Un compilateur Estelle multi-processeurs pour l'expérimentation d'algorithmes distribués sur machines parallèles

Claude Jard, Jean-Marc Jézéquel

INRIA

UNITE DE RECHERCHE
INRIA-RENNES

Institut National
de Recherche
en Informatique
et en Automatique

Domaine de Voluceau
Rocquencourt
BP 105
78153 Le Chesnay Cedex
France
Tél (1) 39 63 55 11

Rapports de Recherche

N° 993

Programme 3

UN COMPILATEUR ESTELLE MULTI-PROCESSEURS POUR L'EXPERIMENTATION D'ALGORITHMES DISTRIBUES SUR MACHINES PARALLELES

Jean-Marc JEZEQUEL
Claude JARD

Mars 1989

RR-0993

IRISA INSTITUT DE RECHERCHE EN INFORMATIQUE ET SYSTÈMES ALÉATOIRES

Campus Universitaire de Beaulieu
35042-RENNES CÉDEX
FRANCE
Téléphone: 99 36 20 00
Télex: UNIRISA 950 473 F
Télécopie: 99 38 38 32

Publication Interne n°453 - Janvier 1989 - 54 Pages

Un compilateur Estelle multi-processeurs pour l'expérimentation d'algorithmes distribués sur machines parallèles

Jean-Marc JEZEQUEL
Claude JARD
E-mail: jard@irisa.fr, jezequel@irisa.fr

Résumé

Ce rapport présente dans le détail un outil de génération de code parallèle à partir de descriptions formelles d'algorithmes distribués exprimées dans le langage Estelle. Nous avons considéré un contexte assez simple de distribution, où l'on traite un sous-ensemble " statique " d'Estelle et où la machine cible est homogène (exemple du calculateur iPSC). Nous commençons par présenter et justifier le concept d'expérimentation d'algorithmes distribués sur machines parallèles, pour lequel notre compilateur Estelle a été conçu. Nous discutons alors comment les constructions Estelle ont été codées dans des structures du langage *C*, et comment elles sont interprétées par un noyau d'exécution distribué. Des annexes techniques donnent une idée précise de l'état actuel de l'outil réalisé, appelé Echidna.

A Multi-processor Estelle-to-C Compiler to Experiment Distributed Algorithms on Parallel Machines

Abstract

This paper presents a first attempt to generate parallel codes from Estelle descriptions. We have dealt with a simple context in which only a static subset of Estelle and an homogeneous target machine are considered. We begin to present and justify the concept of experimentation of distributed algorithms for which our Estelle compiler has been designed. Then we discuss how the Estelle constructs are mapped onto C-structures and how they are interpreted by a distributed runtime kernel. A technical annex gives an idea of the current version of the tool, named Echidna.

CENTRE NATIONAL DE LA RECHERCHE SCIENTIFIQUE
(L. A. 227)
UNIVERSITÉ DE RENNES 1 I. N. S. A. DE RENNES

INSTITUT NATIONAL DE RECHERCHE
EN INFORMATIQUE ET EN AUTOMATIQUE
(LABORATOIRE DE RENNES)

Table des matières

1 Introduction **3**
- 1.1 L'expérimentation, un aspect de la validation des algorithmes 3
- 1.2 Echidna: des outils pour l'expérimentation 5

2 Le modèle d'exécution **7**
- 2.1 Notre modèle Estelle 7
- 2.2 L'exécuteur 11
- 2.3 Le runtime Estelle 12
- 2.4 L'interface avec le système 13

3 La représentation des objets Estelle en *C* **14**
- 3.1 La spécification 14
- 3.2 Les tâches 14
- 3.3 Les ports 16
- 3.4 Les messages 17
- 3.5 Les transitions 17
- 3.6 Traduction des instructions propres à Estelle 19
- 3.7 La structure du code généré 22
- 3.8 L'accès aux variables et renommage 24
- 3.9 La traduction du Pascal 25
  - 3.9.1 Les constantes 25
  - 3.9.2 Types et variables 26
  - 3.9.3 Les instructions 29
  - 3.9.4 Expressions 30

4 Mise en œuvre du compilateur **31**

5 Conclusion **31**

1 Introduction

1.1 L'expérimentation, un aspect de la validation des algorithmes

La programmation efficace des machines parallèles et distribuées est encore un défi. Nous savons tous que concevoir et réaliser des algorithmes distribués corrects (fiables) et performants est un exercice difficile. Aussi, tant dans les laboratoires de recherche que dans le milieu industriel on estime aujourd'hui nécessaire de prévoir une étape dite de *validation* entre la phase de spécification et la mise en œuvre d'un algorithme distribué.

De nombreuses équipes travaillent sur des outils de validation d'algorithmes distribués. Ces outils, fort différents dans leurs formes et leurs finalités, ont pourtant en commun d'accepter en entrée la description formelle d'une spécification d'algorithme distribué (ou protocole) et de fournir comme résultat un certain degré de confiance sur tel ou tel aspect ou propriété de l'algorithme distribué.

En simplifiant, on peut dire que le concepteur dispose de trois niveaux d'intervention pour valider son algorithme :

- la vérification formelle de propriétés : elle fournit un résultat sûr de validité mais présente de nombreuses limites d'applicabilité. Seuls des modèles simplifiés du protocole correspondant à un niveau d'abstraction assez élevé peuvent être raisonnablement analysés par les méthodes actuelles. Ceci induit le problème (encore largement ouvert) de la préservation de la validité des propriétés au cours du raffinement du protocole.

- la simulation du protocole dans un environnement simulé (simulation centralisée généralement) : elle peut traiter des modèles d'algorithmes assez complets et permet de détecter efficacement des erreurs éventuelles sur un sous-ensemble des comportements possibles du protocole. La principale difficulté réside dans la nécessité de décrire et simuler l'environnement d'exécution de l'algorithme. Celui-ci est généralement trés simplifié : il n'est pas réaliste de vouloir prendre en compte tous les paramètres d'un système d'exécution réel, comme par exemple l'influence précise de la taille des messages sur les temps de transmission, ou la durée des actions (non calculables sans exécution).

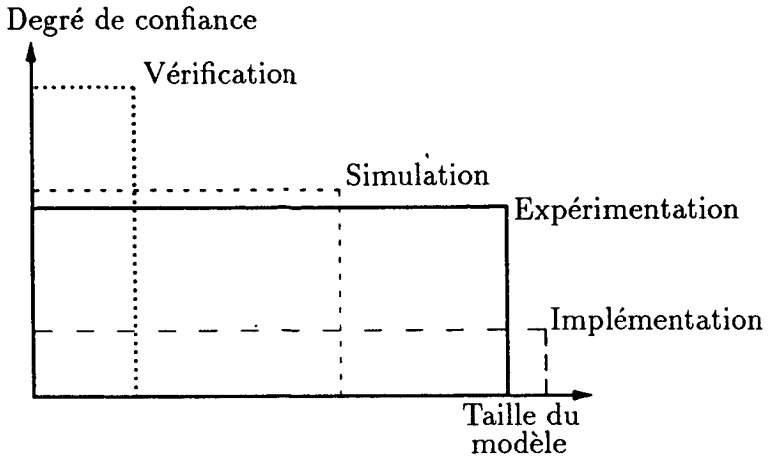


Figure 1 : La couverture des méthodes de validation

- l'observation et le test d'une implantation prototype du protocole : bien que présentant l'avantage d'utiliser un environnement d'exécution réel, cette approche doit surmonter deux difficultés : l'absence d'outils pour observer un système réparti dans son ensemble, et le fait que le comportement du prototype peut dépendre étroitement de la mise en œuvre (par exemple, la résolution du non-déterminisme). Aussi il est difficile de tirer des conclusions générales quant au comportement de l'algorithme distribué réel.

Ces différentes approches, on le voit, sont complémentaires et doivent être utilisées de concert par le validateur. Nous proposons en fait de définir et justifier une technique intermédiaire entre la simulation et l'implantation prototype. C'est cette approche que nous appelons *expérimentation.*

L'expérimentation a pour but de contribuer à valider des algorithmes dans un environnement réel, en explorant le plus grand nombre de leurs comportements possibles. Certains paramètres n'ont plus besoin d'être simulés, ils sont fournis par la machine parallèle elle-même. Si cette machine est suffisamment générale et que son environnement est bien maîtrisé, on pourra transposer certains aspects du comportement de l'algorithme

distribué expérimenté à n'importe quel type de machine, en quelque sorte *modulo* la machine d'expérimentation. Les expérimentations effectuées sur l'hypercube iPSC d'Intel et des algorithmes synchroniseurs[2] abondent dans ce sens.

Un avantage de l'expérimentation sur machine parallèle est que par construction on tire parti de la puissance de calcul de ce type de machine, ce qui permet d'envisager la validation d'algorithmes comportant plusieurs dizaines de milliers de processus.

L'inconvénient principal en découle: il faut mettre en œuvre des techniques spéciales pour pouvoir effectuer des prises de mesures et de l'observation distribuées.

1.2 Echidna: des outils pour l'expérimentation

Le projet Echidna a pour but de fournir les outils nécessaires à l'expérimentation d'algorithmes distribués sur machines parallèles. Il est construit autour d'un compilateur d'algorithmes distribués (décrits formellement dans le langage Estelle) fournissant du code pour machines multi-processeurs, et d'un ensemble d'outils pour observer le comportement de l'algorithme distribué sous expérimentation.

La description formelle doit modéliser le système distribué dans son ensemble (on parle de système fermé). Ceci permet de travailler sur l'algorithme complet, distribué sur le réseau, et non pas seulement sur son implantation sur un nœud, où la communication distante ne fait pas partie du modèle, mais est vue comme l'appel d'une procédure primitive [13].

Ayant une bonne expérience de l'étude des techniques de description formelle, nous avons choisi pour décrire les algorithmes distribués le langage Estelle maintenant normalisé par l'ISO[9]. Estelle nous semble en effet particulièrement bien adapté à la description d'algorithmes distribués: sa sémantique fondée sur des automates communicants étendus avec le langage de programmation Pascal est très proche de la sémantique naturelle des systèmes répartis faiblement couplés.

Estelle rend accessible au concepteur d'algorithme distribué la puissance d'expression de Pascal, ainsi que quelques mécanismes empruntés au langage Ada[1] (mécanisme modèles-instances, modularité partielle...). Ceci permet

[1] Ada is a registred Trade Mark of the DoD.

d'écrire des compilateurs efficaces produisant du code performant.

De nombreux travaux ont déjà été effectués autour de Estelle sur des activités relevant de la validation (voir la synthèse [3]). Plusieurs outils Estelle existent aux Etats-Unis, au Canada, en Europe, où deux projets européens Esprit ont été développés (Sedos et Sedos-demo [7]).

Ainsi, avec la même description on peut vérifier, simuler, ou même implémenter un algorithme distribué: on est absolument sûr de bien travailler sur le même objet.

Notre contribution à ce domaine de recherche est d'être capable de traduire automatiquement des descriptions Estelle en codes parallèles exécutables sur système distribué, afin de faire de l'expérimentation.

Pour qu'une machine parallèle soit intéressante dans le cadre de l'expérimentation, il faut qu'elle ait un caractère suffisamment général, qu'elle soit puissante et que son environnement soit bien maîtrisé. Notre modèle de *station d'expérimentation* possède les propriétés suivantes:

- Ensemble de sites (processeurs ou machines) communicants seulement par échanges de messages au travers d'un réseau de communications point à point.
- Pas de perte de messages, temps de transfert des messages fini mais non prévisible.
- Les canaux de communication sont FIFO: les messages ne se déséquencent pas sur un canal.

Différentes machines peuvent mettre en œuvre une telle abstraction. Nous nous sommes intéressés en premier lieu à développer notre approche sur l'hypercube *iPSC* (Intel) dont nous disposons à l'Irisa. Nous utilisons actuellement la version iPSC/2 dans laquelle nous disposons de 64 processeurs 80386 avec une mémoire de 4 méga-octets chacun (voir [8] et [4] pour une présentation détaillée).

Nous avons aussi travaillé sur des réseaux de stations SUN (au dessus de la couche TCP/IP) et sur des PC. Nous sommes maintenant en train d'explorer le monde des Transputers à travers l'hypercube T-20 (de Floating Point Systems) installé à Grenoble en France, et le super-calculateur T-node.

Ce type de machines évoluant rapidement, il ne faut pas se lier à une architecture et à un processeur particulier. Aussi avons nous choisi le langage *C* comme cible pour notre compilateur Estelle. Le *C* est à peu près normalisé et portable, et surtout un compilateur *C* relativement efficace est disponible pour chacune des machines parallèles citées ci-dessus, ce qui n'est malheureusement pas le cas pour Pascal, qu'il semblait pourtant plus naturel de choisir comme langage intermédiaire entre Estelle et la machine. Mettre en œuvre les constructions Pascal du langage Estelle en C est une tâche fastidieuse mais nous a permis de générer du code relativement compact, bien structuré et qui peut être aisément interfacé avec d'autres programmes et librairies.

Le but de ce rapport est de présenter en détail une première ébauche à la génération de code parallèle à partir de descriptions Estelle. Nous nous sommes situés dans un contexte simple dans lequel seul un sous-ensemble "statique" d'Estelle et une machine cible homogène (mêmes processeurs et compilateurs) sont considérés.

