signaux de l'application Saturne représente un total de 2575 lignes alors que pour le deuxième prototype, il représente 16467 lignes. La quantité de code spécifique à l'application Saturne est donc importante.

description::= *module* nomApplication *signaux instances end module ;*

signaux ::= *unSignal* | *unSignal* signaux
unSignal ::= *sensSignal* identificateur ;
sensSignal ::= *input* | *output* | *signal*

instances ::= *uneInstance* | *uneInstance instances*
uneInstance ::= *instance* identificateur : identificateur ; *renommages*
renommages ::= *unRenommage* | *unRenommage renommages*
unRenommage ::= *sensSignal* identificateur $\Leftarrow$ identificateur ;

Figure 6.3: La grammaire en BNF utilisée par le générateur de code

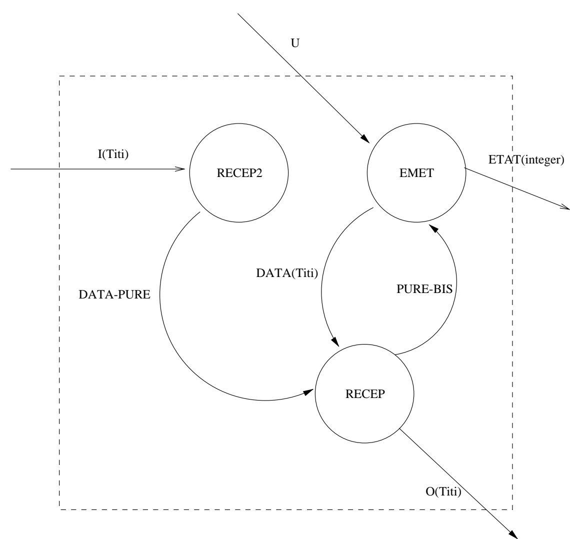

Figure 6.4: Une application Saturne

```
module envoi:

        input I(Titi);
        input U;
        output ETAT(integer);
        output O(Titi);
        signal DATA(Titi);
        signal DATA-PURE;
        signal PURE-BIS;

instance emet : emetteur;
        output DATA(Titi) ⇐ DATA;
        output ETAT(integer) ⇐ ETAT;
        input PURE-BIS ⇐ PURE-BIS;
        input U ⇐ U;

instance recep : recepteur;
        input DATA(Titi) ⇐ DATA;
        input DATA-PURE ⇐ DATA-PURE;
        output PURE-BIS ⇐ PURE-BIS;
        output O(Titi) ⇐ O;

instance recep2 : recepteur2;
        input I(Titi) ⇐ I;
        output DATA-PURE ⇐ DATA-PURE;

end module;
```

Figure 6.5: Description d'une application Saturne

Une approche intéressante consiste à générer le code spécifique (source C ++ et IDL) à partir d'une description de l'application. La génération du code offre une indépendance de l'application Saturne avec l'environnement d'exécution. Le code Esterel peut fonctionner quelque soit le système de communication et le système d'exploitation. Il suffit de modifier le générateur de code pour qu'il puisse s'adapter aux différents systèmes d'exploitation. Ceci permet de faire évoluer l'environnement d'exécution de manière transparente pour les utilisateurs. On peut aisément concevoir que les utilisateurs puissent développer leurs applications dans des environnements de type Unix qui leur offrent tous les outils de mise aux points nécessaires. L'évolution vers un environnement temps réel s'effectue alors grâce à la génération d'un code différent. De plus, les concepteurs d'applications Saturne ne sont pas nécessairement formés à l'utilisation des outils de communications et du système d'exploitation sous jacent. La génération du code leur permet de se concentrer uniquement sur le code Esterel.

Le générateur de code utilise un fichier de description de l'application Saturne dont la grammaire est décrite dans la figure 6.3. Un exemple de description ainsi que les noyaux correspondants sont donnés dans les figures 6.4 et 6.5. Cette grammaire a été définie par Christine Ledey, ingénieur chez Dassault Aviation. Le concepteur d'application Saturne écrit son application Esterel, puis, la décrit dans un fichier comme celui de la figure 6.5. Il appelle ensuite le générateur avec ce fichier de description. Il obtient ainsi son code C++ prêt à être compilé et exécuté sur une machine CHORUS. La grammaire a été conçue pour permettre la connexion de plusieurs noyaux entre eux. On commence par citer grâce aux mots clefs *input, output et signal* **tous** les signaux de l'application. Le mot clef *input* (respectivement *ouput*) définit les signaux venant de (respectivement sortant vers) l'environnement extérieur. Le mot clef *signal* permet de déclarer les signaux qui vont servir à la communication entre les différents noyaux. On spécifie ensuite les noms des noyaux grâce au mot clef *instance*. Enfin, pour chaque instance, les signaux d'entrée et de sortie **associés au noyau** sont cités. L'application Saturne est en fait un assemblage de plusieurs de ces instances de noyau. Un renommage intervient alors au niveau des noms de signaux. Cette combinaison de noyau Esterel peut être comparée à l'application de l'opérateur de composition sur des processus CCS[2][37] : l'utilisateur définit un ensemble de noyaux type, puis, lorsqu'il les instancie pour construire son application, il renomme les signaux afin de réaliser les connections entre les différentes instances.

[2]CCS pour **C**alculus of **C**ommunicating **S**ystems.

6.2.3 Objectifs visés

Les objectifs de ce prototype étaient de voir si le sous-système Saturne développé dans le premier prototype permettait d'exécuter une application réelle. Le développement du prototype deux nécessita un mois de travail. Toutefois, le développement consista uniquement à écrire et tester le générateur de code ainsi que réaliser la communication entre l'ihm qui s'exécutait sur une station SunOS et l'application Saturne qui s'exécutait sur CHORUS. Le sous-système Saturne écrit lors du premier prototype fut intégralement récupéré, et ce sans modification. La taille de l'application de suivi de terrain, en dehors du fait qu'elle nous a obligé à implanter un générateur de code, a permis de mettre à jour un autre problème. L'application, une fois compilée représentait environ 3,5 Méga octets d'exécutable pour une dizaine d'acteurs auxquels il faut ajouter les acteurs de COOL. Si l'on exécute une application de cette taille sur une machine unique de type Intel 486 SX cadencé à 25 Mhz avec 9 Méga octets de mémoire, on observe des problèmes de performances. En effet, dans ce cas de figure, il était impossible d'activer les noyaux plus de quatre fois par seconde. Si nous descendions au delà d'un top d'horloge tous les 250 milli secondes[3], l'application adoptait un comportement anormal du à des pannes temporelles de certains noyaux synchrones. Rappelons qu'un noyau engendre une panne temporelle lorsque son temps d'exécution dépasse la durée d'un intervalle entre deux tops. Dans une application synchrone, ce type de panne désynchronise les noyaux entre eux, ce qui engendre un comportement indéterminé.

6.3 Le troisième prototype

Le troisième prototype comporte les mêmes noyaux Esterel que le premier prototype. L'objectif de ce nouveau prototype est de tester la possibilité d'utiliser de la redondance active sur les noyaux synchrones. Un mécanisme de vote est implanté sur la réception des signaux Esterel. Les groupes de signaux sont réalisés à l'aide de groupes d'objets COOL. La figure numéro 6.6 décrit l'application Esterel et les groupes d'objets COOL utilisés. Le noyau *Pepin* est tripliqué et le noyau *Graines* est dupliqué. Ici, on applique un double vote sur les signaux Esterel. L'émission d'un signal vers un groupe de noyaux Esterel est réalisée par un appel de méthode sur un groupe d'objets COOL. Si le noyau émetteur fait lui même partie d'un groupe de n_e objets, chaque noyau récepteur reçoit alors n_e messages. Le premier vote est donc effectué sur ces n_e messages reçus. Le deuxième vote est aussi réalisé par les noyaux récepteurs. Quand les noyaux récepteurs ont effectués le vote sur leurs n_e messages, ils s'échangent le message qu'ils considèrent comme juste et effectuent un

[3] A titre de comparaison, la réalisation de Christine Ledey était cadencée à 50Hz.

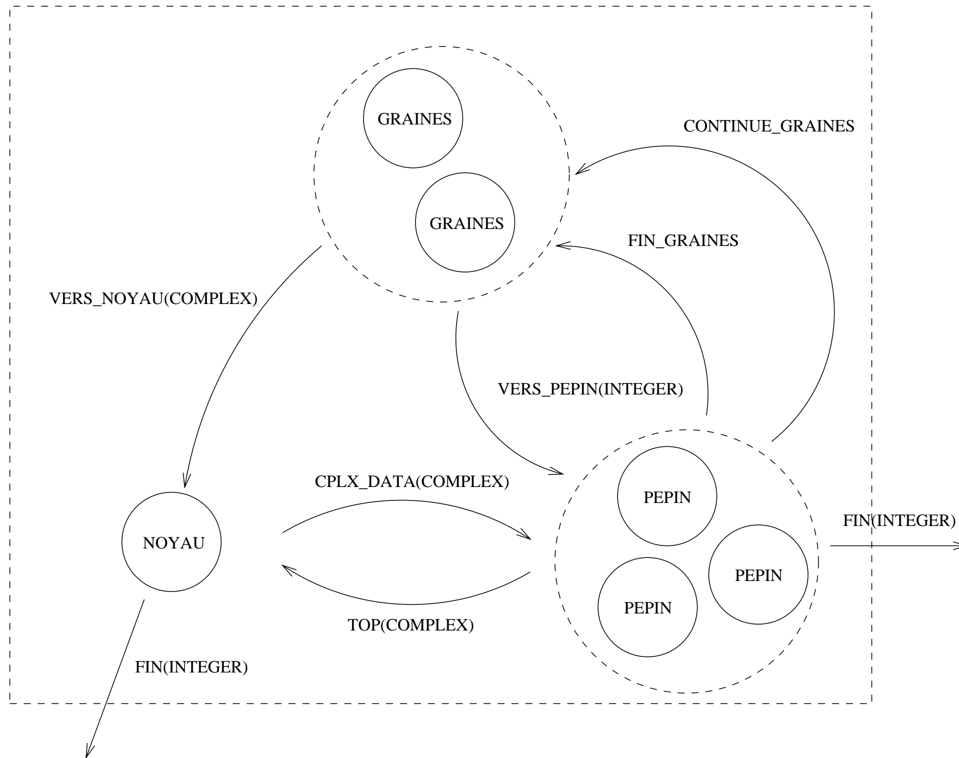

Figure 6.6: L'application Esterel utilisée pour la mise en oeuvre du vote

deuxième vote. Si le groupe des récepteurs contient n_r noyaux synchrones, alors ils reçoivent chacun $n_r - 1$ messages sur lesquels ils effectueront un vote. Ceci constitue ce que nous avons appelé le deuxième niveau de vote. La figure 6.7 illustre ces échanges de messages : les flèches (1) correspondent aux messages utilisés pour le premier vote, les flèches (2) pour le deuxième vote. Les noyaux émetteurs sont les *Pepins*, les récepteurs sont les *Graines*. Ce mécanisme de double vote constitue un double contrôle qui permet :

- De détecter la panne d'un noyau émetteur ou récepteur plus rapidement,
- De garantir que les noyaux récepteurs calculeront leur transition avec la même valeur du signal d'entrée.

Ce mécanisme de redondance active est totalement transparent pour les noyaux qui émettent les signaux. En effet, un noyau qui effectue une émission invoque une méthode dont il ne sait pas si elle fait partie d'un seul objet ou d'un groupe d'objets COOL.

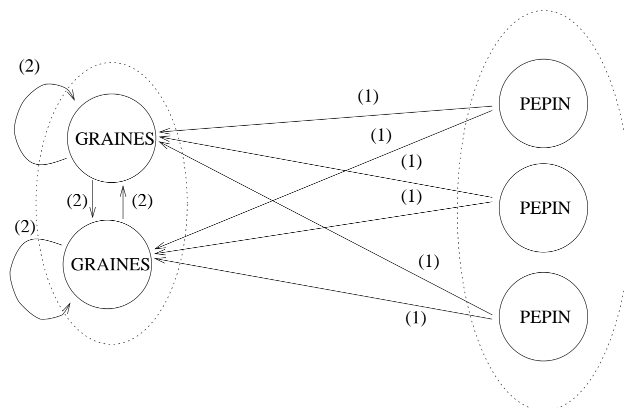

Figure 6.7: Le mécanisme de double vote

Les problèmes auxquels nous avons été confrontés pour ce prototype concernent l'invocation sur groupe de COOL. La version de COOL que nous avons utilisée (COOL V3.R0b) n'est pas adaptée **à nos besoins de tolérance aux pannes**. Dans cette version, il est impossible de stopper brutalement un objet sans entraîner une panne du groupManager ou un blocage des objets qui étaient en communication avec l'objet fautif. Dans le cadre de nos expérimentations, nous avons simulé la panne par un arrêt normal de l'acteur : COOL permet l'ajout ou la suppression dynamique de serveurs dans un groupe d'objets. Ajoutons que COOL ne permet pas aux utilisateurs de spécifier de nouvelles politiques d'invocation. Dans notre cas, un protocole de diffusion atomique nous aurait été nécessaire. Enfin, il est actuellement impossible d'utiliser des invocations *"oneway"* sur des groupes d'objets.

6.4 Le dernier prototype

Le prototype de Saturne sur CHORUS/COOL que nous allons décrire maintenant constitue la dernière mise en oeuvre effectuée durant cette étude. Après l'utilisation de la redondance active pour augmenter le niveau de fiabilité d'une application Saturne, nous avons tenté d'associer de la redondance active avec des points de reprise. Cette association permet de recouvrir des pannes de noyaux synchrones et de sites grâce à l'utilisation de sites non actifs au démarrage du système. Ces mécanismes permettent de maintenir un nombre identique de réplique dans un groupe même en cas de défaillance.

```
application proto1:

    kernel pepin 3 SM1;
    kernel noyau 1 SM2;
    kernel graines 2 SM3;

    instance pepin interPepin1;
    instance pepin interPepin2;
    instance pepin interPepin3;
    instance noyau interNoyau;
    instance graines interg1;
    instance graines interg2;

    mode pepin 1 m0 2116;
    mode noyau 1 m0 2116;
    mode graines 1 m0 2116;
    mode graines 1 m1 2116;
    ⋮
    mode graines 2 m1 2116;

    log pepin 1 p1log;
    log noyau 1 n1log;
    log graines 1 g1log;
end;
```

Figure 6.8: Exemple d'un fichier de configuration

Pour conserver un nombre identique de réplique dans les groupes de noyaux Esterel, les concepteurs d'applications Saturne définissent un fichier de configuration qui stipule comment est utilisée la redondance active. La figure 6.8 donne un exemple de ces fichiers. Ici, on y stipule le nombre de réplique de chacun des noyaux *Pepin, Noyau* et *Graines*. On déclare aussi les modes d'exécutions. Ces modes définissent sur quelles machines s'exécutent les différentes répliques d'un noyau synchrone. L'occurrence de pannes ou la remise en état d'un site fait passer l'application d'un mode à un autre. Toutes ces informations sont contrôlées par les FM. Le FM intègre aussi le lancement des répliques, les changements de mode et tous les services offerts pour les mécanismes de tolérance aux pannes. Seul les services de détection de pannes d'un noyau synchrone et les services de journalisation sont implantés dans les SM. Enfin, le fichier de configuration spécifie aussi avec le mot clef *log* quels sont les noyaux synchrones qui effectuent les journalisations sur fichier des points de re-

prise. Il faut préciser ici que faute de temps, tous les tests n'ont pas été effectués sur ce prototype. Une description détaillée de tous ces mécanismes ainsi que le bilan de ce prototype peuvent être consultés dans [53].

6.5 Mesure des performances des prototypes

6.5.1 Performances des prototypes

Cette étude se termine par l'évaluation des différents prototypes réalisés. Ici, notre objectif n'est pas la recherche des valeurs optimales permettant d'exécuter le plus rapidement possible une application Saturne. Nous ne nous intéressons pas aux résultats numériques mais plutôt au comportement des prototypes lorsque les différents paramètres d'une application Saturne sont modifiés. Cette analyse a pour but de tenter de mettre en évidence le fonctionnement du modèle Saturne sur CHORUS et surtout de vérifier que ce fonctionnement reste conforme à celui énoncé par le modèle Saturne du CERT ONERA. Cette évaluation porte sur quatre aspects. Le premier aspect constitue un point important dans les applications temps réel : c'est le déterminisme des temps de réponse. Nous avons utilisé le sous-système COOL pour la réalisation des différents prototypes afin d'évaluer les possibilités de développement d'application temps réel à l'aide de technologies de type CORBA. Dans cette optique, nous devons vérifier que l'appel de méthode COOL est déterministe. Les deux aspects suivants constituent une observation du prototype face aux variations des deux paramètres importants : la durée d'un intervalle entre deux tops d'horloge et les quantums d'ordonnancement des tâches transformationnelles. Enfin, les derniers tests de performances consistent à étudier l'impact de la répartition d'une application Saturne. Les chiffres présentés ci dessous ont été obtenus sur un réseau de quatre PC sous ClassiX R2.3 et COOL V3r0b. Les quatre PC ne sont pas sur le même réseau Ethernet. En effet, deux d'entre eux (Astérix et Idéfix) sont situés sur un même réseau Ethernet. Les deux autres machines (Falbala et Obélix) sont sur un réseau Ethernet différent. Les deux réseaux Ethernet sont connectés par un pont. Les machines Astérix et Idéfix sont deux Intel 486 cadencés à 33 Mhz avec 8 Méga octets de mémoire vive. Obélix et Falbala sont deux Intel 486 cadencés à 66 Mhz avec 16 Méga octets de mémoire vive. **De part la différence de puissance des quatre machines et l'architecture réseau qui les connecte, les résultats de cette étude doivent être considérés avec prudence.**