Nous présentons comment les constructions Estelle sont mises en correspondance avec les structures *C*, et comment celles-ci sont interprétées par un Noyau d'Exécution Distribué (NED). La description des outils pour l'observation (en cours de développement) fera l'objet d'une autre communication (quelques idées préliminaires peuvent être trouvées dans [11]). Nous donnons en annexe un exemple de code généré, le listing du noyau d'exécution et le manuel d'utilisation de la version courante de notre outil Echidna (lequel commence à être régurlièrement utilisé par des programmeurs parallèles).

2 Le modèle d'exécution

2.1 Notre modèle Estelle

Nous présentons ici le modèle Estelle que nous avons considéré. Nous supposons que le langage Estelle complet est connu du lecteur dans ses grandes lignes (voir [6] et [5] pour des présentations). Nous rappelons ici quelques aspects et surtout les restrictions que nous avons introduites.

Une spécification Estelle est la description du comportement d'un système

fermé, c'est à dire sans interaction avec l'extérieur. Un système est soit un ensemble de sous-systèmes s'exécutant en parallèle asynchrone, soit un ensemble de tâches s'exécutant en parallèle (à ce niveau, le parallélisme peut être synchrone ou asynchrone).

Dans le cadre de l'expérimentation sur machines parallèles, nous imposons un parallélisme asynchrone entre sites différents. Le placement des processus sur les sites physiques est explicitement défini par le programmeur Estelle : le premier paramètre d'un module peut être un entier désignant le site, l'association étant effective lors de la création du processus (initialisation) [2].

Le comportement d'un système se compose de la déclaration de constantes, de types, puis de modèles de canaux, qui sont des files FIFO bi-directionnelles. Vient ensuite la déclaration de *modules* (ou sous-systèmes), possédant un ensemble fini de ports (ou points d'interaction: moitiés d'instances de canaux) par lesquels transitent des messages (ou interactions). Ces modules peuvent être paramétrès (syntaxe procédurale) et exporter certaines de leurs variables internes.

Une ou plusieurs déclarations de comportements (ou corps: *body*) sont associées à chaque *module*, et c'est seulement lors de la création d'une tâche (ou processus) qu'un corps va être affecté à une instance de module.

Après les déclarations de modules, on trouve les déclarations de variables de type module (instances des définitions de modules qui seront des tâches filles de la tâche courante), puis les variables locales (instances de types déclarés dans le champs de visibilité), et enfin la liste des états de contrôle de l'automate.

L'automate lui-même est composé d'une partie initialisation et d'une liste de transitions, qui sont des couples ($< condition >, < action >$). La condition (ou *garde* de la transition) peut porter sur l'état de l'automate, la présence et le contenu d'un message sur un port, les variables locales de l'automate étendu, une clause de priorité, un délai, ou une combinaison quelconque (éventuellement vide) de ces conditions. L'action se compose d'un éventuel changement d'état, suivi d'une instruction composée Pascal, étendue avec des instructions d'émission de messages, de création/suppression de tâches, et des connexions/déconnexions de canaux.

[2] Chaque nœud est supposé avoir une identité unique (un entier positif), et il existe un nœud 0. Une machine centralisée est vue comme le nœud 0 d'un réseau à un seul nœud

Une description Estelle est donc un arbre dont les nœuds sont des tâches et la racine la spécification englobante. Les fils d'une tâche sont les instances des modules (variables de type *module*). La structure de cet arbre est dynamique.

Dans Echidna cependant, nous nous sommes limités au sous-ensemble d'Estelle, qui permet toujours la création dynamique de l'arbre, mais pas sa modification. Cette limitation, aussi mise en œuvre dans Véda [10] ne semble pas trop restrictive au niveau d'abstraction ou nous nous situons, car la reconfiguration dynamique n'est le plus souvent utilisée que comme facilité d'écriture et pour optimiser la mémoire d'une implantation. De plus, plusieurs concepts d'Estelle deviennent alors caducs : le partage de variables entre fils d'un module n'est plus possible, ce qui rend inutile une règle importante d'Estelle, la priorité père-fils. La distinction entre processus et activités qui précise la nature du parallélisme perd aussi son sens.

Ce contexte simplifie considérablement notre implantation parallèle. Une perspective intéressante (mais assez difficile) serait de mettre en œuvre le langage Estelle complet avec tout son dynamisme. Estelle pourrait alors être utilisé comme un vrai langage de programmation de machines parallèles.

La dernière restriction sémantique concerne la clause *delay* du langage. Pour pouvoir observer des exécutions, nous avons bâti un service de temps global permettant d'estampiller les événements observables. Ce temps global pourrait être utilisé pour implanter le délai Estelle si nécessaire.

Il y a aussi quelques restrictions syntaxiques annexes : l'emboîtement des procédures et des transitions n'est pas permis. Ce n'est pas essentiel, et disparaîtra lorsque notre compilateur sera intégré à une station de développement Estelle.

Comme Echidna doit générer du code pour un système distribué, une tâche terminale (feuille) n'est réellement démarrée sur un site donné que si elle doit y être exécutée. Les tâches non terminales sont passives : leur automate est réduit à une simple initialisation.

Finalement, il ne reste sur chaque site que l'ensemble des tâches terminales créées et placées dynamiquement sur ce site. L'exécution du programme Estelle va donc consister à partir de là à exécuter en parallèle les tâches sur chaque site.

Pour mettre en œuvre ce modèle d'exécution, notre compilateur traduit la description formelle Estelle en un ensemble de structures de données (exprimées dans le langage de programmation *C*), qui décrivent totalement les constantes, types, modèles de canaux et de modules (associés à leurs

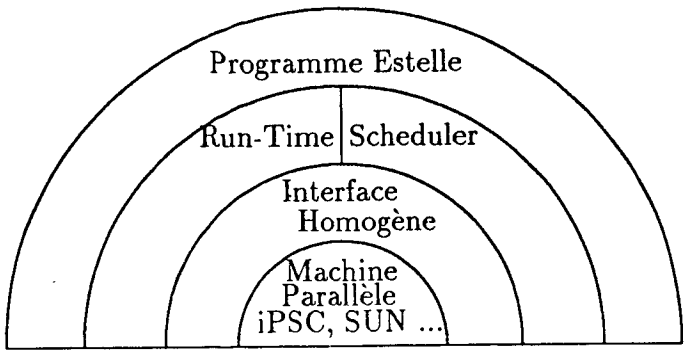


Figure 2 : Le compilateur Echidna

corps) de la spécification.

Cet ensemble de structures de données est ensuite compilé par le compilateur *C* local, puis lié avec le NED avant d'être chargé et exécuté sur le système cible (voir figure 2).

Notre approche est donc sensiblement différente de l'approche production d'une implantation dans un environnement hétérogène, car au contraire de [13] qui le premier proposa un compilateur Estelle vers *C*, nous produisons du code pour un système distribué complet, le NED ayant à charge de reconnaître et de gérer la communication non-locale entre tâches Estelle.

Les autres fonctions du NED sont de fournir les primitives spécifiques d'Estelle et le *runtime* Pascal bâti à partir de celui du *C*. Il réalise aussi l'interface avec le système distribué sous-jacent, notamment pour le placement des tâches, la communication distante et les entrées-sorties.

En pratique le NED, totalement écrit en *C*, est découpé en trois modules:

- L'exécuteur (enchaînement des tâches)
- Runtime Estelle
- Interface avec le système

Les fonctionalités de chacun de ces modules sont détaillées dans les paragraphes suivants.

2.2 L'exécuteur

Un module exécutable est obtenu après édition de liens entre le NED et le code généré par le compilateur. Nos expériences sont paramétrables en durée et grâce à un *germe* pour le générateur aléatoire. Une expérience est reproductible si on la met dans les mêmes conditions initiales.

Le module exécutable est chargé sur tous les sites, son point d'entrée est la fonction *main* de l'exécuteur, dont le texte est en substance:

```
main()
{
  int seed,duration;
  getOptions();
  init(seed);
  configuration();
  startScheduler(duration);
  terminate(0);
}
```

getOptions() analyse la ligne de commande pour trouver les options et paramètres de l'exécution du protocole, tels que le germe du générateur aléatoire (pour assurer des expériences reproductibles), ou demander un comportement particulier du scheduler...

init() réalise une initialisation générale des structures de données, du générateur aléatoire, et de la partie dépendante système.

configuration() est une fonction externe, générée par le compilateur, qui crée la tâche englobante (racine de l'arbre), lui passe les paramètres de la spécification, puis l'initialise en exécutant sa partie *initialize*, ce qui crée ses descendants et les initialise récursivement. Après que l'arbre est entièrement construit sur chaque site, seules sont initialisées les tâches terminales qui doivent tourner sur ce site. On peut alors démarrer le scheduler, qui assurera l'enchaînement des tâches présentes sur ce site pendant la durée spécifiée par l'utilisateur. Quand l'expérience se termine (normalement ou sur erreur), chaque site envoie sa trace locale vers une station de travail où elle pourra être analysée et exploitée (animation, vérification de propriétés...).

L'enchaînement des tâches est le travail du scheduler. Le scheduler va prendre en charge l'exécution des tâches initialisées sur chaque nœud du

réseau. Il travaille à deux niveaux de parallélisme:

- parallélisme inter-nœuds: c'est le vrai parallélisme de la machine, ce scheduler peut être vu comme un algorithme distribué.
- parallélisme intra-nœuds: ou *quasi-parallélisme* simulé par chaque projection du scheduler sur chaque site.

startScheduler démarre l'activité d'exécution, dépendant de la politique définie par la ligne de commande de l'utilisateur. Les différents choix possibles sont une exécution déterministe ou non-déterministe et un parallélisme synchrone ou asynchrone.

Ainsi, le scheduler effectue sur chaque site, l'algorithme simplifié suivant :

Repeter jusqu'a FinExperience
Recueillir les messages arrivant de l'exterieur(s'il y en a);
Pour chaque processus du site,
choisir la premiere transition tirable ou
choisir au hasard une transition parmi toutes les
transitions tirables de priorite maximale;
Pour chaque processus du site,
(ou seulement pour un sous-ensemble d'entre eux
si l'asynchronisme est simulé)
tirer la transition choisie(s'il y en a une) de
maniere atomique;
FinRepeter

Le code du scheduler (de faible taille: environ 400 lignes de *C* portable) est modulaire et accessible, donc facilement modifiable, pour par exemple implémenter d'autres sémantiques (plus orientées implémentation par exemple).

2.3 Le runtime Estelle

Le runtime a pour rôle de reconstruire une machine virtuelle permettant l'exécution d'un programme Estelle, à partir du runtime *C* qui existe autour de la machine. Il doit donc fournir les primitives de manipulation d'objets

Estelle et Pascal qui n'existent pas en C, un générateur aléatoire (pour le choix non déterministe parmi les transitions tirables) et reconnaître la communication distante pour l'acheminer par le réseau.

En effet, quand on envoie un message sur un port connecté à un port d'un autre site, le message est passé au réseau qui le délivre au site destinataire, où une fonction $RecueillirMessage()$, appelée à chaque tour du scheduler le recueillera et le déposera dans la file d'attente du port destinataire.

Le module contenant le runtime consiste en environ 1000 lignes de C, et est directement portable sur n'importe quelle machine disposant d'un compilateur C classique.

2.4 L'interface avec le système

Le but de ce module est de fournir de manière homogène l'abstraction *station d'expérimentation* quelque soit le système distribué sous-jacent.

Il fournit des primitives permettant d'émettre des messages vers les autres sites, et de tamponner les messages qui arrivent des autres sites. Il fournit une horloge locale, gère les entrées-sorties, et l'identificateur de site.

Selon la proximité de la machine parallèle par rapport au modèle de la station d'expérimentation, ce module peut être court (100 lignes de C dans le cas de l' *iPSC/2*), ou bien relativement complexe, comme dans le cas d'un réseau de Sun connectés par *ethernet* avec TCP/IP, où il faut rebâtir toute une couche logicielle à partir du concept de *socket*.

Dans tous les cas, seule cette partie du NED est à adapter à chaque nouveau système, ce qui contribue notablement à la portabilité de l'outil Echidna.

Lorsque l'on considère un seul site, Echidna se présente comme un simulateur Estelle complet pouvant fonctionner sur de petites machines (type PC).

Le NED est actuellement opérationnel pour les machines Hypercubes Intel *iPSC/1* et *iPSC/2*, Stations Sun en réseau par TCP/IP, Gould et PC.

3 La représentation des objets Estelle en *C*

3.1 La spécification

La spécification d'une description formelle en Estelle est un objet qui décrit le comportement d'un système fermé. C'est une boite noire, sans interaction avec l'extérieur.

Quand on regarde ce qui se passe dans cette boite noire pendant une exécution, on distingue un certain nombre d'objets qui interagissent entre eux, comme illustré en figure 3.

Ce sont:

- les tâches, formées de l'association d'un comportement (body) à un sous-système (module).
- les ports (points d'interactions) qui sont les interfaces des tâches avec l'extérieur.
- les messages, ou interactions, que s'échangent les tâches le long de canaux reliant leurs ports.
- les transitions, qui sont des séquences atomiques d'actions gardées pouvant être exécutées par les tâches.

Nous allons maintenant examiner comment sont représentés en *C* ces différents objets, et leurs primitives de manipulation.

3.2 Les tâches

Une tâche est composée d'une spécification (*module*) et d'un corps (*body*). Un module est créé lors de sa déclaration dans le module immédiatement englobant (père), et son *corps* lui est attribué lors de l'initialisation de la tâche.