* **Etude du déterminisme**

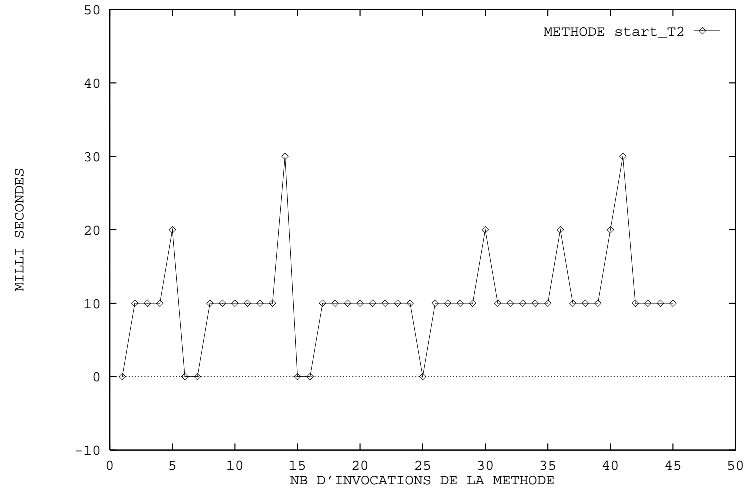

Figure 6.9: Temps d'exécution d'une méthode COOL

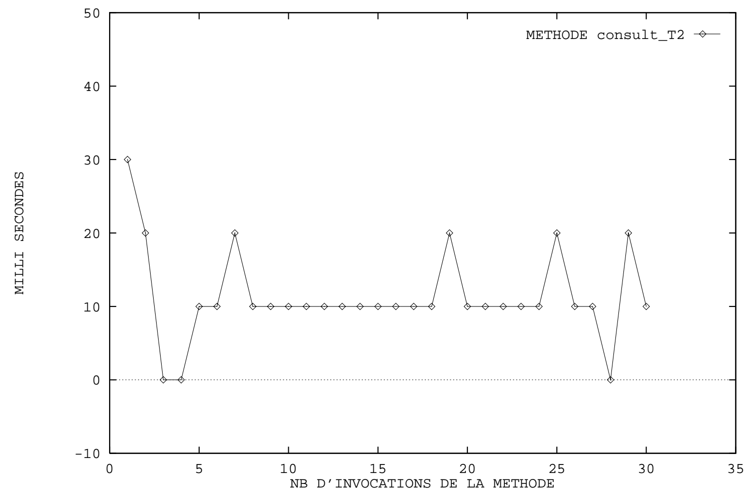

Figure 6.10: Temps d'exécution d'une méthode COOL

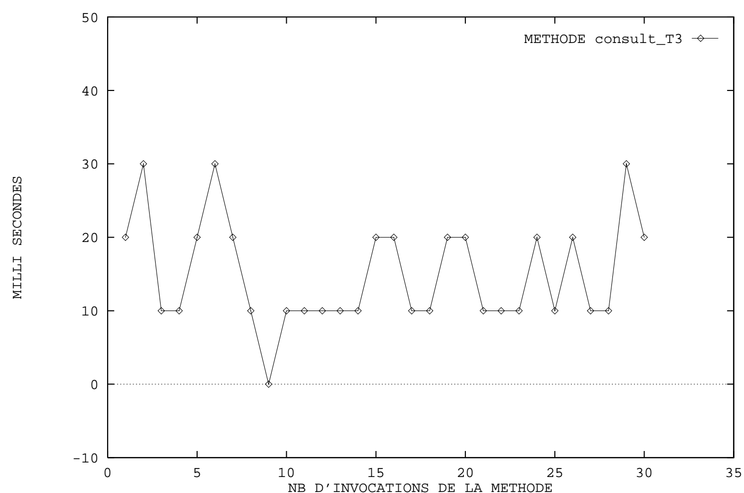

Figure 6.11: Temps d'exécution d'une méthode COOL

Les courbes 6.9, 6.10 et 6.11 nous montrent les temps de réponse d'invocations de méthodes COOL. Ces temps sont très différents d'une exécution à une autre : on peut avoir des écarts de 20 à 30 milli secondes[4]. Ces mesures ont été effectuées sur le prototype numéro un, dans un environnement centralisé (sur une machine Intel 486 cadencée à 33 Mhz avec 8 Méga octets de mémoire) afin de supprimer tout indéterminisme lié au support de communication physique qui est de type Ethernet. De nombreuses causes peuvent être responsables de ces écarts importants :

- L'indéterminisme des IPC CHORUS. Ce problème connu entraîne nécessairement des conséquences sur les communications par COOL puisque dans notre cas, les communications COOL s'appuient sur les IPC CHORUS,
- La présence de nombreuses allocations dynamiques dans les souches et les squelettes de COOL. Or, en fonction de l'état de la machine et de l'algorithme d'allocation mémoire utilisé, le temps nécessaire pour une allocation dynamique peut ne pas être identique d'une exécution à l'autre. La solution pourrait être apportée par le choix d'un algorithme d'allocation mémoire déterministe où toute la mémoire nécessaire durant l'exécution serait réservée lors du démarrage de l'application,
- Des problèmes de priorité. Dans le cadre d'application temps réel, il est nécessaire de maîtriser finement la priorité de chaque tâche qui s'exécute dans

[4]Il faut toutefois préciser que les outils de mesure utilisés ici, c'est à dire l'appel système *chorusTimeL* ne semble pas être d'une très grande précision sur les architectures Intel utilisées.

le système. **Or, lors d'une invocation sur un objet COOL, l'ORB ne récupère pas la priorité de l'activité cliente. Ainsi, l'activité qui traite l'invocation du client chez le serveur peut ne pas avoir la même priorité que le client. Si la priorité de cette activité est inférieure à celle du client, alors le traitement demandé par le client peut être intérompu par une autre activité du système, ce qui n'est pas satisfaisant si le client doit être très prioritaire.** Pour supprimer ce problème, COOL doit permettre de spécifier la priorité de l'activité CHORUS qui réalisera l'invocation chez le serveur,

- Le coût de l'allocation d'une activité CHORUS. Comme nous venons de le dire ci-dessus, COOL crée une activité CHORUS à chaque invocation. Or, la création d'une activité CHORUS est coûteuse. Elle peut aussi être indéterministe si la pile d'exécution de l'activité est allouée dynamiquement. **Pour des applications temps réel, il pourrait être intéressant de disposer d'un ensemble d'activités CHORUS qui sont allouées à l'initialisation de l'application et dédiées au traitement des invocations COOL.**

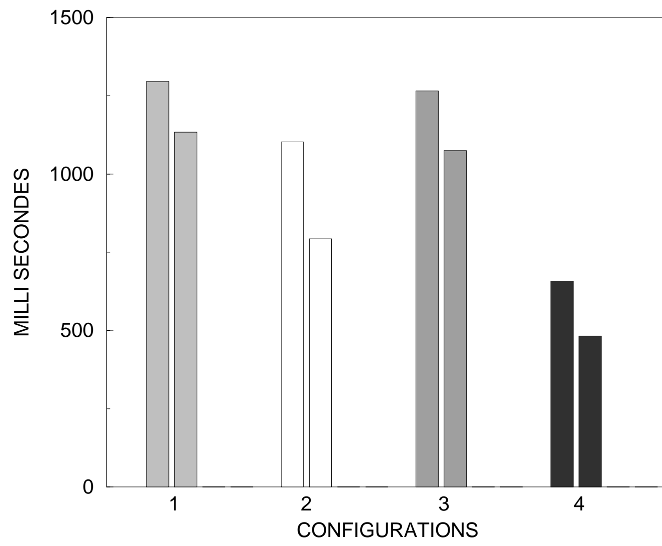

Figure 6.12: Temps de calcul des tâches T1 et T2

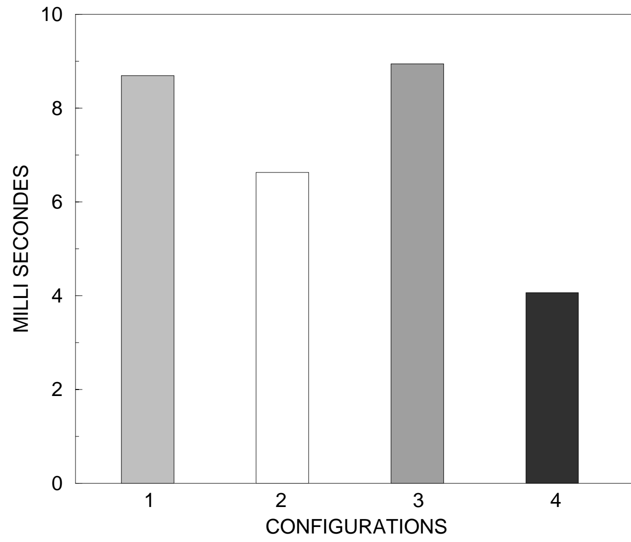

Figure 6.13: Temps de calcul des tâches T3

* **Etude de la répartition**

	Sites concernés			
Configurations	Obélix	Falbala	Astérix	Idefix
1	tout			
2	tous les SM	AM de *Noyau*	AM de *Pepin*	AM de *Graines* LM, TM
3	SM de *Pepin*	SM de *Noyau*	SM de *Graines* TM, LM	tous les AM
4	SM et AM de *Pepin*	SM et AM de *Noyau*	SM et AM de *Graines*	LM, TM

Tableau 6.1: Configurations utilisées pour la répartition du prototype 1

C'est l'impact de la répartition d'une application Saturne sur ses temps d'exécution qui nous intéresse ici. Aussi, nous avons considéré quatre configurations qui sont décrites dans la table 6.1. Les tests effectués pour ces mesures ont été réalisés sur le premier prototype.

Si l'on regarde les histogrammes 6.12 et 6.13, on constate que la configuration quatre est celle qui minimise le temps de calcul des tâches transformationnelles. Par contre, les configurations une et trois où tous les AM sont rassemblés sur un même site sont celles qui ont les plus mauvais résultats. L'explication de ce phénomène est donnée par la suite. Concernant les communications entre AM et SM, la configuration la plus adaptée est la configuration numéro quatre (voir l'histogramme 6.14).