Lors de l'exécution, une tâche est représentée par un bloc de contrôle contenant:

- des informations à l'usage du NED: numéro du processus, site d'exécution, état d'activité, accès à la liste des descripteurs de transitions du processus...

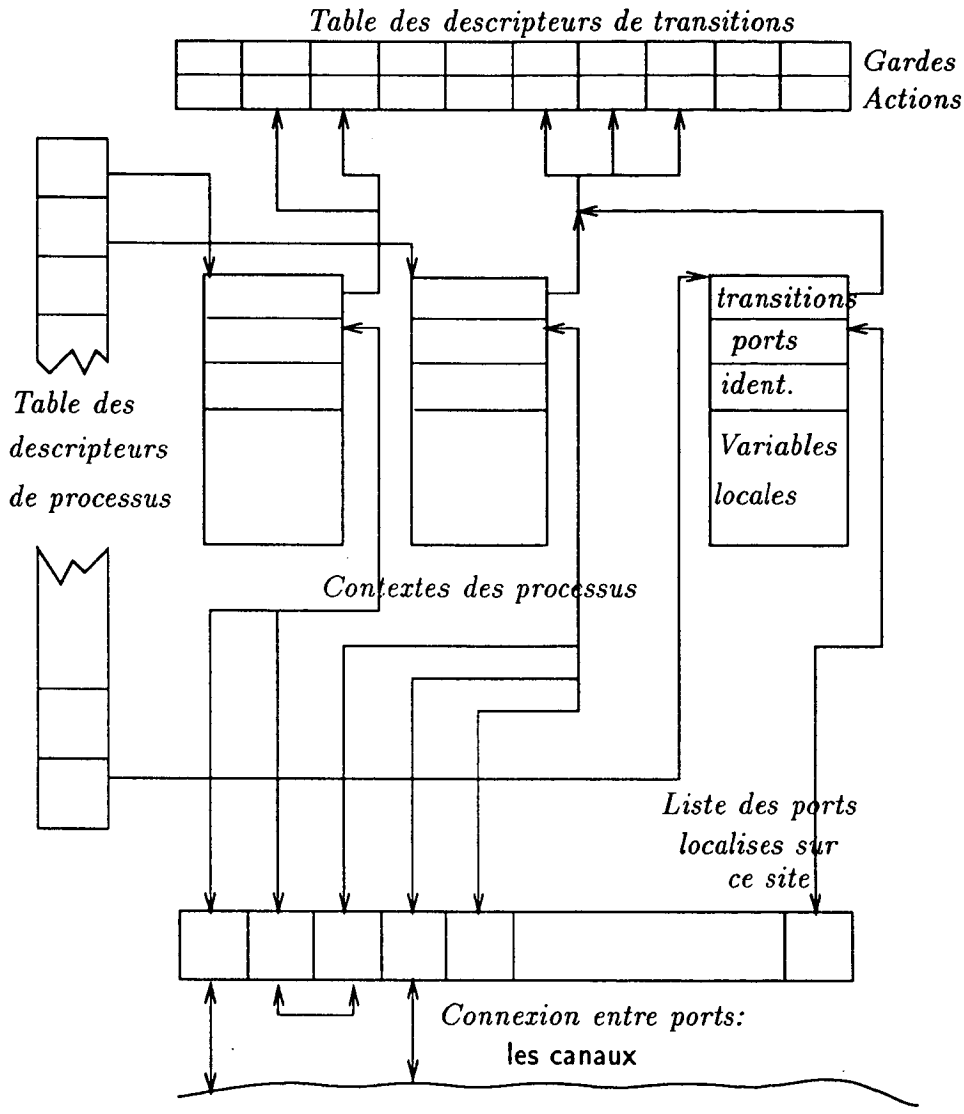


Figure 3 : Forme des structures de données sur un site

- les accès aux ports du module
- les accès aux descendants éventuels (pointeurs vers des blocs de contrôle)
- les variables locales du processus, y compris l'état de contrôle de l'automate, vu comme une variable locale de type énuméré.
- les paramètres passés à l'initialisation.

En *C*, ce bloc de contrôle est représenté par le type générique:

```
typedef struct context {
  ker_workspace kw; /*espace prive pour le NED, (invisible)*/
  union sur tous les modules de {
    liste de (ports *)  lp;
    union sur tous les body possibles de chaque module de {
       liste de (struct context *) lt;
       liste de (variables locales) lv;
       liste de (parametres) lpar;
    }
  }
} context;
```

3.3 Les ports

Les ports sont les accès vers l'extérieur des tâches. Ils sont traités de manière générique (type privé du NED) et connus simplement dans le code généré par un descripteur prédéfini.

Un port contient:

- une identité unique dans tout le système
- l'identité du site sur lequel est placé le port
- un pointeur vers le port auquel il est connecté (correspondant), si ce port se trouve sur le même site
- l'identité unique du correspondant

- l'identité du site sur lequel est placé le correspondant
- un pointeur vers la tâche dont dépend le port
- une file, non bornée *à priori*, contenant les messages reçus sur ce port

Remarque : l'identité unique d'un port est obtenue grâce à un compteur incrémenté de la même façon sur tous les sites du système pendant la phase de configuration, qui se déroule elle-même de la même manière sur tous les sites.

Le NED fournit les primitives pour créer les ports, les détruire (quand leur tâche n'est pas initialisée sur un site), les attacher et les connecter. Les ports restant sur un site sont rangés dans une liste qui sera parcourue lors de la réception d'un message externe, afin de retouver le port destinataire.

3.4 Les messages

Ce sont les objets qui circulent par l'intermédiaire des ports. Un message contient:

- une partie fixe donnant:
  - la taille totale du message
  - l'identificateur du port de destination
  - le type du message
- une partie mobile, générée par le compilateur, donnant les paramètres du message (son contenu)

Le NED fournit les primitives de manipulation des messages: création, destruction, copie, envoi local et distant, dépôt dans la file d'un port et réception de messages distants.

3.5 Les transitions

Une transition Estelle est un couple $< condition, action >$, où une condition est composée d'une liste (ev. vide) de clauses:

- *clause when* : présence d'un message sur un port donné du module

- *clause provided* : condition booléenne sur le contenu d'un message tête de file d'un port, ou sur les variables locales de la tâche

- *clause from* : condition sur l'état de contrôle de l'automate

- *clause priority* : priorité dynamique de la transition, par rapport aux autres transitions de la *même tache*

- *clause delay* : spécifiant l'intervalle de temps pendant lequel la transition peut être tirée

Une *action* est un éventuel changement d'état de l'automate, suivi d'un bloc Pascal (*begin...end;*) pouvant comporter des instructions spécifiques à Estelle (émissions de messages ect.).

Le langage Estelle offre la possibilité de regrouper sous un même texte des transitions différentes grâce à la clause ANY. Par exemple:

```
any i:1..maxports do
  when port[i].message   <action>
```

est équivalent à:

```
when port[1].message   <i:=1;action>
when port[2].message   <i:=2;action>
        ...
when port[maxports].message   <i:=maxports;action>
```

Remarque: Il est à noter que l'on peut imbriquer des clauses ANY.

Donc, à une ou plusieurs variables d'indice près, les parties *conditions* et *actions* sont communes à toutes les transitions d'un tel groupe de transitions, que nous appèlerons méta-transition. Une méta-transition est donc conceptuellement équivalente à un tableau multidimensionnel de transitions. Par souci de généralité, une transition simple sera vue comme une méta-transition réduite à un seul élément.

De manière générale, une transition au sens Estelle sera représentée par le numéro de sa méte-transition et une clé identifiant cette transition particulière dans sa méta-transition. En pratique cette clé ressemble à un accès à un tableau multi-dimentionnel et permet de positionner les indices des clauses ANY de la méta-transition.

Nous avons choisi de générer pour chacune de ces méta-transitions seulement deux fonctions C (évaluation de la garde et tir de la transition) acceptant en paramètres, en plus du contexte de la tâche, la clé d'identification de transition. Le segment de code est donc commun à toutes les transitions d'une méta-transition.

Evaluation d'une garde La fonction d'évaluation de la garde d'une méta-transition commence par positionner des variables temporaires correspondant aux indices des clauses ANY grâce à la clé passée en paramètre, puis à l'aide des primitives fournies par le NED, teste sur le descripteur de contexte les clauses spécifiées: présence de messages, état[3]... Si celles-ci sont toutes vérifiées, la fonction rend la priorité de la transition (évaluée dynamiquement à l'aide de la clause *priority*). Si l'une au moins des clauses n'est pas vérifiée, la fonction d'évaluation rend la valeur -1.

Dans le cas d'une transition spontanée sans clause de priorité, l'évaluation de la garde est simplement une fonction NOP qui rend la priorité minimale.

La partie *initialize* d'un module est représentée par une transition spontanée, tirée une seule fois lors de l'initialisation de la tâche.

Tir d'une transition La partie *action* est traduite par une fonction C, qui comme l'évaluation de la garde, commence par positionner les variables temporaires correspondant aux indices des clauses ANY à l'aide de la clé, puis exécute le bloc Pascal associé, en y incluant l'éventuel changement d'état (affectation sur la variable Etat de type énuméré (enum)).

Les instructions supplémentaires propres à Estelle sont traduites de manière à utiliser la *machine virtuelle Estelle* construite par le NED.

3.6 Traduction des instructions propres à Estelle

Détaillons la traduction des instructions Estelle, en commançant par celles qu'on ne peut trouver que dans les gardes.

When Cette clause est rencontrée dans les gardes des transitions pour tester la présence d'un message donné sur un port, et ouvrir la visibilité sur

[3]L'état de l'automate étant considéré comme une variable (de type énuméré) locale au contexte de la tâche, le test sur cet état n'est pas particularisé.

son éventuel contenu. Elle se traduit dans la fonction d'évaluation de la garde par:

```
{
  msg * m;
  if (((m = get_msg(NomPort)) == NIL) ||
    (m → msg_kind != TypeSpecifie)) return (-1);
    VariableTemp1 = m → ParametreFormel1;
    VariableTemp2 = m → ParametreFormel2;
                  ...
    VariableTempn = m → ParametreFormeln;
  /* Tests éventuels sur le type du message et ses paramêtres */
}
```

get_msg rend un pointeur sur le message en tête du port spécifié (*nil* si la file est vide). Le test de présence d'un message Estelle sur un port échoue donc si la file d'entrée est vide ou si le message présent n'est pas du type attendu. En cas de succès, les paramètres associés au message sont recopiés dans des variables locales ayant pour noms les identificateurs donnés dans le texte de la clause *when*.

From, Provided Ces clauses permettent de tester l'état de l'automate et de ses variables locales, elles sont traduites directement par les tests adéquats en *C*.

Priority Comme on l'a déjà vu, la priorité dynamique de la transition est retournée par la fonction d'évaluation (simple affectation de variable).

Delay Cette clause n'est pour l'instant pas implémentée.

Any Le " dépliage " d'une méta-transition engendrée par une clause *any* est réalisé comme expliqué au paragraphe précédent.

Les instructions suivantes ne peuvent être rencontrées que dans les parties *action* des transitions.

Init Instruction d'initialisation de module, qui permet de lui associer un corps (*body*). Sa syntaxe est la suivante:

$$init < InstanceDeModule > with < body > (paramètres)$$

Dans le cadre de Echidna (Estelle statique), cette instruction n'est autorisée que dans la partie initialisation d'un module passif. Cette instruction est traduite en *C* par:

```
if (mk_proc(& contexte, Site, TailleContexte, ListeTransitions,
                NombreSousModules,Nombreports) == SurceSite) {
  contexte → ParametreFormel1 = ParametreEffectif1;
  contexte → ParametreFormel2 = ParametreEffectif2;
                    ...
  contexte → ParametreFormeln = ParametreEffectifn;
init_proc(contexte);/*tire la transition d'initialisation de la tâche*/
}
```

mk_proc rend un pointeur sur le contexte du processus qu'il vient d'allouer. Si le processus n'est pas destiné à s'exécuter sur le site physique, la copie des paramètres effectifs et l'allocation des variables locales ne sont pas effectuées (seule la structure de la hiérarchie des processus est temporairement gardée pour calculer une identité unique pour les processus et ports).

Connect et Attach *Connect* permet à un module de connecter entre eux les ports de ses sous-modules, et *Attach* de " raccorder " le port d'un sous-module à un port du module englobant. Ces instructions sont implémentées par appel direct des primitives du NED leur correspondant. Signalons ici une restriction temporaire concernant la connexion de ports d'un module: nous imposons que les modules concernés soient initialisés avant qu'on puisse connecter leurs ports.

Output Cette instruction permet l'envoi d'un message sur un port. Sa syntaxe est la suivante:

$$output < NomDePort > . < NomInteraction > (ParamètresDuMessage)$$

Elle est traduite par la création d'un message avec le bon sélecteur de type, l'affectation des paramètres du message, puis l'envoi effectif du message sur le port désigné:

```
{
  msg * m;
  m = mk_msg(TailleMessage);
    m → ParametreFormel1 = ParametreEffectif1;
    m → ParametreFormel2 = ParametreEffectif2;
                    ...
    m → ParametreFormeln = ParametreEffectifn;
  send_msg(NomPort,m);
}
```

Nextstate, To Ces instructions permettent de modifier l'état principal de l'automate. Elles sont traduites par des affectations directes sur la variable *Etat* du contexte de la tâche.