La répartition des acteurs du sous-système Saturne possède toutes les caractéristiques, en terme de performances, d'une application parallèle effectuant des calculs et des communications. Dans le domaine du calcul parallèle, on utilise des d'algorithmes de placement statique de processus. La répartition des noyaux synchrones pourrait peut être bénéficier de ceux-ci. Il semble clair que la répartition des noyaux synchrones sur un réseau de machines dépende essentiellement de l'application Saturne elle même . Les facteurs à prendre en compte sont :

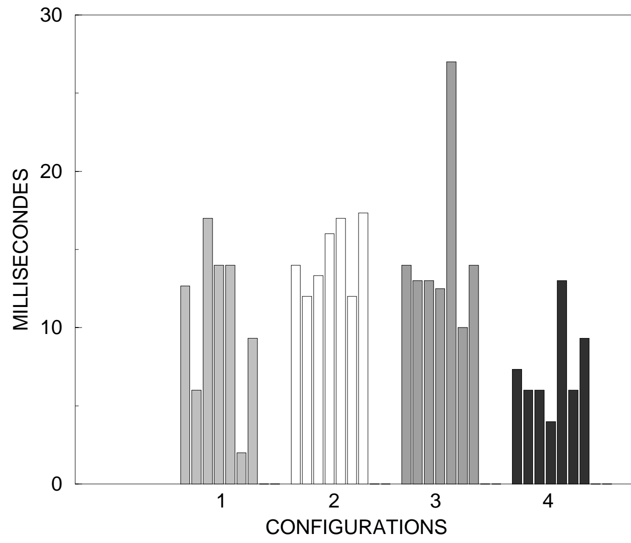

Figure 6.14: Communications entre les AM et les SM

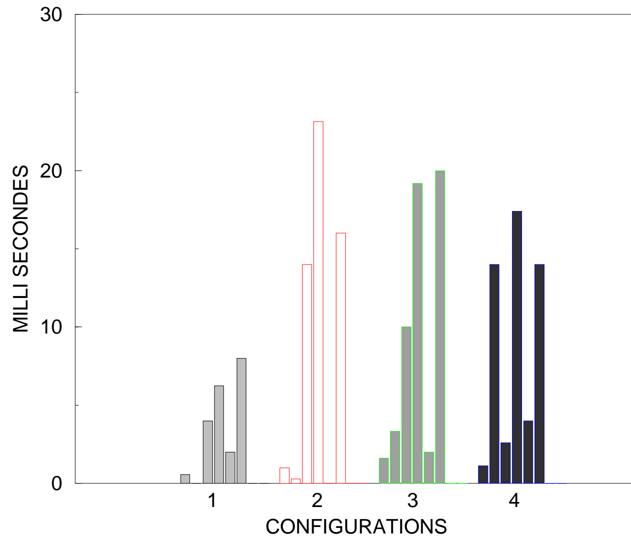

Figure 6.15: Communications entre les SM

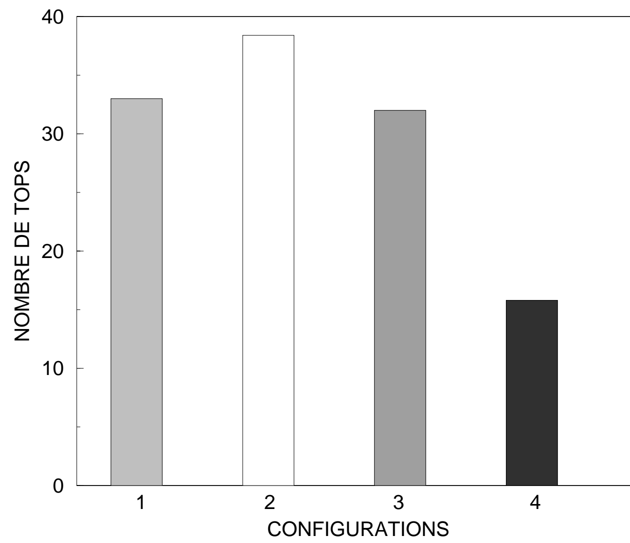

Figure 6.16: Nombre de tops pour exécuter l'application

1. La quantité de signaux échangés entre les noyaux. Si celle-ci est très grande, il est bien sûr important de regrouper sur un même site les noyaux qui s'échangent le plus de signaux. Dans ce cas, une configuration comme la configuration deux peut devenir intéressante,

2. La quantité et la taille des tâches transformationnelles. Si celles-ci sont très importantes, la configuration quatre est préférable car elle offre le plus de temps processeur aux tâches. Rappelons que les tâches transformationnelles consomment le temps processeur non utilisé par les noyaux synchrones entre deux tops. Ce qui explique l'efficacité de la configuration quatre. En effet, dans les quatre configurations, l'intervalle entre deux tops est identique mais la quantité des tâches sur un site est différente puisque la répartition des AM est différente. Dans la configuration quatre, il y a moins de concurrence entre les tâches puisqu'elles sont réparties. Ce qui fait diminuer leur temps total de calcul. Les configurations une et trois sont celles où les tâches transformationnelles disposent du moins de ressource processeur puisque dans ces configurations, toutes les tâches transformationnelles sont regroupées sur un même site,

3. L'importance de la communication entre AM et SM. Si le SM interagit beau-

coup avec le AM, en consultant très souvent des résultats de tâches par exemple, la configuration quatre est préférable puisque les communications entre AM et SM sont locales.

Ces trois points sont vérifiés par les histogrammes 6.12, 6.13, 6.14 et 6.15. Enfin dernière remarque concernant la durée en top de l'exécution d'une application Saturne. Si l'on regarde la figure 6.16, on constate que cet histogramme est très voisin avec les histogrammes 6.12 et 6.13. **Ceci s'explique par le fait qu'il existe une relation entre la durée de calcul des tâches et le temps total d'exécution d'une application Saturne.** Nous expliquerons cette relation dans le paragraphe traitant de la variation du quantum d'ordonnancement.

* Etude d'impact de la variation de l'intervalle entre deux tops

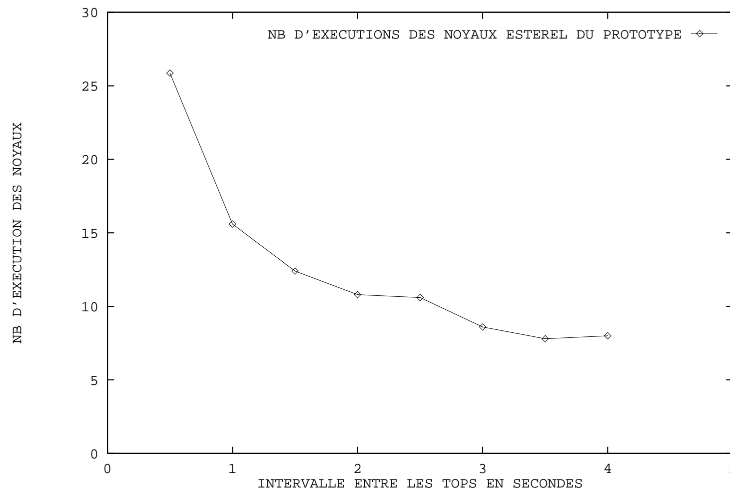

Figure 6.17: Nombre d'exécution des noyaux Esterel

La durée de l'intervalle entre deux tops est un facteur important du modèle Saturne. En effet, si cette valeur est très grande, l'application réagit moins rapidement aux événements extérieurs. Si par contre cette valeur est trop petite, certains noyaux risquent de voir leur exécution déborder sur le top suivant, ce qui amène l'application dans un état complètement incohérent. Les mesures effectuées ici ont deux objectifs : évaluer les conséquences de la variation de cet intervalle sur les tâches transformationnelles et le temps de calcul de l'application. Ces mesures ont été effectuées avec le premier prototype et avec la configuration quatre.

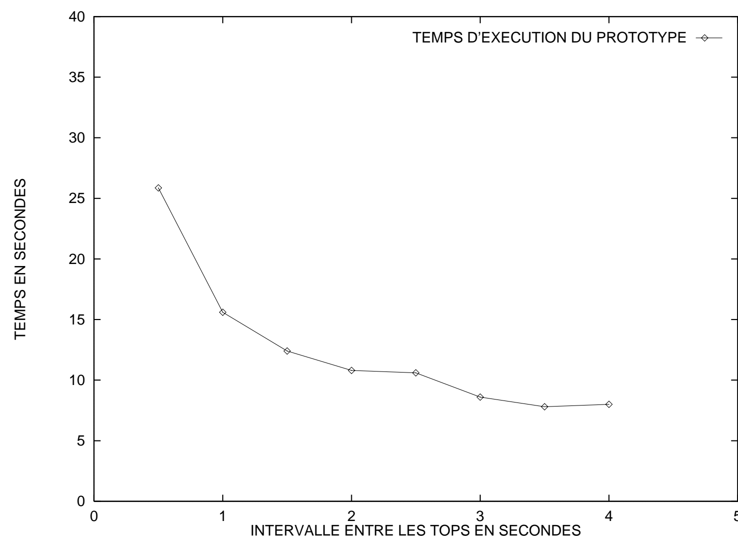

Figure 6.18: Temps d'exécution des noyaux Esterel

Si l'on regarde les courbes 6.17 et 6.18, on constate que plus les intervalles sont importants, et plus le temps d'exécution de l'application se raccourcit. La courbe 6.17 correspond à un temps logique en nombre de top, la courbe 6.18 correspondant à un temps physique en secondes. Ce phénomène pourrait s'expliquer par deux raisons :

- En fait, bien que les tâches transformationnelles s'exécutent de façon asynchrone, les noyaux sont dépendants de la terminaison des tâches. En effet, dans notre code Esterel, une fois que les noyaux ont lancé leurs tâches, ils attendent la fin de celles-ci, soit pour créer d'autres tâches, soit pour envoyer des signaux. En espaçant les tops, on laisse plus de temps processeur aux tâches et on diminue le nombre des changements de contexte d'activité CHORUS. Ce qui provoque une fin plus rapide des tâches sans pour autant changer leur temps total de calcul comme le montre la courbe 6.19. **Ce temps total de calcul des tâches, qui est quasi constant quelque soit la valeur des tops qui cadencent les noyaux Esterel, illustre bien que si les noyaux sont dépendants des tâches, les tâches elles, sont indépendantes. Elles s'exécutent de manière asynchrone par rapport au reste de l'application. Et en particulier elles s'exécutent indépendamment des tops et des noyaux synchrones,**

- La deuxième raison est qu'en espaçant les tops, le nombre d'événements (fin de tâches, réception d'un signal) par top augmente. En d'autres termes, à chaque

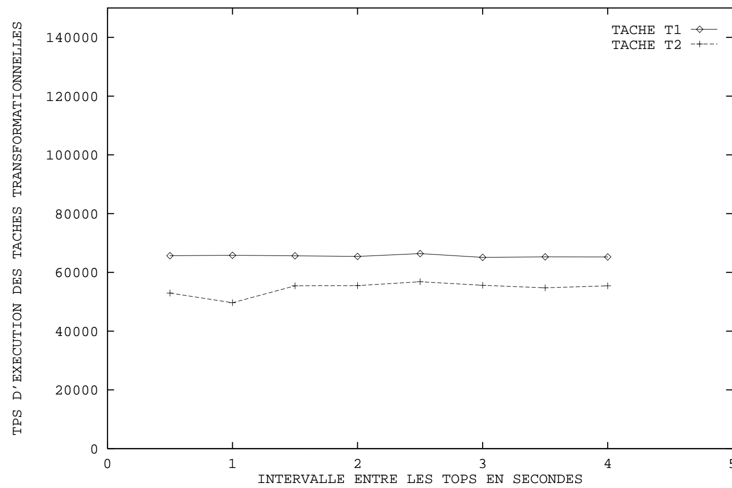

Figure 6.19: Temps de calcul des tâches en fonction de l'intervalle

top, les noyaux traitent plus d'événements. Le nombre d'événements entre chaque top atteint une limite qui correspond aux différentes synchronisations réalisées grâce aux signaux entre les noyaux synchrones. Ceci expliquerait que les courbes 6.17 et 6.18 tendent à se stabiliser avec des valeurs supérieures à 3. Ce deuxième point est certainement celui qui influence le plus l'allure de ces deux courbes.

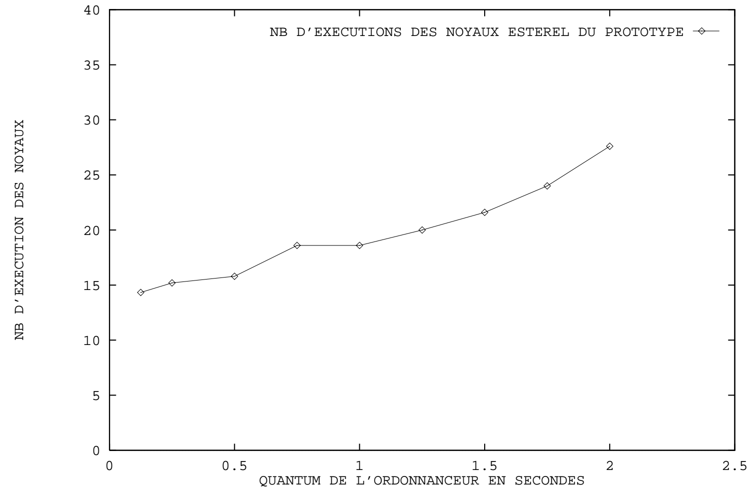

Figure 6.20: Nombre d'exécution des noyaux en fonction du quantum

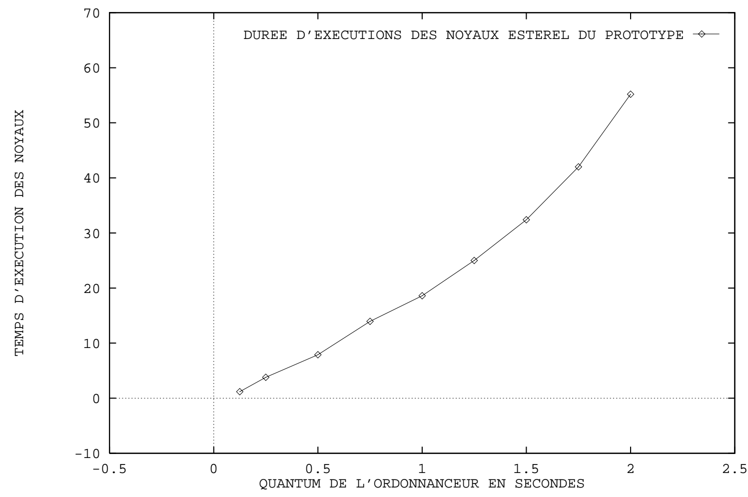

Figure 6.21: Temps d'exécution des noyaux en fonction du quantum

* **Variation du quantum d'ordonnancement**

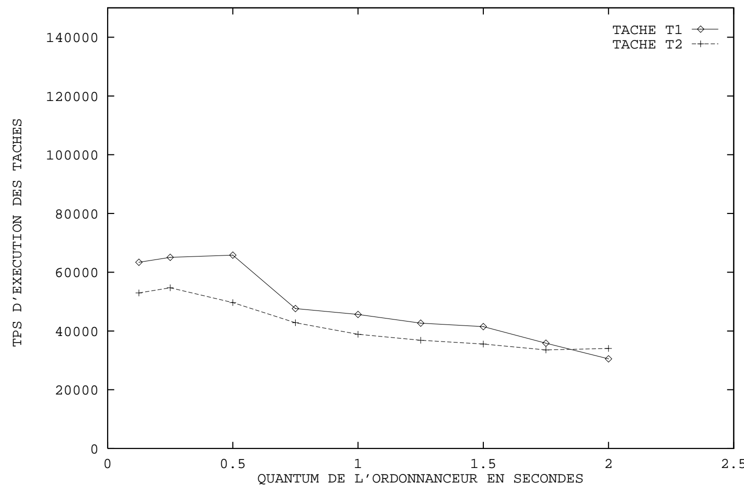

Figure 6.22: Temps de calcul des tâches en fonction du quantum

Ces mesures ont été effectuées sur le prototype un avec la configuration quatre. L'objectif est de voir si l'ordonnancement des tâches influence l'exécution de l'application.

Si l'on regarde la courbe 6.22, on constate que la durée de calcul des tâches diminue au fur et à mesure que le quantum augmente ce qui s'explique par le fait qu'il y a de moins en moins de changement de contexte dans notre ordonnanceur, en effet, cet aspect est comptabilisé dans la durée des tâches. Les courbes 6.20 et 6.21 donnent la durée d'exécution en temps logique (top d'horloge) et en temps physique de l'application Esterel. Or, plus le quantum est long, et plus la durée d'exécution augmente. Ceci s'explique peut être par les mauvaises performances de notre ordonnanceur. En effet, il a été développé uniquement pour offrir un support d'exécution à nos tâches transformationnelles. Nous n'avons pas cherché à l'optimiser. En particulier, lorsqu'une tâche se termine pendant un quantum, l'ordonnanceur n'attribue pas le reste du quantum à une autre tâche, ce qui pourrait expliquer l'augmentation de la durée de l'application. Les courbes 6.21 et 6.22 montrent également que les noyaux synchrones sont dépendants des tâches transformationnelles. Nous avons choisi arbitrairement un algorithme de round-robin pour nos expérimentations mais il est clair que l'algorithme qui sera utilisé dans une implantation réelle devra tenir compte de l'échéance des tâches, comme avec un algorithme de type EDF[5] par

[5] EDF pour **E**arliest **D**eadline **F**irst.

exemple.

* **Conclusion sur les mesures de performances**

Pour conclure sur ces mesures, on peut essentiellement dire que le comportement du sous-système Saturne sur CHORUS/COOL correspond sur bon nombres de points au modèle spécifié par le CERT ONERA. Toutefois, il faudrait effectuer une analyse plus poussée des prototypes avec les mécanismes de redondance. En effet, faute de temps il n'a pas été possible d'approfondir cet aspect. Enfin, sur l'étude de la répartition des noyaux synchrones, il faudrait confirmer les résultats énoncés avec de nouvelles mesures. Ces nouvelles mesures devront être effectuées sur des applications Esterel dont le comportement est différent : varier la quantité de tâches, la quantité de signaux, etc.

6.5.2 Techniques permettant d'améliorer les performances des prototypes

Dans la réalisation de nos différents prototypes, nous n'avons pas cherché à optimiser les performances. Un certain nombre de solutions qui restent à étudier devraient permettre d'optimiser ces résultats. On peut citer à titre d'exemple :

- La "factorisation" des émissions de signaux sur le réseau. Nous avons vu que le noyau Esterel appelle de lui même les fonctions de sorties. Dans ces fonctions de sorties on trouve les invocations à distance qui permettent de transférer les signaux d'un noyau à un autre. L'optimisation proposée consisterait à temporiser toutes ces émissions jusqu'à ce que le noyau ait terminé ses sorties, concaténer tous les messages destinés à un noyau donné en un seul message, puis transmettre cet unique message. En effet, lorsque l'on émet un message sur un réseau, très souvent, quelque soit la taille du message, il existe un temps de latence non négligeable, or, les signaux Esterel transportent peu, voir pas du tout d'information. Emettre sur le réseau chaque signal individuellement est donc très coûteux,

- Tenter de remplacer les diffusions utilisées pour la redondance et pour la transmission du top par d'autres mécanismes moins coûteux,

- Utiliser les services optimaux du système d'exploitation. Dans notre cas, ceci revient très souvent à utiliser des acteurs superviseurs. En effet, ces acteurs sont plus performants grâce :

- Au fait que la durée des traps est raccourcie car le noyau ne fait ni de SVC[6], ni de copie des arguments de l'espace utilisateur vers l'espace superviseur par un svCopyin (respectivement svCopyout pour le retour des résultats),
- Car il n'y a aucun aucun changement de contexte (ce n'est qu'un simple appel de fonction),
- Enfin parce que les IPC sont fortement améliorés.