All L'instruction *all i : BorneInf..BorneSup do < instruction >* permet de réaliser une boucle non directiviste, sans déclaration préalable de la variable de boucle. Nous l'avons implémentée par une boucle classique, avec variable de boucle interne à un bloc *C*.

Trace Cette instruction est notre seule extension au langage Estelle (qui ne possède pas d'instructions d'entrée-sorties). Sa syntaxe est celle d'un *writeln* Pascal, mais sa sémantique est propre au contexte de l'expérimentation [2]. En effet, pour ne pas perturber l'expérimentation d'un algorithme distribué sur machine parallèle, il est souvent nécessaire de ne pas effectuer d'entrée-sorties supplémentaires (les messages de trace viennent perturber les messages de l'application). C'est pourquoi une option de compilation permet de forcer l'instruction *trace* à effectuer les entrée-sorties en mémoire, afin d'être transférées physiquement seulement après que l'expérience soit terminée.

3.7 La structure du code généré

Le squelette du code *C* généré par notre compilateur est donc le suivant:

```
# include < echidnak.h > /* déclarations exportées par le NED */
# define ( liste des constantes déclarées en Estelle par const )
typedef ( liste des types déclarés en Estelle par type )
typedef struct {
  msg_type msg_kind;
  union {
      /* sur les paramètres de chaque type de message possible */
        } u;
} msg;

typedef union s_context {
  ker_workspace kw; /*espace prive pour le NED, (invisible)*/
  union sur tous les modules de {
    liste de (ports *) lp;
    union sur tous les body possibles de chaque module de {
      liste de (struct context *) lt;
      liste de (variables locales) lv;
      liste de (parametres) lpar;
    }
  }
} context;


/* vient ensuite le texte des méta-transitions, pour i de 1 à n */
int guard_i(a, p)
  int a;
  context * p;
{
  /* Utilisation de a pour positionner les variables de ANY */
  /* code de l'évaluation de la garde */
  si passante retour (priorité) sinon retour (-1);
}
int action_i(a, p)
  int a;
  context * p;
```

```
{
  /* Utilisation de a pour positionner les variables de ANY */
  /* code du tir de la transition */
}


/* Définition des variables exportables vers le NED */
/* le tableau transition[] donne la correspondance entre le texte */
/* d'une méta-transition et son numéro pour le NED */
int (* transition[])() = {
        guard_1, action_1,
        guard_2, action_2,
        ...
        guard_n, action_n
};
/* Le tableau transany[] donne la borne de la variation des clés */
/* des clauses any pour chaque transition */
int transany[n] = { nany_1, ..., nany_n };


void configuration()
{
  /* Réalise la création du module englobant (spécification) */
  /* et en appelle la transition d'initialisation */
```

3.8 L'accès aux variables et renommage

Afin de générer du code réentrant pour les transitions, les variables des *body* Estelle doivent être accédées à travers la référence au contexte d'un processus. Nous avons choisi une solution simple à ce problème en utilisant le pré-processeur *C* : avant de générer les transitions d'un processus, les variables sont *définies* comme des références par un pointeur de contexte à l'union de la structure codant les *body* Estelle. Ces définitions sont effacées après la génération.

Le renommage de certains objets est aussi nécessaire, puisque la structure de bloc du langage Estelle est partiellement détruite dans le code généré (voir

la structure du code généré). Pour ne pas trop détruire la lisibilité du code *C*, nous avons choisi de renommer seulement si deux identificateurs de portées différentes avaient la même représentation (pour une portée donnée, le clash est détecté comme une double définition par le compilateur). Dans ce cas l'identificateur est préfixé par un numéro identifiant de façon unique son niveau de déclaration.

Enfin, pour éviter les clashs avec les identificateurs des fonctions *C* standard et de noyau d'exécution, les identificateurs du source sont écrits avec leur première lettre en majuscule.

3.9 La traduction du Pascal

Syntaxiquement parlant, les langages Pascal et C ne diffèrent que très peu, et la traduction manuelle du premier dans le second paraît assez facile. La situation se complique singulièrement au niveau sémantique.

Notre traduction s'inspire des principes du travail de Per Bergsten à l'université de Gothenburg en Suède, qui a écrit le traducteur PtoC et l'a mis dans le domaine public de la recherche.

On liste ici un certain nombre de problèmes qu'il a fallu résoudre, et on présente brièvement les solutions choisies. Les constructions Pascal qui ne sont pas citées avaient un équivalent direct en C dans le contexte simplifié considéré, où l'emboîtement des procédures n'est pas permis.

3.9.1 Les constantes

Les déclarations de constantes sont traitées de deux façons différentes. Les constantes qui ne sont pas des chaînes sont simplement définies en utilisant l'instruction *# define* du préprocesseur *C*. Les chaînes, elles, sont converties en tableaux statiques de caractères afin de ne pas dupliquer inutilement les chaînes de caractères dans le code généré. Donc

```
const
      c1 = 123 ;
      c2 = 'c1';
      c3 = -c1 ;
```

est traduit en

```
\# define c1        123
static char c2[] = "c1";
\# define c3        -c1
```

3.9.2 Types et variables

Les types sont assez faciles à traduire. Un traitement particulier doit être appliqué pour coder les intervalles, les tableaux, les records variants, les pointeurs récursifs et les types ensemble.

Le source :

```
type
      idx   =  1 .. 15;
      enum  =  (bleu, blanc, rouge);
      tab   =  array [idx, idx] of enum;
      ens   =  set of char;
      prec  =  ^noeud;
      noeud =  record
                 next : prec;
                 case b : boolean of
                   false : (c : enum);
                   true  : (d : idx);
               end;
var   P : prec;
      s : ens;
begin
      s := ['b'..'j']*['c'..'p'] ; (*intersection*)
      new (P, true)
end;
```

devient :

```
      typedef char      Idx;
      typedef enum { Bleu, Blanc, Rouge }       Enum;
      typedef struct { struct {Enum  a[15 - 1 + 1];} a[15 - 1 + 1];} Tab;
      typedef struct { setword      s[9]; }     Ens;
      typedef struct s1 * Prec;
      typedef struct s1 {
```

```
            Prec      Next;
            Boolean     B;
            union {
                  struct  {
                        Enum      C;
                  } v1;
                  struct  {
                        Idx       D;
                  } v2;
            } u;
      }     Noeud;

      Prec     P;
      Ens      S;

      setncpy(S.s, inter(conset[0], conset[1]), sizeof(S.s));
      claimset();
      P = (struct s1 *)malloc((unsigned)
            (unionoffs(P, u.v2.D) + sizeof(P->u.v2)));
}

static setword  q0[] = {
  8,
  0x0000,  0x0000,  0x0000,  0x0000,  0x0000,  0x0000,
  0xFFF8,  0x0001
};
static setword  q1[] = {
  7,
  0x0000,  0x0000,  0x0000,  0x0000,  0x0000,  0x0000,
  0x07FE
};
setword  *conset[] = { q1,  q0 };
```

L'explication est la suivante :

- Les types intervalle sont codés par les types arithmétiques C standard, en fonction d'une table de correspondance inclue dans le compilateur.

Cette table est explorée du haut vers le bas jusqu'à trouver un type C qui peut contenir l'intervalle. Pour éviter des problèmes de conversion de type dans les expressions, seuls les types char, short et long sont considérés. Cette façon de faire permet souvent de générer un code exécutable plus petit que celui obtenu par une compilation Pascal.

- Les tableaux C ont une sémantique bien particulière (adresses), ce qui oblige l'encapsulation des tableaux Pascal dans une structure C. Cette structure possède un unique membre, noté a. Les tableaux multidimensionnels sont éclatés en une suite imbriquée de tableaux simples comme l'autorise la sémantique de Pascal. On doit aussi tenir compte que les tableaux C sont indicés à partir de 0. On laisse le calcul des expressions constantes au soin du compilateur C.

- Les types énumérés Pascal ont leur équivalent C.

- Les *records* et leurs variantes sont traduits par des *struct* et des *unions*. Comme C impose que tous les membres des unions soient nommés, il faut générer des noms pour les variantes du record. Les champs variants sont notés u, et les sous-champs $vxxx$ où xxx est un entier positif unique calculé par le compilateur. Le problème de l'allocation dynamique des structures pointées est difficile. Le calcul de la taille est effectué par une macro-C (*unionsoffs*).

- Les pointeurs Pascal sont convertis en pointeurs C. Dans le cas (fréquent) où le type pointeur référence un type non encore défini, on doit générer un nom intermédiaire ($sxxx$) pour ce type référencé.

- Enfin, les types ensemble Pascal n'ayant pas de contre-partie C, il faut les coder en tableaux de bits et utiliser des fonctions d'accès particulières pour tous les opérateurs de manipulation d'ensembles. Le premier élément dans le tableau des mots donne la taille de l'ensemble (en mots). Les autres contiennent la valeur des bits. La taille du tableau est calculée sous l'hypothèse que le type *unsigned short* sur le calculateur cible peut contenir 16 bits. Les résultats intermédiaires sont stockés dans une zone statique (*conset*) de taille fixe.

3.9.3 Les instructions

Les seules instructions nécessitant quelque attention sont les instructions *with* et *for*. L'instruction with est traduite en une instruction composée où des variables pointeurs (nommées $wxxx$), référençant les records correspondant, sont declarées et initialisées. A l'intérieur du champ de visibilité du with, les champs accessibles sont renommés pour inclure le pointeur. Ceci a pour effet que l'adresse du record n'est évaluée qu'une seule fois comme le Pascal standard l'impose. Le code :

```
var r : record
      c1 : integer;
      c2 : char;
end;
begin
      with r do begin
            c1 := 10;
            c2 := 'd'
      end
end;
```

devient :

```
struct s1 {
      Integer      C1;
      Char      C2;
} R;

{
      register struct s1 *w1 = &R;

      w1->C1 = 10;
      w1->C2 = 'd';
}
```

L'instruction *for* Pascal n'a pas la même sémantique que la construction C de boucle. Pascal impose la sortie de boucle lorsque la borne supérieure a

été atteinte, les bornes ne devant être évaluées qu'une seule fois. Alors que la boucle C est terminée lorsque la condition de boucle devient fausse. Ceci implique que :

```
for (i = 0; i <= exp; i++) ;
```

ne se comporte pas toujours comme :

```
for i := 0 to exp do ;
```

la première version bouclant indéfiniment si exp vaut *maxint* ou si exp est l'expression *i*. Pour cette raison, les bornes de la boucle sont stockées dans des variables locales (nommées *bxxx*) et l'expression de sortie est évaluée à la fin. On obtient la traduction suivante :

```
Integer   I;

{
      Integer   b1 = 0,
      b2 = Exp;

      if (b1 <= b2)
        for (I = b1; ; I++) {
              if (I == b2) break;
        }
}
```

3.9.4 Expressions

- La traduction de l'affectation est immédiate puisque les tableaux sont encapsulés dans des structures.
- L'indexation *tab[i]* est traduite en *Tab.a[index(i,clo,chi)]*. *index* est une macro C qui vérifie à l'exécution que *i* est bien compris dans les bornes clo et chi (calculées par le compilateur) et rend *i* - *clo* dans ce cas.
- La sélection (identique Pascal et C) *a.b* est traduite en *A.vxxx.B* si *b* est un champ variant.
- Les références aux pointeurs, ainsi les paramètres passés en référence (var) sont préfixés par un astérisque.

4 Mise en œuvre du compilateur

Le compilateur a été complètement écrit en Pascal et se présente sous la forme d'environ 7000 lignes de source. L'analyse est effectuée classiquement de façon récursive descendante.

Il consiste en trois procédures principales qui réalisent successivement :

- la conversion du source Estelle en un arbre syntaxique,
- des transformations sur cet arbre (destructuration) pour préparer la génération du code C,
- et la traversée finale de l'arbre pour imprimer les constructions C correspondant à chaque nœud.

Estelle complet est analysé : le sous-ensemble utilisé est défini par filtrage lors de l'analyse syntaxique. La plupart des vérifications sémantiques sont effectuées, soit statiquement lors de la compilation, soit dynamiquement à l'exécution. Aucune récupération d'erreur n'est tentée.

Le résultat obtenu est relativement efficace (une centaine de lignes par seconde) et s'est avéré aisément portable, y compris sur des petites machines (PC).

5 Conclusion

Le concept d'expérimentation d'algorithmes distribués sur machines parallèles nous semble intéressant dans le processus général de validation. Expérimenter devrait permettre de mieux comprendre et comparer les performances des algorithmes dans un environnement parallèle réel, mettant en évidence des phénomènes difficiles à faire apparaître lors d'une vérification ou d'une simulation.

L'outil Echidna a été conçu pour cette activité, et développé pour les hypercubes Intel iPSC/1 et iPSC/2. Il a été transporté sur des réseaux de stations SUN (sur TCP/IP), sur l'ordinateur Gould et sur micro PC. Nous projetons d'attaquer le monde des Transputers.

Du fait que le langage d'entrée d'Echidna est un sous-ensemble du langage Estelle, norme ISO, langage pour lequel de nombreux outils ont

été développés, on peut, à partir de la même description formelle, non seulement expérimenter, mais aussi simuler, vérifier ... Cette approche a déjà été explorée avec succès sur des problèmes réels comme l'étude des algorithmes synchroniseurs à l'Irisa [1], ou la validation d'un protocole de diffusion fiable MAC/Tr dans le cadre du projet Esprit Delta4 [12].