Sur CHORUS l'optimisation des performances peut aussi être améliorée par les moyens suivants :

- Utiliser la constante K_NOSWAPOUT afin d'éviter la pagination d'une région (blocage de la région en mémoire principale)
- Utiliser la constante K_NODEMAND qui permet, lors de l'allocation d'une région d'obtenir tout de suite la page en mémoire principale. Il n'y aura donc pas de défaut de page lors de l'accès à une zone de la région,
- Utiliser certains appels systèmes ayant une sémantique proche mais des performances différentes, on peut citer : MutexGet et MutexRel à la place de semP et semV, les mini-portes à la place des IPC locaux.

6.6 Adaptation de COOL et de CHORUS au modèle Saturne

Nous avons tout au long de ce chapitre présenté nos prototypes, les problèmes que nous avions rencontrés ainsi que les performances que nous avons obtenus pendant leur exécution. Nous pouvons maintenant répondre à la question posée par cette étude : CHORUS et COOL sont ils adaptés pour une implantation de Saturne? Nous traiterons cette question en commençant par répondre pour CHORUS, puis pour COOL.

CHORUS est un système temps réel préemptif. Les outils fournis par CHORUS, et en particulier la partie ordonnancement, offrent suffisamment de réactivité pour exécuter une application à base de noyaux synchrones. La manipulation des tâches transformationnelles par l'interface réactive peut être facilitée si le système d'exploitation offre des processus légers[7]. En effet, il est pratique que les tâches transformationnelles partagent le même espace d'adressage que l'interface réactive

[6] SVC pour **S**uper**V**isor **C**all.
[7] On parle aussi de "*threads*".

afin que celle-ci puisse facilement lire les données des tâches. CHORUS propose ces fonctionnalités. CHORUS offre tous les services qui permettent d'exécuter des noyaux synchrones et des tâches transformationnelles mais uniquement dans un environnement centralisé. En effet, les IPC CHORUS ne sont pas suffisamment déterministes. L'utilisation de COOL aggrave d'ailleurs cette situation. Nous avons déjà cité les hypothèses qui pourraient expliquer l'indéterminisme des invocations de méthodes COOL, et qui sont principalement :

- La priorité du serveur COOL n'est pas maîtrisée par le client,
- Les allocations dynamiques dans les souches et les squelettes CORBA,
- Les allocations dynamiques d'activités CHORUS pour traiter les invocations.

L'utilisation de COOL permet de bénéficier des apports des technologies CORBA. CORBA offre principalement la transparence à la localisation, l'interopérabilité et un outil pour réaliser facilement des applications réparties. Dans le cadre d'applications strictement développées sur CHORUS, la transparence à la localisation n'apporte rien puisqu'elle est déjà fournie par CHORUS. Par contre, l'interopérabilité entre des versions de COOL sur CHORUS et des versions de COOL sur des systèmes Unix ou Windows permettrait une communication aisée entre le monde Unix et CHORUS. Auparavant, il fallait utiliser les sockets TCP/IP. Cette interopérabilité est importante car il est tout à fait envisageable de concevoir que des interfaces homme machine d'applications temps réel s'exécutent sur des machines Unix alors que l'application elle même fonctionne sur un système CHORUS.

COOL a permis la réalisation des quatre prototypes en seulement trois mois et demi de développement et de tests. L'utilisation des activités COOL est plus aisée que les activités CHORUS et il est évident que les services de synchronisation répartie et d'invocation sur groupes sont des outils intéressants. Grâce aux services d'invocation sur groupes, les mécanismes de votes ont été implantés aisément. Un point important concerne le mode d'invocation sur groupe. Le client lorsqu'il effectue une invocation ne sait pas à priori si l'invocation est faite sur un objet ou un groupe de serveurs. Ceci permet au code applicatif d'être indépendant du nombre de répliques mises en oeuvre pour assurer la fiabilité. Cependant, la gestion de groupes dans la version de COOL que nous avons utilisé ne correspond pas **à nos besoins**. Les problèmes face à la tolérance aux pannes que nous avons cités dans ce chapitre ne permettent pas actuellement d'envisager des mécanismes de redondances actives sur Saturne **qui correspondent aux besoins de Dassault Aviation.**

Chapitre 7

Conclusion

Les objectifs de cette étude étaient de voir dans quelles conditions il était envisageable de mettre en oeuvre le modèle Saturne au dessus de CHORUS et de COOL. Ce travail nous a aussi permis d'étudier la possibilité d'ajouter au modèle Saturne des propriétés de tolérance aux pannes et d'utiliser des technologies CORBA pour la phase de développement afin d'évaluer les apports de CORBA pour des applications temps réel embarquées.

Après une étude du modèle Saturne et une conception d'un sous-système sur CHORUS/COOL, un certain nombre de prototypes furent réalisés. Les prototypes ont montré qu'il est possible d'utiliser CHORUS/COOL pour implanter le modèle Saturne. Les technologies CORBA ont permis de développer très rapidement les différents prototypes que nous avons présentés dans le chapitre précédent, tout en conservant un comportement identique au modèle Saturne du CERT ONERA. **Toutefois, un certain nombre de problèmes importants et difficiles restent à régler :**

- Les IPC de CHORUS ne sont pas suffisamment déterministes à ce jour pour des applications temps réels à contraintes strictes,
- La possibilité de faire cohabiter un réseau asynchrone et un réseau synchrone (tel que ATM, FDDI ou un Ethernet déterministe) sur un même site CHORUS. Le réseau asynchrone est utilisé pour les communications entre tâches transformationnelles qui peuvent être d'un débit élevé mais qui ne possèdent pas de fortes contraintes temporelles. Le réseau synchrone est quant à lui dédié à la communication entre les noyaux synchrones avec des données de faible volume mais qui possèdent de fortes contraintes temporelles. Cet aspect n'a pas été abordé dans nos prototypes car il représente un travail volumineux, difficile a prendre en compte dans le cadre d'un mémoire d'ingénieur. Toutefois, la mise en oeuvre d'un réseau synchrone avec CHORUS est un travail qui

a déjà été réalisé par certains groupes, tel que le groupe ARCADE[29, 56] du CNET qui a développé une interface FDDI,

- L'indéterminisme des appels de méthode COOL dont nous avons donné les éventuelles causes dans le chapitre précédent,
- L'amélioration des services de diffusion de COOL et en particulier l'utilisation de méthodes *"oneway"* sur des groupes d'objets ainsi qu'une implantation garantissant une atomicité des diffusions,
- Et enfin, l'adaptation de COOL à l'apparition de pannes franches d'objets serveurs ou d'objets clients. En effet, COOL n'étant pas conçu à l'origine pour la tolérance aux pannes, son comportement face à une panne franche d'un objet COOL ne convient pas **à nos problèmes** (voir le chapitre six).

Tous ces problèmes très importants devront être résolus pour réaliser une implantation satisfaisante de Saturne sur CHORUSCOOL. Enfin, de nombreux aspects peuvent encore être explorés dans des études futures. Nous avons utilisé le modèle Saturne mono-synchrone, il faudra étudier les conséquences des extensions multi-synchrones. Dassault Aviation a pour objectif de multiplier par 10 le nombre d'heures de vol sans maintenance de ses avions. Les mécanismes de tolérance aux pannes devront masquer les défaillances matérielles. La définition de ces mécanismes en fonction des besoins du logiciel avionique ainsi que ses conséquences sur le système est un sujet d'étude ardus mais qui offre des perspectives prometteuses. Le travail présenté ici n'est qu'une ébauche des mécanismes de tolérance aux pannes qui pourraient être utilisés avec Saturne.

Bibliographie

[1] F. Boniol M. Adelantado. Programming distributed reactive systems : a strong and weak synchronous coupling. CERT ONERA, September 1993.

[2] G. Berry. The constructive semantics of pure esterel. Ecole des Mines de Paris, Sophia-Antipolis, 1995.

[3] G. Berry and L. Cosserat. The synchronous programming language Esterel and its mathematical semantics. In S. Brookes and G. Winskel, editors, *Seminar on Concurrency*, pages 389–448. Springer Verlag Lecture Notes in Computer Science 197, 1984.

[4] G. Berry, P. Couronné, and G. Gonthier. Programmation synchrone des systèmes réactifs: le langage Esterel. *Technique et Science Informatiques*, 6(4):305–315, 1987.

[5] G. Berry and G. Gonthier. The Esterel synchronous programming language: Design, semantics, implementation. *Science Of Computer Programming*, 19(2):87–152, 1992.

[6] A.P.W. Bohm. Dataflow computation. CWI Tract, Center for Mathematics and Computer Science, 1983.

[7] F. Boniol. COREA : A synchronous calculus of parallel communicating reactive automata. CERT, PARLE'94, Athènes, Grèce, July 1994.

[8] M. Adelantado F. Boniol. Programming communicating distributed reactive automata : the weak synchronous paradigm. CERT ONERA, International Conference on Decentralized and Distributed Systems, Palme de Mallorca, Spain, September 1993.

[9] M. Adelantado F. Boniol. SATURNE : un modèle de description de systèmes multi-agents temps réel et intelligents. CERT ONERA, mai 1994.

[10] A. Bouali. XEVE : An esterel verification environment. CMA-Ecole des Mines de Paris, Rapport Interne, June 1996.

[11] F. Boulanger. Intégration de modules synchrones dans la programmation par objets. Thèse de doctorat, PARIS XI Orsay, décembre 1993.

[12] Randal E. Bryant. Symbolic boolean manipulation with ordered binary decision diagrams. Fujitsu Laboratories Ltd, ACM Computing Surveys, May 1991.

[13] E. Audureau P. Enjalbert L. Farinas Del Cerro. Logique temporelle : sémantique et validation de programmes parallèles. Edition Masson, 1990.

[14] CISI INGENIERIE. Esterel V3 : language reference manual. Agence Provence ouest, 1995.

[15] Kenneth Birmann Timothy Clark. Performance of the isis distributed computing toolkit. June 1994. Technical report TR-94-1432.

[16] Kenneth Birmann Robert Cooper. The isis project : Real experience with a fault tolerant programming system. ACM Operating Systems Review, April 1991.

[17] M. Adelantado F. Boniol M. cubéro-castan B. Lécussan R. Porche V. David. Projet SATURNE : modèle de programmation et modèle d'exécution pour un système temps réel d'aide à la décision. CERT ONERA, 5th Euromicro Workshop on Real-Time, Oulu, Finland, juin 1993.

[18] A. Bouali A. Ressouche V. Roy R. de Simone. The FCTOOLS user manual (version 1.0). INRIA, Rapport Technique numéro 191, April 1996.

[19] Y. Faure. SATURNE objet : une approche distribuée orientée objet des systèmes temps réel mixtes réactifs/interruptibles. ENSAE, Rapport de DEA, juin 1996.

[20] O. Gaultier and O Metais. Mise en place d'une plate-forme CHORUS, conception et implantation d'un ordonnanceur à échéance au sein du noyau CHORUS. Technical report, Rapport CNAM/CEDRIC, 1994.

[21] P. Le Guernic and T. Gautier. Dataflow to Von Neumann : the SIGNAL approach. INRIA-RENNES, Rapport numéro 1229, 1990.

[22] P. Le Guernic, T. Gautier, M. Le Borgne, and C. Le Maire. Programming real time applications with SIGNAL. INRIA-RENNES, Rapport numéro 1446, 1991.

[23] N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud. Programmation et vérification des systèmes réactifs : le langage Lustre. *Technique et Science Informatiques*, 10(2):139–157, 1991.

[24] D. Harel and A. Pnueli. On the developement of reactive systems. pages 477–498. In Logic and Models of Concurrent Systems. Proc NATO Advanced Study Institute on Logics and Models for Verifications and Specification of Concurrent Systems. New York, 1985.

[25] INRIA. Rapport d'activité scientifique 1995 du projet MEIJE. Parallélisme, synchronisation et temps réel, 1995.

[26] C. Santellani J. Delacroix. A test bed for distributed real-time scheduling experimentation based on CHORUS micro-kernel. Proc. of European Research Seminar on Advances in Distributed Systems ERSADS'95 , L'Alpe d'Huez, France, April 1995.

[27] Kenneth Birmann Thomas A. Joseph. Reliable communication in the presence of failure. ACM Trans. on Computer Systems, February 1987.

[28] S. Landis and S. Maffeis. Building reliable distributed systems with CORBA. Cornell University, Isis distributed Systems Inc, 1994.

[29] L. Leboucher and J.B. Stefani. Admission control for end to end distributed bindings. Centre National d'Etudes des Télécommunications (CNET), Lecture Notes in computer science, Teleservice and Multimedia Communications, vol 1052, November 1995.

[30] P. Weiss X. Leroy. Le langage Caml. Interédition, 1993.

[31] INMOS Limited. OCCAM 2 reference manual. Prenticel Hall, 1987.

[32] S. Maffeis. A flexible system design to support object groups and object oriented distributed programming. Departement of Computer Science, University of Zurich, 1993.

[33] S. Maffeis. Adding group communication and fault tolerance to CORBA. Departement of Computer Science, University of Zurich, June 1995.

[34] S. Maffeis. *Run Time Support for Object Oriented Distributed Programming.* PhD Thesis, Université de Zurich, February 1995.

[35] S. Maffeis, A. Vaysburd, and S. Georgiakaki. A interoperable middleware infrastructure supporting object group communication. Departement of Computer Science, Cornell University, 1993.

[36] C. André H. Boufaied D. Gaffé J.P. Marmorat. Environnement pour la programmation synchrone des systèmes réactifs. pages 27–41. Présenté à Real Time & Embedded Systems, Paris , Laboratoire Informatique, université de Nice Sophia Antipolis, CMA-Ecole des Mines de Paris, janvier 1996.

[37] R. Milner. A calculus of communicating system. volume 92. Lecture Notes in computer science, edited by G. Goos and J HartManis, Springer Verlag.

[38] T. J. Mowbray and R. Zahavi. The essential CORBA. OMG Document, 1995.

[39] L. Jategaonkar Jagadeesan C. Puchol J.E. Von Olnhausen. Safety property verification of esterel programs and applications to telecommunications software. Proc. of the Seventh Conference on Computer Aided Verification, July 1995.

[40] OMG. The common object request broker: Architecture and specification. Document 91.12.1, 1991.

[41] OMG. The common object request broker architecture and specification, revision 1.2. TC Document 93-12-43, December 1993.

[42] R. Orfali, D. Harkey, and J. Edwards. The essential distributed objects survival guide. 1995.

[43] G. W. George E. Kryal OSAF FORUM Draft. The perception and use of standards and components in embedded software development. July 1996.

[44] J.E. Pin. Théorie des automates finis. 1993.

[45] Z. Manna A. Pnueli. A hierarchy of temporal properties. Proc. of the 2th symph. ACM of principle of distributed computer, 1990.

[46] M. Adelantado F. Boniol R. Porche. Un modèle synchrone distribué et déterministe pour le temps-réel. CERT ONERA, mai 1993.

[47] M. Adelantado F. Boniol R. Porche. Etude d'un modèle hiérarchique et structuré multi-synchrone pour la programmation de systèmes réactifs. CERT ONERA, janvier 1996.

[48] M. Adelantado F. Boniol V. David B. Lecussan R. Porche. Predictability in distributed intelligent real-time systems. CERT ONERA, First IEEE Workshop on Parallel and Distributed Real-Time Systems, Newport Beach, California, April 1993.

[49] O. Potonniée. thèse de doctorat : Etude et prototypage en ESTEREL de la gestion de processus d'un micro-noyau de système d'exploitation réparti avec garantie de service. PARIS VI/CNET, avril 1996.

[50] T. M. Hickey D. Malki A. Vaysburd W. Vogels R. Renesse K. P. Birman B. Glade K. Guo M. Hayden. Horus: A flexible group communications system. March 1995.

[51] L. Rosenfeld. Mémoire partagée répartie dans un autocommutateur Alcatel A4400. Mémoire d'ingénieur C.N.A.M., September 1996.

[52] M. Rozier, V. Abrassimov, and F. Armand. Overview of the CHORUS Distributed Operating Systems. Technical report, Chorus Systèmes, CS/TR-90-25.1, February 1991.

[53] F. Singhoff. *Réalisation d'un sous-système Saturne tolérant les pannes*. Rapport de DEA Système informatique, Université Paris 6, septembre 1996.

[54] Y. Sorel. Génération d'exécutifs distribués optimisés pour les langages synchrones. Proc. R.T.S. Paris, January 1994.

[55] Y. Sorel. Massively parallel computing systems with real time constraints the : Algorithm architecture adequation methodology. Proc Massively Parallel Computing Systems, the Challenges of of General Purpose and Special Purpose Computing, Ischia, May 1994.

[56] J.B. Stefani, G.S. Blair, G. Coulson, M. Papathomas, P. Robin, F. Horn, and L. Hazard. A programming model and system infrastructure for real-time synchronization in distributed multimedia systems. *IEEE Journal on selected areas in communications*, 14(1):249–263, January 1996.

[57] Kenneth Birmann Pat Stephenson. Fast causal multicast. Proc. of the ACM, June 1991.

[58] Kenneth Birmann André Schiper Pat Stepheson. Lighweight causal and atomic group multicast. ACM Trans. on Computer Systems, August 1991.

[59] Chorus Systèmes. CHORUS Kernel V3 R5 Programmer's Reference Manual. May 1994. CS/TR-92-26.3.

[60] Chorus Systèmes. CHORUS Kernel V3 R5 Specification and interface. May 1994. CS/TR-94-2.1.

[61] Chorus Systèmes. ClassiX R2 programmer's manual. May 1994. CS/TR-95-2.1.

[62] Chorus Systèmes. CHORUS/COOL-ORB Programmer's Guide. February 1996. CS/TR-96-2.1.

[63] Chorus Systèmes. CHORUS/COOL-ORB Programmer's Reference Manual. February 1996. CS/TR-96-3.1.

[64] M. Cosnard D. Trystram. Architectures et algorithmiques parallèles. Ed. Interéditions. Paris, 1994.

[65] A. Aho J. Ullman. Concepts fondamentaux de l'informatique. Edition DUNOD, 1993.

[66] Robbert van Renesse Kenneth P. Birman Robert Cooper. The horus system. July 1993.