De plus, la possibilité qu'offre Echidna de produire du code parallèle à partir d'un langage d'assez haut niveau, est très appréciée des programmeurs des machines parallèles, qui sur les hypercubes par exemple, programment directement en C ou Fortran (sans vision globale de la distribution). Si cet intérêt est confirmé dans le futur, il sera envisageable de s'appliquer à traiter le langage Estelle dans son entier en réglant le problème (difficile) de la mise en œuvre distribuée du dynamisme du langage.

En dehors de l'activité de validation, nous pensons utiliser Echidna pour simuler du parallélisme massif (par exemple des algorithmes cellulaires), pour enseigner l'algorithmique répartie et même pour générer des implantations parallèles prototypes.

Bibliographie

[1] M. Adam, Ph. Ingels, and M. Raynal. *Algorithmes Distribués synchrones et systèmes répartis asynchrones : concepts, mises en œuvre et expérimentations.* Technical Report 411, IRISA University of Rennes, June 1988. 27 p.

[2] M. Adam, Ph. Ingels, and M. Raynal. The meaning of synchronous distributed algorithms run on asynchronous distributed systems. In *The Third International Symposium on Computer and Information Sciences, Izmir*, November 1988.

[3] G.V. Bochmann. Usage of protocol development tools : the results of a survey. In 7^{th} *IFIP International Workshop on Protocol Specification, Testing, and Verification, Zurich, Suisse*, North Holland, May 1987.

[4] L. Bomans and D. Roose. *Benchmarking the iPSC/2.* Technical Report 114, Katholieke Universiteit Leuven, October 1988.

[5] S. Budkowski and P. Dembinski. An introduction to Estelle: a specification language for distributed systems. *Computer Networks and ISDN Systems*, 14:3–23, 1987.

[6] J.P. Courtiat, P. Dembinski, R. Groz, and C. Jard. Estelle : un langage ISO pour les algorithmes distribués et les protocoles. *Technique et Science Informatique*, 6(2), 1987.

[7] C. Diaz, M. Vissers and J.P. Ansart. Sedos: software environment for the design of open distributed systems. In *Proceedings of the Esprit '85 week*, North Holland, 1985.

[8] M. Heath. The hypercube: a tutorial overview. In Michael T. Heath, editor, *Hypercube Multiprocessors 1986*, pages 7–11, Oak Ridge National Laboratory, 1986.

[9] ISO. Estelle *: a Formal Description technique based on an Extented State Transition Model.* ISO TC97/SC21/WG6.1, 1986.

[10] C. Jard, R. Groz, and J.F. Monin. Development of VEDA : a prototyping tool for distributed algorithms. In *IEEE Trans. on Software Engin.*, March 1988.

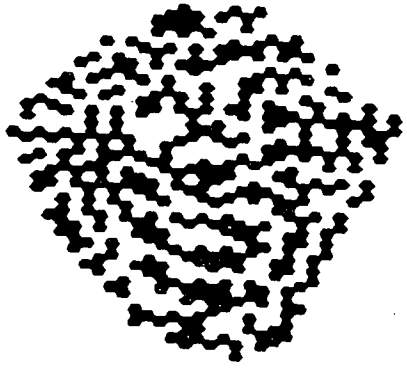
[11] C. Jard and J.-M. Jézéquel. Outils pour l'expérimentation d'algorithmes distribués sur machines parallèles. In *Actes du Colloque C^3 d'Angoulème*, GRECO C^3/CNRS, December 1988.

[12] N. Plouzeau. *A Validation Experiment of the MAC/TR Protocol.* Technical Report 88021, Bull, Centre de recherche groupe, November 1988.

[13] S. Vuong, A. Lau, and R. Chan. Semiautomatic implementation of protocols using an Estelle-C compiler. *IEEE Transactions on Software Engineering*, SE-14(3):384–393, March 1988.

```
{ A game of life in parallel }
specification life; default individual queue;
const Nx = 40; Ny = 25; Nx_1 = 39; Ny_1 = 24;          { Nx * Ny processes }
type  cx = 0 .. Nx_1; cy = 0 .. Ny_1; Neigh = (N, E, S, O);     { 2-D grid  }
      state_typ = boolean;
channel news (caller, called); by caller : changed;
   module cel process (id : integer; initial_state : state_typ);
        ip I : array [Neigh] of news (caller);          { ! changed (st)    }
           R : array [Neigh] of news (called);          end;
   body algo for cel;
   var env, my_state : state_typ;
   initialize begin my_state := initial_state; env := false end;
   trans provided (my_state <> env) begin               { alternate state   }
                my_state := not my_state;               { broadcast the change }
                trace (id, ord(my_state));              { tracing           }
                all v : Neigh do  output I [v]. changed  end;
   trans any c : Neigh do when R [c]. changed begin env := not env end;
   end;
modvar C : array [cx,cy] of cel;
initialize var gx : cx; gy : cy; begin
        gx := Nx div 2; gy := Ny div 2;                 { seed }
        all x : cx do all y : cy do
                init C [x,y] with algo (y*Nx + x, (x = gx) and (y = gy));
        all u : cx do all v : cy do begin
                connect C [u,v]. I [N] to C [u,(v+1) mod Ny]. R [S];
                connect C [u,v]. R [N] to C [u,(v+1) mod Ny]. I [S];
                connect C [u,v]. I [O] to C [(u+1) mod Nx,v]. R [E];
                connect C [u,v]. R [O] to C [(u+1) mod Nx,v]. I [E] end end;
end.
```

Snapshot of this algorithm execution on the *iPSC/2*, coupled to a graphical tool



NAME

ec, Estelle compiler for Unix and iPSC; ECHIDNA project.

SYNOPSIS

ec [-sngifcCtG] [-m file] [-l file] [-o object_file] [estelle_file]

DESCRIPTION

The ec program translate an Estelle formal description to C data structures, and then compile and link them with a system dependant kernel, to produce code for various architectures (see below).
If an error is found in the source, your favourite editor (as defined in the environment variable EDITOR, or vi otherwise) is invoked on the errored line. When you leave it, compilation is resumed.
If no argument is given, the last version of this page is displayed.

OPTIONS

-s Target system is the sun (default).
-n Target system is a network of Sun (TCP/IP)
-g Target system is the Gould: produce code for the Gould
-i Target system is the iPSC.
-f Target system is the FPS-T20
-m Link with an extra object or C file module, endding with .o or .c (can be repeted)
-l Link with a library file, generaly beginning with "l" (can be repeted)
-c Perform only Estelle to C translation.
-C Perform C compilation and link with the kernel.
-t Don't produce extra-code to check index range validity.
-G Debug option for use with dbx.
-o object_file : Names the executable program "object_file".

USAGE

You can provide the following optional parameters to your executable Estelle program:
-dxxxx The duration 'nnnn' of the execution in seconds (default=0 means infinity)
-sxxxx A seed (integer) xxxx for the random generator
-u Prevent the bufferisation of trace output
-q Make execution quiet : no header nor warning displayed
-a Simulate an asynchronous execution of processes within a node
-D Select a deterministic semantic for the execution: the first firable transition of a process is actually fired, priorities are ignored.

To load and run your program, see system specific instructions.
For exemple:
- On a Sun, enter: program_name [arg1 ..[argn]]
- On a Sun network: getsun sun1 sun2 ..sunN
  loadsun "program_name [arg1 ..[argn]]"
- On the ipsc/2: getcube [cubesize] #see ipsc manual
  load program_name [arg1 ..[argn]]
- On the Gould, rlogin gould; cd directory and like on a Sun.

If no execution duration is specified, a <CTRL C> must be entered to stop the program, and remaining processes should be killed.

BUGS

- unrecognized parameters on command-line are ignored
Bugs and others problems are to be reported to:

J-M Jezequel TB105 IRISA poste 537 e-mail: jezequel@irisa.fr
or:
C. Jard TB107 IRISA poste 538 e-mail: jard@irisa.fr

```
/*
%Z% %H%(AOP-IRISA) --ECHIDNA Compiler-- Version %I%
*/
/*
**      Code derived from specification Life
*/
                                /* Kernel declarations */
# include "echidnak.h"
                                /* User   constants */
# define Nx 40
# define Ny 25
# define Nx_1 39
# define Ny_1 24
                                /* User   types */
typedef char    Cx;
typedef char    Cy;
typedef enum { N, E, S, O }     Neigh;
typedef Boolean State_typ;
                                /* User type for interactions */
typedef enum {  Changed } msg_type;
typedef struct {
        msg_type        msg_kind;
        union {
                                /* Channel News */
                struct {
                        int     dummy;
                }       m_Changed;
        } u;
} msg;
                                /* User type for process contexts */
typedef union s_context {
                                /* Block module of Life */
# define nbports_Life         0
        struct {
                union {
                                /* Local vars of Life */
# define nbsons_Life      1000
                        struct {
                                struct { struct { union s_context *
                                        a[Ny_1 + 1]; } a[Nx_1 + 1]; }  v_i3_C;
                        }       Life;
                }       u;
        }       Life;
                                /* Block module of Cel */
# define nbports_Cel          8
        struct {
                struct { address        a[4]; } I;
                struct { address        a[4]; } R;
                union {
                                /* Local vars of Algo */
# define nbsons_Algo    0
                        struct {
                                State_typ       v_Env, v_My_state;
                                Integer v_Id;
                                State_typ       v_Initial_state;
                        }       Algo;
                }       u;
        }       Cel;
} context;
                                /* User procedures and transitions */
# define numlt_Life     1
# define nbt_Life       1
# define numlt_Algo     2
# define nbt_Algo       3
                                /* Transitions for Life */
# define i3_C   p->Life.u.Life.v_i3_C

# define guard_1        nop
int action_1(a,p)
        int     a;
        context * p;
        {
        Cx      Gx;
        Cy      Gy;
        Gx = Nx / 2;
        Gy = Ny / 2;
        {
                Cx      X;
                Cx      b1 = 39,
                        b2 = 0;
                if (b1 >= b2)
                        for (X = b1; ; X--) {
                                {
                                        Cy      Y;
                                        Cy      b3 = 24,
                                                b4 = 0;
                                        if (b3 >= b4)
                                                for (Y = b3; ; Y--) {
        if (mk_proc(&i3_C.a[index(X, 0, 39 )].a[index(Y, 0, 24 )],
        Y * Nx + X, sizeof(p->Cel.u.Algo),
        numlt_Algo, nbt_Algo, nbsons_Algo, nbports_Cel)) {
                i3_C.a[index(X, 0, 39 )].a[index(Y, 0, 24 )]->
                Cel.u.Algo.v_Id = Y * Nx + X;
                i3_C.a[index(X, 0, 39 )].a[index(Y, 0, 24 )]->
                Cel.u.Algo.v_Initial_state = (X == Gx) && (Y == Gy);
                (void)init_proc( i3_C.a[index(X, 0, 39 )].a[index(Y, 0, 24 )]);
                                                }
                                                if (Y == b4) break;
                                        }
                                }
                                if (X == b2) break;
                        }
        }
        {
                Cx      U;
                Cx      b5 = 39,
                        b6 = 0;
                if (b5 >= b6)
                        for (U = b5; ; U--) {
                                {
                                        Cy      i14_V;
                                        Cy      b7 = 24,
                                                b8 = 0;
                                        if (b7 >= b8)
                                                for (i14_V = b7; ; i14_V--) {
        { if ((i3_C.a[index(U, 0, 39 )].a[index(i14_V, 0, 24 )] ==
            (context *)0) ||
            ( i3_C.a[index(U, 0, 39 )].a[index((i14_V + 1) % Ny, 0, 24 )] ==
            (context *)0)) ker_error(13);
          connect_ip(&(i3_C.a[index(U, 0, 39 )].a[index(i14_V, 0, 24 )]->
          Cel.I.a[index((int)(N), 0, 3 )]),
          &(i3_C.a[index(U, 0, 39 )].a[index((i14_V + 1) % Ny, 0, 24 )]->
          Cel.R.a[index((int)(S), 0, 3 )]));}
        { if ((i3_C.a[index(U, 0, 39 )].a[index(i14_V, 0, 24 )] ==
            (context *)0) ||
            ( i3_C.a[index(U, 0, 39 )].a[index((i14_V + 1) % Ny, 0, 24 )] ==
            (context *)0)) ker_error(13);
          connect_ip(&(i3_C.a[index(U, 0, 39 )].a[index(i14_V, 0, 24 )]->
          Cel.R.a[index((int)(N), 0, 3 )]),
          &(i3_C.a[index(U, 0, 39 )].a[index((i14_V + 1) % Ny, 0, 24 )]->
          Cel.I.a[index((int)(S), 0, 3 )]));}
        { if ((i3_C.a[index(U, 0, 39 )].a[index(i14_V, 0, 24 )] ==
            (context *)0) ||
```

```
            ( i3_C.a[index((U + 1) % Nx, 0, 39 )].a[index(i14_V, 0, 24 )] ==
            (context *)0)) ker_error(13);
          connect_ip(&(i3_C.a[index(U, 0, 39 )].a[index(i14_V, 0, 24 )]->
          Cel.I.a[index((int)(O), 0, 3 )]),
          &(i3_C.a[index((U + 1) % Nx, 0, 39 )].a[index(i14_V, 0, 24 )]->
          Cel.R.a[index((int)(E), 0, 3 )]));}
        { if ((i3_C.a[index(U, 0, 39 )].a[index(i14_V, 0, 24 )] ==
            (context *)0) ||
            ( i3_C.a[index((U + 1) % Nx, 0, 39 )].a[index(i14_V, 0, 24 )] ==
            (context *)0)) ker_error(13);
          connect_ip(&(i3_C.a[index(U, 0, 39 )].a[index(i14_V, 0, 24 )]->
          Cel.R.a[index((int)(O), 0, 3 )]),
          &(i3_C.a[index((U + 1) % Nx, 0, 39 )].a[index(i14_V, 0, 24 )]->
          Cel.I.a[index((int)(E), 0, 3 )]));}
                                                  if (i14_V == b8) break;
                                              }
                                  }
                                  if (U == b6) break;
                              }
        }
}
# define nany_1 1
                                  /* Transitions for Algo */
# define Id     p->Cel.u.Algo.v_Id
# define Initial_state  p->Cel.u.Algo.v_Initial_state
# define Env    p->Cel.u.Algo.v_Env
# define My_state     p->Cel.u.Algo.v_My_state
# define guard_2        nop
int action_2(a,p)
        int     a;
        context * p;
        {
        My_state = Initial_state;
        Env = False;
}
# define nany_2 1
int guard_3(a,p)
        int     a;
        context * p;
        {
        if (    ((My_state != Env))
        )
        return(0);
        return(-1);
}
int action_3(a,p)
        int     a;
        context * p;
        {
        My_state = (Boolean)(!My_state);
        (void)sprintf(cbuf,"%10d%10d\n", Id, (unsigned)(My_state)),update();
        {
                Neigh   i8_V;
                Neigh   b9 = 3,
                        b10 = 0;
                if ((int)(b9) >= (int)(b10))
                        for (i8_V = b9; ; i8_V = (Neigh)((int)(i8_V)-1)) {
                                /* Send message Changed */
                                {
                                        msg * m;
                                        m = (msg *) mk_msg(sizeof(msg_type) +
                                            sizeof(m->u.m_Changed));
                                        m->msg_kind = Changed;
                                        send_msg(p->Cel.I.a[index((int)(i8_V),
                                                        0, 3 )], m);
                                }
                                if (i8_V == b10) break;
                        }
        }
}
# define nany_3 1
int guard_4(a,p)
        int     a;
        context * p;
        {
        Neigh   i9_C;
        msg     *m;
        i9_C =  a;
        if (((m = get_msg(p->Cel.R.a[index((int)(i9_C), 0, 3 )])) == (msg *)NIL) ||
        (m->msg_kind != Changed)) return(-1);
        return(0);
        return(-1);
}
int action_4(a,p)
        int     a;
        context * p;
        {
        Neigh   i9_C;
        msg     *m;
        i9_C =  a;
        m = (msg *)get_msg(p->Cel.R.a[index((int)(i9_C), 0, 3 )]);
        rm_msg(p->Cel.R.a[index((int)(i9_C), 0, 3 )]);
        Env = (Boolean)(!Env);
}
# define nany_4 4
# undef Id
# undef Initial_state
                                  /* Transition accesses */
int (* transition[])() = {
        guard_1, action_1,
        guard_2, action_2,
        guard_3, action_3,
        guard_4, action_4
};
int transany[4] = {nany_1, nany_2, nany_3, nany_4};
                                  /* Main configuration */
void configuration()
{ context * p; address root;
        (void)mk_proc(&root, -1,
        sizeof(p->Life.u.Life),
        numIt_Life, nbt_Life, nbsons_Life,0);
        setup_proc( init_proc( root ));
}
/*
**      End of program code
*/
```

```
#define version "@(#)JM Jezequel (ADP-IRISA)  echidnak.h 2.8 1/16/89 21:20:15"
/* contains all the public declarations of constants and types */

/*
**      General type declaration
*/

typedef int ** address; /* general untyped pointer */
typedef char    Boolean;
# define False (Boolean)0
# define True (Boolean)1
typedef char    Char;
typedef int     Integer;
# define Maxint (int)( (1 << ((sizeof(int)*8)-1) ) -1)
typedef double  Real;


/*
**      System primitives
*/

extern void  exit();
extern char *sprintf();
extern void update();
extern char *cbuf;
extern double Time();

/*
**      Kernel primitives
*/

extern void ker_error();
extern void ker_warning();
extern Integer randomint();
extern Real randomrel();
extern address mk_msg();
extern void send_msg();
extern address get_msg();
extern void rm_msg();
extern void connect_ip();
extern void attach_ip();
extern int  nop();
extern int  mk_proc();
extern address init_proc();
extern void setup_proc();


# define LINE __LINE__
extern  void caseerror();
extern int trunc();
extern int round();
extern void abort();
extern double exp();
extern double log();
extern double pow();
extern double sin();
extern double cos();
extern double tan();
extern double atan();
extern double sqrt();
extern double fabs();

/*
**      Definitions for pointers
*/

# define unionoffs(p, m)  (((long)(&(p)->m))-((long)(p)))
# define NIL 0
extern char *malloc();
extern void free();
extern char *strncpy();

/*
**      Definitions for set-operations
*/

# define claimset() (void)currset(0, (setptr)0)
# define newset() currset(1, (setptr)0)
# define saveset(s) currset(2, s)
# define setbits 15
typedef unsigned short  setword;
typedef setword *       setptr;
extern Boolean  member(), le(), ge(), eq(), ne();
extern setptr   u_Union(), diff();
extern setptr   insmem(), mksubr();
extern setptr   currset(), inter();
extern setptr   tmpset;
extern setptr   conset[];
extern void     setncpy();
extern char *strncpy();
extern int strncmp();
# define cmpstr(x, y) strncmp((x), (y), sizeof(x))

/*
**      Compilation options to check index out of range
*/
#ifndef UNCHECK
# define index(exp,lo,hi) ( (exp<lo)||(exp>hi) ) ? ker_error(7),0 : (exp-lo)
#else
# define index(exp,lo,hi) (exp-lo)
#endif
```

```
#define v_kernel_h "@(#)JM Jezequel (ADP-IRISA)  kernel.h 2.3 11/8/88 17:24:27"
/*
**  Definitions for processes
*/

# define maxpr 1023 /* max number of actual processes on a site */
# define evalue 0
# define tir 1
# define setany 2


typedef address memblock[100];/* size is of little interest, as the actual size
                                 is computed dynamycally */

typedef struct workspace {
  int  number, site;
  int  nbips, nbsubmod;
  int  firsttrans, nbtrans;
  Boolean active;
  memblock context;   /* store the 'context' as generated for each module, with
                   first nbips (address)ips, then nbsubmod (address)sub-modules,
                   and lastly paramaters and local variables of the module body
                   so, workspace.varloc[0] is the first ip of the module    */
                    /* the pointer '* context' points here, so a correction
                   is needed to acces the actual workspace. This is made by
                   the 'cont2ws' macro.                                     */
  } workspace;

# define sws (sizeof(workspace)-sizeof(memblock))  /*actual size of workspace*/
# define cont2ws(c) (workspace *)( ((long)(c)) - sws )

/*
**  Definitions for interactions (messages)
*/

typedef struct msg {
  int  msgsize;
  int  idipdest;
  memblock textmsg;
}  msg;

# define smsg (sizeof(msg)-sizeof(memblock))
# define a2msg(c) (msg *)( ((long)(c))-smsg )


/*
**  Definitions for interactions points
*/
typedef struct ip * star_ip;

typedef struct fifo {
  address  m;
  struct fifo *  l;
  star_ip  pt;
}  fifo;

typedef struct ip {
  int  identity, site;
  struct ip *  corresp;
  int  idcorresp, sitecorresp;
  workspace *  pr;
  fifo * qd, * qg;
}  ip;

typedef struct list_ips {
  struct list_ips *  s;
  ip *  p;
  int  id;
}  list_ips;
```

```
char  sch_version[]  = "@(#) 1/17/89(ADP-IRISA) --ECHIDNA Scheduler-- Version 2.8";

# include "echidnak.h"
# include "kernel.h"

# define maxenabled 127

/* structure to identify a transition function */
typedef struct id_trans {
  int  num_trans, num_any;
  }  id_trans;


static id_trans enabled_trans[maxenabled]; /* firable trans. per process */
static id_trans firable_trans[maxpr];        /* chosen firable trans. for
                                                each process, -1 if none */


extern workspace * tabproc[]; /* from runtime: hold actual processes for this
                                 site */
extern int  nbproc; /* actual number of processes in tabproc */

extern int actualsite; /* actual identifier of this site */

extern int  nbips;  /* actual number of ports for this site */

extern int (*transition[])(); /* from generated file */
extern int transany[];        /* from generated file */

void (* fire)();
void (* eval)();

/* macro to execute a transition function */
#define perform_trans(kind,ntrans,nany,c) (*transition[ntrans*2+kind])(nany,c)

/* kind is either evalue(=0) or tir(=1)                              */
/* ntrans and nany identify the actual transition (like id_trans)    */
/* p is the context of the process whose transition is performed     */



/* fire the initialization transition of process whose context is a, */
/* unless it is a leaf for another site */
address init_proc(a)
  address  a;
{
  workspace *  p;

  p = cont2ws(a);
  if ((p->site == -1) || (p->site == actualsite))
  (void)perform_trans(tir, p->firsttrans, 1, a);
  return a;
}


/*
** Here is the code to eval guards of transitions
*/


/* for each active process, eval its guards,
   choose at random one enabled guard, and put it in firable_trans[] */
/* Non-determinism is actually implemented */
void ND_eval_guards()
{
  register int  k;
  register int  j;
  register int  i;
  workspace *  p;
  int  prio, maxprio, nbfirable;

  for ( i = 0; i < nbproc ;i++) {
    nbfirable = 0;
    p = tabproc[i];
    if (p->active) {
      maxprio = 0;
      for (j = p->firsttrans + 1; j < (p->firsttrans + p->nbtrans) ; j++)
        for (k = 0;k < transany[j] ; k++) {
          prio = perform_trans(evalue, j, k, (address)(p->context));
          if ((prio > maxprio)) {
            maxprio = prio;
            nbfirable = 1;
            enabled_trans[0].num_trans = j;
            enabled_trans[0].num_any = k;
          }
          else
            if ((prio == maxprio)) {
              if ((nbfirable == maxenabled))
                ker_error(6);
              enabled_trans[nbfirable].num_trans = j;
              enabled_trans[nbfirable].num_any = k;
              nbfirable++;
            }
        }
    }
    if ((nbfirable == 0)) {
      firable_trans[i].num_trans = -1;
      inactive(i);
    }
    else {
      firable_trans[i] = enabled_trans[randomint(0, nbfirable-1)];
    }
  }
}

/* for each active process, eval its guards,
   and select first enabled guard, and put it in firable_trans[] */
/* Non-determinism is no more implemented, and priorities are meaningless */
void D_eval_guards()
{
  register int  k;
  register int  j;
  register int  i;
  workspace *  p;
  int nbfirable;

  for ( i = 0; i < nbproc ;i++) {
    p = tabproc[i];
    nbfirable = 0;
    if (p->active) {
      for (j = p->firsttrans + 1;
        ((nbfirable==0) && (j < (p->firsttrans + p->nbtrans))) ; j++)
        for (k = 0;k < transany[j] ; k++)
          if (perform_trans(evalue, j, k, (address)(p->context)) != -1){
            nbfirable=1;
            firable_trans[i].num_trans = j;
            firable_trans[i].num_any = k;
            break;
          }
    }
    if ((nbfirable == 0)) {
      firable_trans[i].num_trans = -1;
      inactive(i);
```

```
    }
  }
}

/*
** Here is the code to fire selected transitions
*/

/* for each active process, fire transition whose identity     */
/*   is in the array firable_trans[]: synchronous parallelism  */
void SYNC_fire_trans()
{
  register int i;
  for (i = 0;i < nbproc ;i++)
    if (firable_trans[i].num_trans != -1)
      {
/*        light(1, 0);*/
        (void)perform_trans(tir, firable_trans[i].num_trans,
          firable_trans[i].num_any, (address)(tabproc[i]->context));
/*        light(0, 0);*/
      }
}

/* for some random active process, fire transition whose identity */
/*   is in the array firable_trans[]: asynchronous parallelism    */
void ASYNC_fire_trans()
{
  register int i;
  for (i = 0;i < nbproc ;i++)
    if ((firable_trans[i].num_trans != -1) && (randomint(1,10)<7))
      {
/*        light(1, 0);*/
        (void)perform_trans(tir, firable_trans[i].num_trans,
          firable_trans[i].num_any, (address)(tabproc[i]->context));
/*        light(0, 0);*/
      }
}

/*
**
*/

/* run the scheduler during d seconds, or infinity if d=0*/
void start_scheduler(d)
  int d;
{
  while (( d == 0 ) || (Time() < d))
    {
      /* light(0, 1); */
      check_extern_msg();
      (*eval)();
      (*fire)();
    }
}


/*
** Standalone run of a unique process
*/

/* run the process during d seconds, or infinity if d=0*/
/* the first enabled guard is fired */
void D_standalone(d)
  int d;
{
  register int  k;
  register int  j;
  workspace * p;
  int nbfirable,trans1,transend;

  p = tabproc[0];
  trans1 = p->firsttrans + 1;
  transend = p->firsttrans + p->nbtrans;

  while (( d == 0 ) || (Time() < d))
    {
      check_extern_msg();
      /* light(0, 1); */
      if (p->active) {
        nbfirable=0;
        for (j = trans1; j < transend ; j++)
          for (k = 0;k < transany[j] ; k++)
            if (perform_trans(evalue, j, k, (address)(p->context)) != -1){
              nbfirable=1;
              (void)perform_trans(tir, j, k, (address)(p->context));
            }
        if (nbfirable==0) inactive(0);
      }
    }
}


extern void init();          /* from runtime */
extern void configuration(); /* from generated file */


/* Here is the list of global options that can be set on command-line */

int BUFFER = 1;      /* if set, deliver traces only at the end of the
                        execution. Use -u to reset BUFFER.
                     */

int QUIET = 0;       /* if set, prevent header and warning to be displayed
                        Use -q to set QUIET mode.
                      */

int TRACE = 0;       /* if set, trace each transition */
                        /* currently not implemented */

int DISPLAY_STAT = 0;/* print some stats at the end of execution */
                        /* currently not implemented */

int ASYNCHRONOUS = 0;/* Synchronous parallelism is simulated on a site
                        If set, the parallelism is fully asynchronous
                        Use -a to set asynchronous mode.
                     */

int DETERMINISM = 0; /* Non-determinism is implemented to choose a firable
                        transition. If DETERMINISM is set, the first
                        firable transition of a process is actually fired,
                        priorities are ignored.
                        Use -D to set DETERMINISM.
                     */


/* Here is the entry point of the program */
main(argc, argv)
int argc;
char *argv[];
{
```

```
  int duration  = 0; /* means infinity */
  int user_seed = 0; /* means no provided seed for the random generator */

  register int  carg;    /* current arg to scan */
  /* Parse the command line */
  for (carg = 1; carg < argc; ++carg) {
    if (argv[carg][0] == '-') {
      switch (argv[carg][1]) {
        case 'd':  /* duration specified */
          duration = atoi(&argv[carg][2]);
          break;
        case 's':  /* seed specified */
          user_seed = atoi(&argv[carg][2]);