Annexe A

Sources de l'exemple de l'émetteur/récepteur du chapitre deux

A.1 Fichier d'entête (esterel.h) :

```
#ifndef esterel_h
#define esterel_h

typedef struct mymsg {
                    long type;
                    int req;
                    int info;
                    };

#endif
```

A.2 Fichier principal (main.h) :

```
#include <stdio.h>
#include <sys/types.h>
```

```
#include <sys/ipc.h>
#include <sys/msg.h>
#include <unistd.h>
#include <signal.h>
#include <errno.h>
#include "esterel.h"


int msgid, pid, j;
int compteur=0;
struct sigaction action;

void fils(void);
void the_end(int sig);
void read_input_recepteur(void);
void read_input_emetteur(void);

/*********************************************/
/*********************************************/

void main (void)
{

action.sa_handler=the_end;
sigaction(SIGHUP,&action,NULL);


/* creation de la file */
if ( (msgid= msgget(150,IPC_CREAT|0666)) < 0)
        {
        perror("pb sur msgget");
        exit(0);
        };


recepteur();   /* inititialisation  */
emetteur();    /* des noyaux Esterel */
```

```
if (pid=fork())
        fils();


for(j=1;j<=5;j++)
        {
        read_input_emetteur();
        emetteur();
        };


while(1);

};

/***********************************************/
/***********************************************/

void fils (void)
{
while(1)
        {
        read_input_recepteur();
        recepteur();
        };
};

/***********************************************/
/***********************************************/

void read_input_emetteur(void)
{
        compteur++;
        printf("read_input_emetteur : envoi de %i \n",compteur);
        emetteur_I_I(compteur);
};

/***********************************************/
/***********************************************/
```

```
void read_input_recepteur(void)
{
        struct mymsg mm;
        mm.type=0L;

        if (msgrcv(msgid,&mm,sizeof(struct mymsg),0L,0) == -1)
                {
                perror("pb sur msgrcv");
                exit(0);
                };

        printf("read_input_recepteur : data recue sur
                msgrcv()  %i  \n",mm.info);
        recepteur_I_DATA(mm.info);
};

/************************************************/
/************************************************/

void the_end (int sig)
{
        printf("the end ....\n");

        if (msgctl(msgid,IPC_RMID,NULL)<0)
                {
                perror("pb sur suppression");
                exit(0);
                };

        exit(0);
};
```

A.3 Fichiers concernant les signaux :

A.3.1 Signal de sortie ETAT du noyau EMETTEUR :

```
void emetteur_O_ETAT (int i)
{
        printf("emission en sortie du
                noyau EMETTEUR (ETAT) : %i \n",i);
};
```

A.3.2 Signal de sortie DATA du noyau EMETTEUR :

```
#include "esterel.h"

void emetteur_O_DATA (int i)
{
        struct mymsg tt;
        int mymsgid;
        tt.info=i;
        tt.type=1;

        printf("emission en sortie du noyau
                EMETTEUR (DATA) : %i \n",i);

if ((mymsgid=msgget(150,0))==-1)
        {
        perror("Noyau 2 : pb sur msgget ds emetteur_O_DATA ");
        exit(0);
        };

if (msgsnd(mymsgid,&tt,sizeof(struct mymsg),1) == -1)
        {
        perror("Noyau 2 : pb sur msgsnd ds emetteur_O_DATA ");
        exit(0);
        };
```

```
};
```

A.3.3 Signal de sortie O du noyau RECEPTEUR :

```
void recepteur_O_O (int i)
{
        printf("emission en sortie du
                noyau RECEPTEUR (O) : %i \n",i);
};
```

Annexe B

Sources Esterel des prototypes un, trois et quatre

B.1 Les sources Esterel de Pepin

```
module Pepin:

% Tache de controle des taches asynchrones
task Control()(integer);
return SigControl1;

% Types externes et constantes
type COMPLEX;

% Signaux transmis a l'exterieur de Pepin
output FIN(integer);
output TOP(COMPLEX);
output FIN_GRAINES;
output CONTINUE_GRAINES;

% Signaux recus de l'exterieur
input CPLX_DATA(COMPLEX);
input VERS_PEPIN(integer);


procedure start_T2(integer)(COMPLEX, COMPLEX, COMPLEX);
procedure consult_T2(COMPLEX)(integer);
```

```
procedure print_COMPLEX()(COMPLEX);
procedure print_integer()(integer);

signal FINIR_GRAINES in
   var index_1 : integer in
      var p1, p2 : COMPLEX in
          await CPLX_DATA do
          emit FINIR_GRAINES;
          p1 := ?CPLX_DATA;
          call start_T2(index_1)(p2, p1, p1);
          exec Control()(index_1) return SigControl1;
          call consult_T2(p2)(index_1);
          call print_COMPLEX()(p1);
          call print_COMPLEX()(p2);
          emit FIN(index_1);
          emit TOP(p2);
      end
   end
end
||
[
       do
          loop
              await VERS_PEPIN;
              call print_integer()(?VERS_PEPIN);
              emit CONTINUE_GRAINES;
          end;
          watching FINIR_GRAINES;
          emit FIN_GRAINES;
];
end
end module
```

B.2 Les sources Esterel de Graines

```
module Graines:
```

```
% Types externes et constantes
type COMPLEX;

output VERS_PEPIN(integer);
output VERS_NOYAU(COMPLEX);

input  FIN_GRAINES;
input  CONTINUE_GRAINES;

procedure print_COMPLEX()(COMPLEX);
procedure print_integer()(integer);
function gen_integer() : integer;
function gen_COMPLEX() : COMPLEX;

do
loop
   var p1 : COMPLEX in
       var  p2 : integer in
            p1:=gen_COMPLEX();
            p2:=gen_integer();
            call print_COMPLEX()(p1);
            call print_integer()(p2);
            emit VERS_NOYAU(p1);
            emit VERS_PEPIN(p2);
            await CONTINUE_GRAINES;
       end
   end
end
watching FIN_GRAINES;
end module
```

B.3 Les sources Esterel de Noyau

```
module Noyau:

% Tache de controle des taches asynchrones
task Control()(integer);
```

```
return SigControl1;
return SigControl2;
return SigControl3;
return SigControl4;
return SigControl5;
return SigControl6;

% Types externes et constantes
type COMPLEX;
constant unite, i, pi : COMPLEX;

% Signaux transmis a l'exterieur de Noyau
output FIN(integer);
output CPLX_DATA(COMPLEX);

% Signaux recus par Noyau
input TOP(COMPLEX);
input VERS_NOYAU(COMPLEX);


procedure start_T1(integer) (integer, integer);
procedure consult_T1(integer)(integer);

procedure start_T2(integer)(COMPLEX, COMPLEX, COMPLEX);
procedure consult_T2(COMPLEX)(integer);

procedure start_T3(integer)(string, integer);
procedure consult_T3(integer)(integer);

function  isActive(integer) : boolean;
function  isSuspended(integer) : boolean;
function  isFinished(integer) : boolean;

procedure print()(integer, integer, integer);
procedure print_COMPLEX()(COMPLEX);

var index_1, index_2, index_3, index_4, index_5 : integer in
   var param1 := 8 : integer,      % Id 1 fait 16 tours
       param2 := 8 : integer,
       param3 := 1 : integer,      % Id 2 fait 2 tours et tue Id 1
```

```
       param4 := 1 : integer
   in
      signal KILL_T1 in
signal INDEX1(integer), INDEX2(integer) in
do
    call start_T1(index_1)(param1, param2);
    emit INDEX1(index_1);
    exec Control()(index_1) return SigControl1;
    call consult_T1(param1)(index_1);
    call print()(param1, param2, 0);
    emit FIN(index_1);
 watching KILL_T1;
      ||
 await immediate INDEX1 do
   var Id1 : integer in
     Id1 := ?INDEX1;
     call start_T1(index_2)(param3, param4);
     exec Control()(index_2) return SigControl2;
     if not isFinished(Id1) then
       emit KILL_T1;
     end;
     call consult_T1(param3)(index_2);
     call print()(param3, param4, 0);
     emit FIN(index_2);
   end;
 end;
      ||
 await KILL_T1;
 var p1 := 7 : integer in
    call start_T3(index_5)("PtitLoup", p1);
    emit INDEX2(index_5);
    exec Control()(index_5) return SigControl5;
    call consult_T3(p1)(index_5);
    call print()(p1, 0, 0);
    emit FIN(index_5);
 end
     ||
await INDEX2 do
  var Id2 : integer in
    Id2 := ?INDEX2;
```

```
  end var
end
end signal
      end signal
   end var
   ||
   var p1, p2, p3 : COMPLEX in
       call start_T2(index_3)(p1, unite, i);
       exec Control()(index_3) return SigControl3;
       call consult_T2(p1)(index_3);
       call print_COMPLEX()(p3);
       call print_COMPLEX()(p1);
       call print_COMPLEX()(unite);
       call print_COMPLEX()(i);
       emit FIN(index_3);
       emit CPLX_DATA(p1);

       await TOP do
          p2 := ?TOP;
          call start_T2(index_3)(p1, pi, p2);
          exec Control()(index_3) return SigControl6;
          call consult_T2(p1)(index_3);
          call print_COMPLEX()(p1);
          call print_COMPLEX()(pi);
          call print_COMPLEX()(p2);
          emit FIN(index_3);
       end
     end
   ||
      var p1 := 9 : integer in
          call start_T3(index_4)("PtitLoup", p1);
          exec Control()(index_4) return SigControl4;
          call consult_T3(p1)(index_4);
          call print()(p1, 0, 0);
          emit FIN(index_4);
      end
   ||
     loop
         await VERS_NOYAU;
         call print_COMPLEX()(?VERS_NOYAU);
```

```
     end;
end
end module
```

Annexe C

L'interface homme machine du deuxième prototype