          break;

        case 'a':  /* ASYNCHRONOUS option */
          ASYNCHRONOUS = 1;
          break;
        case 'D':  /* DETERMINISM option */
          DETERMINISM = 1;
          break;
        case 'q':  /* QUIET option */
          QUIET = 1;
          break;
        case 'u':  /* Unbuffer trace output */
          BUFFER = 0;
          break;
        default:  /* unknown switch */
          /* ignore this for now */
          break;
      }
    } /* unknown argument: ignore this for now */
  }

  init(user_seed);
  configuration(); /* to be modified, taking 2 parameters : argc, argv */
  if (!QUIET)
     (void)sprintf(cbuf,
       "%d processes and %d ports initialized.\n", nbproc, nbips),update();

  if ((nbproc==1) && (DETERMINISM)) D_standalone(duration);
  /* Standalone run for the only process on this site */
  else {
  /* start the scheduler choosen by command line option */

    if (ASYNCHRONOUS) fire  = ASYNC_fire_trans;
      else fire = SYNC_fire_trans;
    if (DETERMINISM) eval = D_eval_guards;
     else eval = ND_eval_guards;
    start_scheduler(duration);
  }
  terminate(0);
}
```

```
char ker_version[] = "@(#) 1/17/89(ADP-IRISA) --ECHIDNA Kernel--V2.13";
#include "echidnak.h"
#include "kernel.h"

/* variables of the kernel  */
int  actualsite;
int randseed;
list_ips *  lp_head;
int  unic_idip;
int  pdebugid, nbips;
extern int QUIET; /* command line option**/

workspace * tabproc[maxpr]; /* hold actual processes for this site */

int  nbproc; /* number of processes in tabproc */

/*
**  Functions from the system dependant file : sys_xxx.c
*/

extern void system_init();
extern void sys_send();
extern int sys_msg_there();
extern void sys_get();
extern int phys_site();
extern void light();
extern int noconnexity();


/*
**  Error Functions : global
*/

void ker_error(i)
  int  i;
{
  (void)sprintf(cbuf, "********** Error %10d\n", i),update();
  switch (i) {
    case 1:
    (void)sprintf(cbuf, " no active process \n"),update();
    break ;
    case 2:
    (void)sprintf(cbuf, " invalid command line parameters \n"),update();
    break ;
    case 3:
    (void)sprintf(cbuf, " bad random parameters \n"),update();
    break ;
    case 4:
    (void)sprintf(cbuf, " unspecified reception\n"),update();
    break ;
    case 5:
    (void)sprintf(cbuf, " echidna limit : too much processes \n"),update();
    break ;
    case 6:
    (void)sprintf(cbuf, " echidna limit : too much firable transitions\n"),update();
    break ;
    case 7:
    (void)sprintf(cbuf, " index out of range\n"),update();
    break ;
    case 8:
    (void)sprintf(cbuf, " memory overflow \n"),update();
    break ;
    case 9:
    (void)sprintf(cbuf, " impossible connexion for this network\n"),update();
    break ;
    case 10:
    (void)sprintf(cbuf, " set-space exhausted \n"),update();
    break ;
    case 11:
    (void)sprintf(cbuf, " missing case limb: line n\n"),update();
    break ;
    case 12:
    (void)sprintf(cbuf, " non existing port\n"),update();
    break ;
    case 13:
    (void)sprintf(cbuf, " non existing port\n"),update();
    (void)sprintf(cbuf, " Warning : there is a temporary restriction , modules must be
    break ;
    default:
    (void)sprintf(cbuf, " Bad error number: %d \n",i),update();
  }
  (void)sprintf(cbuf, "_____________________________________\n"),update();
  terminate(i);
}


void ker_warning(i)
  int  i;
{
  if (!QUIET) {
    (void)sprintf(cbuf, "********** Warning %10d\n", i),update();
    switch (i) {
      case 1:
      (void)sprintf(cbuf, " message sent on an unconnected port : message lost \n"),upd
      break ;
      case 2:
      (void)sprintf(cbuf, " multiple connections not allowed, previous connection lost\
      break ;
      case 3:
      (void)sprintf(cbuf, " attached port yet connected, previous connection lost\n"),u
      break ;
      case 5:
      (void)sprintf(cbuf, " no process at the other end of this port: message lost \n")
      break ;
      case 6:
      (void)sprintf(cbuf, " unspecified reception : message lost\n"),update();
      break ;
      case 7:
      (void)sprintf(cbuf, " unspecified external reception : message lost\n"),update();
      break ;
      default:
      (void)sprintf(cbuf, " unexpected Warning !!!\n"),update();
    }
  }
}

void print_version()
{
   if (!QUIET) (void)sprintf(cbuf, "%s\n", ker_version),update();
}



/*
**  Utilities
*/

void init(user_seed)
  int user_seed;
{
  system_init(&actualsite);
  if (user_seed==0) randseed = 15277+actualsite*13;
```

```
    else randseed = user_seed;
  lp_head = (list_ips *)NIL;
  unic_idip = 0;
  nbips = 0;
  pdebugid = 0;
  nbproc = 0;
  print_version();
}

void link_ip(p)
  ip *  p;
{
  list_ips *  nv;

  nv = (list_ips *)malloc((unsigned)(sizeof(*nv)));
  if (nv == (list_ips *)NIL)
    ker_error(8);
  nv->s = lp_head;
  nv->p = p;
  lp_head = nv;
  nbips++;
}

ip * find_ip(id)
  int  id;
{
  register list_ips *  cur = lp_head;

  while (cur != (list_ips *)NIL)
    if ( cur->p->identity == id )
      return (cur->p);  /* ip found in the list */
    else  cur = cur->s;
  return ( (ip *)NIL ); /* ip not found in the list */
}



/* Macro to return a uniform distributed random real number between 0 and 1 */
/* s is the seed, bb the multiplier. 'Maxint &' is used to mask the sign bit*/

#ifndef INT16
#define invmaxint 4.656612875e-10
#define bb 31415821
#else
#define invmaxint 3.0518509475e-5
#define bb 9821
#endif

#define uniform (double)(invmaxint*(randseed=(Maxint & ((randseed*bb)-1))))

/* return a random real number between a and b */
double randomrel(a, b)
  double  a, b;
{
  if ((a < 0) || (b < 0) || (a > b))  ker_error(3);
  return ( (a == b) ? a : (double)((b - a + 1) * uniform) + a );
}

/* return a  random integer between a and b */
int randomint(a, b)
  int  a, b;
{
  if ((a < 0) || (b < 0) || (a > b)) ker_error(3);
  return ( (a == b) ? a : (double)(b - a + 1) * uniform + a );
}
```

```
#undef bb
#undef invmaxint



/*
** Active & Inactive processes, for intented use bye the scheduler
*/

void inactive(y)
  int  y;
{
  tabproc[y]->active = False;
}


void active(y)
  int  y;
{
  tabproc[y]->active = True;
}



/*
** Actions on messages
*/


/* queue message m in the FIFO file of port p */
void deliver(p, m)
  ip *  p;
  address  m;
{
  fifo *  f;

  f = (fifo *)malloc((unsigned)(sizeof(*f)));
  if (f == (fifo *)NIL)
    ker_error(8);
  else {
    f->m = m;
    f->l = (fifo *)NIL;
    f->pt = p;
    if ((p->qg != (fifo *)NIL))
      p->qg->l = f;
    p->qg = f;
    if (p->qd == (fifo *)NIL)
      p->qd = p->qg;
  }
}

/* active the process owner of port p on message m */
void awake(p, m)
  ip *  p;
  address  m;
{
  deliver(p, m);
  active(p->pr->number);
}


/* build a new message whose generated part size is msgsize*/
address mk_msg(msgsize)
  int  msgsize;
```

```
{
  msg *  m;

  m = (msg *)(malloc(msgsize + smsg));
  if (m == (msg *)NIL)
    ker_error(8);
  m->msgsize = msgsize + smsg;
  return ((address)(m->textmsg));
}

/* test for a message available on port p, and open visibility on it */
address get_msg(p)
  ip *  p;
{
  if (p == (ip *)NIL) ker_error(12);
  return ( (p->qd == (fifo *)NIL) ? (address)NIL : p->qd->m );
}

/* delete (consume) the first message available on port p */
void rm_msg(p)
  register  ip *  p;
{
  fifo *  f;

  f = p->qd;
  if ( (p->qd=f->l) == (fifo *)NIL )
      p->qg = (fifo *)NIL;
    free (a2msg(f->m));
    free(f);
}

/* deliver message m to the correspondant of port p */
void send_msg(p, m)
  ip *  p;
  address  m;
{
  msg *  em;

  if (p->sitecorresp == -1 ) {
    ker_warning(1);
    free (a2msg(m));
  }
  else {
    if (p->sitecorresp == actualsite)
      if ((p->corresp->pr == (workspace *)NIL))
        ker_warning(5);
      else  awake(p->corresp, m);
    else {
      em = a2msg(m) ;
      em->idipdest = p->idcorresp;
      sys_send(p->sitecorresp, em->msgsize, em);
      free (em); /* doesn't exist anymore on this site */
    }
  }
}

/* check for incomming external messages, and deliver them to their
   destination port */
void check_extern_msg()
{
  msg *  er;
  int  s;
  ip *  p;

  while (sys_msg_there(&s)) {
    if ( (er = (msg *)(malloc(s)) ) == (msg *)NIL )
      ker_error(8);
    sys_get(er, s);
    if ( (p = find_ip(er->idipdest)) == (ip *)NIL )
      ker_warning(7);
    else {
      awake(p, (address)(er->textmsg));
    }
  }
}



/*
** Fonctions for interactions points
*/


/* create a new port */
ip * mk_ip(site, pr)
  int  site;
  workspace *  pr;
{
  register ip *  p;

  if ( (p = (ip *)malloc((unsigned)(sizeof(*p)))  ) == (ip *)NIL )
    ker_error(8);
  p->identity = ++unic_idip; /* get next unique idendity */
  p->site = site;
  p->corresp = (ip *)NIL;
  p->idcorresp = 0;
  p->sitecorresp = -1;
  p->pr = pr;
  p->qd = (fifo *)NIL;
  p->qg = (fifo *)NIL;
  return p;
}

/* delete an unitialized port (whose not on this site) */
void del_ip(p)
  ip *  p;
{
  free(p);
}


/* connect ports p1 and p2 */
void connect_ip(p1, p2)
  ip **p1, **p2;
{
  if (((*p1) == (ip *)NIL) || ((*p2) == (ip *)NIL))
    ker_error(12);
  if (((*(*p1)).corresp != (ip *)NIL) || ((*(*p2)).corresp != (ip *)NIL))
    ker_warning(2);
  (*(*p1)).corresp = (*p2);
  (*(*p2)).corresp = (*p1);
  (*(*p1)).sitecorresp = (*(*p2)).site;
  (*(*p2)).sitecorresp = (*(*p1)).site;
  (*(*p1)).idcorresp = (*(*p2)).identity;
  (*(*p2)).idcorresp = (*(*p1)).identity;
  if (noconnexity( (*(*p1)).site, (*(*p2)).site ))
      ker_error(9);
}

/* attach port p1 to port p2 */
void attach_ip(p1, p2)
```

```
  ip **p1, **p2;
{
  if (((*p1) == (ip *)NIL) || ((*p2) == (ip *)NIL))
    ker_error(12);
  if ((*(*p2)).corresp != (ip *)NIL)
    ker_warning(3);
  if ((*(*p1)).corresp != (ip *)NIL) {
    (*(*(*p1)).corresp).corresp = (*p2);
    (*(*p2)).corresp = (*(*p1)).corresp;
    (*(*(*p1)).corresp).sitecorresp = (*(*p2)).site;
    (*(*p2)).sitecorresp = (*(*(*p1)).corresp).site;
    (*(*(*p1)).corresp).idcorresp = (*(*p2)).identity;
    (*(*p2)).idcorresp = (*(*(*p1)).corresp).identity;
    if (noconnexity((*(*p2)).site,(*(*p2)).sitecorresp))
      ker_error(9);
  }
  free((*p1));
  (*p1) = (*p2);
}



/*
**  Functions for the processes
*/


/* perform no-operation, when there is a spontaneous transition */
int nop(n, p)
  int  n;
  address  p;
{
  return 0;
}


/* create a new process (module+body) */
int mk_proc(wp, site, tpmobile, trans1, ntrans, nsblocs, nbports)
  address * wp;
  int  site, tpmobile, trans1, ntrans, nsblocs, nbports;
{
  register int  i;
  register workspace *  p;
  register int here;
  /* First, alloc space for process variables iff it is to be run here */
  if (here = ((site == -1) || (phys_site(site) == actualsite)))
    p = (workspace *)malloc( (unsigned)(sws+nbports*sizeof(address)+tpmobile) );
  else p = (workspace *)malloc( (unsigned)(sws+nbports*sizeof(address)) );
  if (p == (workspace *)NIL)
    ker_error(8);
  if ((site >= 0))
    p->site = phys_site(site);
  else
    p->site = -1;
  p->nbips = nbports;
  p->nbsubmod = nsblocs;
  p->firsttrans = trans1-1;
  p->nbtrans = ntrans;
  *wp = (address)(p->context);/* a pointer on the user defined context */
  /* ips initialisation: they are stored as the first field of 'context' */
  for (i=0;i<nbports;i++) {
    p->context[i] = (address)(mk_ip(p->site, p));
  }
  return (here); /* If the process is to be initialized on this node */
}
```

```
/* setup the process in the global array tabproc  */
void initprocessus(x)
  workspace *  x;
{
  if (nbproc == maxpr)
    ker_error(5);
  else {
    x->number = nbproc++;
    x->active = True;
    tabproc[x->number] = x;
  }
}


/* extract from the tree the processes for this node,
   and delete the others */
/* warning : this is a recursive procedure */
void setup_proc(a)
  address  a;
{
  workspace *  p;
  register int  i;

  p = cont2ws(a);
  if ((p->site == actualsite)) {
    /* setup the proc. in the global array ...  */
    initprocessus(p);
    /* setup its ips in the global list for this site */
    for (i = 0; i < p->nbips ; i++) {
      link_ip((ip *)(p->context[i]));
    }
  }
  else {
    /* this is not a process for this site */
    if ((p->nbsubmod == 0)) {
        /* delete its ips iff proc. is a leave (no sub-modules) */
        for (i = 0; i < p->nbips ; i++) {
          del_ip((ip *)(p->context[i]));
        }
    }
    else {
      /* call recursivly setup_proc for each sub-module */
      for (i = p->nbips; i < (p->nbips+p->nbsubmod) ; i++) {
        setup_proc(p->context[i]);
      }
    }
    free(p); /* release the current process */
  }
}



/*
** Now there is the Pascal runtime, above of the C-one
** mainly about set manipulation...
*/



setptr  tmpset;

setptr
u_Union(p1, p2)
       register setptr p1, p2;
```

```
{
        register int            i, j, k;
        register setptr         sp = newset(),
                                p3 = sp;

        j = *p1;
        *p3 = j;
        if (j > *p2)
                j = *p2;
        else
                *p3 = *p2;
        k = *p1 - *p2;
        p1++, p2++, p3++;
        for (i = 0; i < j; i++)
                *p3++ = (*p1++ | *p2++);
        while (k > 0) {
                *p3++ = *p1++;
                k--;
        }
        while (k < 0) {
                *p3++ = *p2++;
                k++;
        }
        return (saveset(sp));
}

setptr
diff(p1, p2)
        register setptr p1, p2;
{
        register int            i, j, k;
        register setptr         sp = newset(),
                                p3 = sp;

        j = *p1;
        *p3 = j;
        if (j > *p2)
                j = *p2;
        k = *p1 - *p2;
        p1++, p2++, p3++;
        for (i = 0; i < j; i++)
                *p3++ = (*p1++ & ~ (*p2++));
        while (k > 0) {
                *p3++ = *p1++;
                k--;
        }
        return (saveset(sp));
}

setptr
inter(p1, p2)
        register setptr p1, p2;
{
        register int            i, j;
        register setptr         sp = newset(),
                                p3 = sp;

        if ((j = *p1) > *p2)
                j = *p2;
        *p3 = j;
        p1++, p2++, p3++;
        for (i = 0; i < j; i++)
                *p3++ = (*p1++ & *p2++);
        return (saveset(sp));
}
```



```
Boolean
member(m, sp)
        register unsigned int   m;
        register setptr sp;
{
        register unsigned int   i = m / (setbits+1) + 1;

        if ((i <= *sp) && (sp[i] & (1 << (m % (setbits+1)))))
                return (True);
        return (False);
}

Boolean
eq(p1, p2)
        register setptr p1, p2;
{
        register int    i, j;

        i = *p1++;
        j = *p2++;
        while (i != 0 && j != 0) {
                if (*p1++ != *p2++)
                        return (False);
                i--, j--;
        }
        while (i != 0) {
                if (*p1++ != 0)
                        return (False);
                i--;
        }
        while (j != 0) {
                if (*p2++ != 0)
                        return (False);
                j--;
        }
        return (True);
}

Boolean
ne(p1, p2)
        register setptr p1, p2;
{
        return (!eq(p1, p2));}

Boolean
le(p1, p2)
        register setptr p1, p2;
{
        register int    i, j;

        i = *p1++;
        j = *p2++;
        while (i != 0 && j != 0) {
                if ((*p1++ & ~ *p2++) != 0)
                        return (False);
                i--, j--;
        }
        while (i != 0) {
                if (*p1++ != 0)
                        return (False);
                i--;
        }
        return (True);
}

Boolean
```

```
ge(p1, p2)
        register setptr p1, p2;
{
        register int    i, j;

        i = *p1++;
        j = *p2++;
        while (i != 0 && j != 0) {
                if ((*p2++ & ~ *p1++) != 0)
                        return (False);
                i--, j--;
        }
        while (j != 0) {
                if (*p2++ != 0)
                        return (False);
                j--;
        }
        return (True);
}

setptr
mksubr(lo, hi, sp)
        register unsigned int   lo, hi;
        register setptr sp;
{
        register int    i, k;

        if (hi < lo)
                return (sp);
        i = hi / (setbits+1) + 1;
        for (k = *sp + 1; k <= i; k++)
                sp[k] = 0;
        if (*sp < i)
                *sp = i;
        for (k = lo; k <= hi; k++)
                sp[k / (setbits+1) + 1] |= (1 << (k % (setbits+1)));
        return (sp);
}

setptr
insmem(m, sp)
        register unsigned int   m;
        register setptr sp;
{
        register int    i,
                                j = m / (setbits+1) + 1;

        if (*sp < j)
                for (i = *sp + 1, *sp = j; i <= *sp; i++)
                        sp[i] = 0;
        sp[j] |= (1 << (m % (setbits+1)));
        return (sp);
}

# define SETSPACE 1024
setptr
currset(n,sp)
        int     n;
        setptr  sp;
{
        static setword  space[SETSPACE];
        static setptr   top = space;

        switch (n) {
          case 0:
                top = space;
                return (0);
          case 1:
                if (&space[SETSPACE] - top <= 15) {
                        ker_error(10);
                }
                *top = 0;
                return (top);
          case 2:
                if (top <= &sp[*sp])
                        top = &sp[*sp + 1];
                return (sp);
        }
        /* NOTREACHED */
}

void
setncpy(S1, S2, N)
        register setptr S1, S2;
        register unsigned int   N;
{
        register unsigned int   m;

        N /= sizeof(setword);
        *S1++ = --N;
        m = *S2++;
        while (m != 0 && N != 0) {
                *S1++ = *S2++;
                --N;
                --m;
        }
        while (N-- != 0)
                *S1++ = 0;
}

void caseerror(n)
        int     n;
{
        ker_error(11);/*n is the line number, second parameter?*/
        }


int Max(m, n)
        int     m, n;
{
        if (m > n)
                return (m);
        return (n);
}


int trunc(f)
double  f;
{
        return f;
}


int round(f)
double  f;
{
        return trunc((double)(0.5+f));
}
```

```
#define version "@(#) 1/4/89(ADP-IRISA) --ECHIDNA System Kernel-- Version 2.5"
#include <cube.h> /* a CHANGER */

/* primittives for I/O */

#define NIL 0
extern int printf();
static char sbuf[1023];
char *cbuf;

extern int BUFFER; /* command line option */

typedef struct list_buf{
        struct list_buf * nextone;
        char * string;
}list_buf;
static list_buf *first_buf = (list_buf *)NIL;
static list_buf *last_buf  = (list_buf *)NIL;

void terminate();
void update()
{
  if (BUFFER) {
        register list_buf * newone;
        if ( (newone=(list_buf *)malloc(sizeof(list_buf))) == (list_buf *)NIL ){
          (void)printf("*** memory overflow during bufferised output.\n");
          terminate(8);
        }
        if ( (newone->string=(char *)malloc(strlen(cbuf)+1)) == (char *)NIL ){
          (void)printf("*** memory overflow during bufferised output.\n");
          terminate(8);
        }
        (void)strcpy(newone->string,cbuf);
        newone->nextone=(list_buf *)NIL;
        if (first_buf == (list_buf *)NIL){
          first_buf=newone;
          last_buf=newone;
        }
        else {
          last_buf->nextone=newone;
          last_buf=newone;
        }
  } else (void)printf("%s",cbuf);
}
void download_trace()
{
  if (BUFFER) {
        register list_buf * cur=first_buf;
        while (cur != (list_buf *)NIL) {
          (void)printf("%s",cur->string);
          {
            /* wait a little bit to avoid system buffer saturation */
            int i;
            for (i=0;i<10000;i++){} ;
          }
          cur=cur->nextone;
        }
  }
}


void terminate(cond)
int cond;
{
  /* the i/o buffer has to be downloaded */
  download_trace();
  exit(cond);
}



void light (v,r)
  int v,r;
  {
        led(v>0);
}

/* primittives for TIME */
unsigned long mclock() ;
static unsigned long baseclock;
double Time()
  {
        return((double)(mclock()-baseclock)/1000);
}


/* initialisation routine for the system dependant module */
void system_init(me)
        int     *me;
  {
        *me=mynode();
        cbuf=sbuf;
        baseclock=mclock();
}

/* embeding routine */
int phys_site(s)
        int     s;
{
        return (s % numnodes());
}


/* communications routines */
void sys_send(n,t,em)
  int n,t; char *em;
  {
        csend(0,em,t,n,0);
}


void sys_get(em,s)
  char *em; int s;
  {
        crecv(0,em,s);
}

int sys_msg_there(s)
  int *s;
  {
        int ok,node,pid;
        if (ok=iprobe(0)) *s=infocount();
        return ok;
}

#ifndef TOPOHYP
int noconnexity(a,b)
  int a,b;
{
        return 0;
}
```

```
#else
int neighbour(a,b)
  int a,b;
{
        register int    d=0;
        register int    i;
        for (i = 0;i<nodedim() ; i++) {
          if (((a % 2) != (b % 2))) d++;
            a = a >> 1;
            b = b >> 1;
        }
        return (d == 1);
}
int noconnexity(a,b)
  int a,b;
{
        return ( (a != -1) && (b != -1) && (a != b) && (!neighbour(a,b)) );
}
#endif
```

Liste des dernières publications internes

PI 447 **VISION 3D : UNE NOUVELLE METHODE DE LA TRIANGULATION POUR LA VISION MONOCULAIRE OU POLYNOCULAIRE DANS UNE SEQUENCE D'IMAGES**
Ming XIE, Patrick RIVES
48 Pages, Janvier 1989.

PI 448 **DYNAMIC VISION : DOES 3D SCENE PERCEPTION NECESSARILY NEED TWO CAMERAS OR JUST ONE ?**
Ming XIE
38 Pages, Janvier 1989.

PI 449 **CONVOLUTION SYSTOLIQUE DE FONCTIONS ARITHMETIQUES**
Patrice QUINTON, Yves ROBERT
30 Pages, Janvier 1989.

PI 450 **MISE EN OEUVRE D'ALGORITHMES NUMERIQUES SUR UN HYPERCUBE**
Brigitte VITAL
28 Pages, Janvier 1989.

PI 451 **LANCER DE RAYON SUR DES ARCHITECTURES PARALLELES : UNE ETUDE DE PERFORMANCE**
Thierry PRIOL, Kadi BOUATOUCH
20 Pages, Janvier 1989.

PI 452 **LANCER DE RAYON : APPROCHES PARALLELES**
Didier BADOUEL, François BODIN, Thierry PRIOL
16 Pages, Janvier 1989.

PI 453 **UN COMPILATEUR ESTELLE MULTI-PROCESSEURS POUR L'EXPERIMENTATION D'ALGORITHMES DISTRIBUES SUR MACHINES PARALLELES**
Jean-Marc JEZEQUEL, Claude JARD
54 Pages, Janvier 1989.

Imprimé en France
par
l'Institut National de Recherche en Informatique et en Automatique