ESTELLE:un langage ISO pour les algorithmes distribués et les protocoles

Jean-Pierre Courtiat, Piotr Dembinski, Roland Groz, Claude Jard

INRIA

UNITÉ DE RECHERCHE
INRIA-RENNES

Institut National
de Recherche
en Informatique
et en Automatique

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

Rapports de Recherche

N° 595

ESTELLE: UN LANGAGE ISO POUR LES ALGORITHMES DISTRIBUÉS ET LES PROTOCOLES

Jean-Pierre COURTIAT
Piotr DEMBINSKI
Roland GROZ
Claude JARD

Décembre 1986

IRISA

INSTITUT DE RECHERCHE EN INFORMATIQUE
ET SYSTÈMES ALÉATOIRES

ESTELLE: un langage ISO pour les algorithmes distribués et les protocoles

ESTELLE: an ISO language for distributed algorithms and protocols

Jean-Pierre COURTIAT*, Piotr DEMBINSKI**, Roland GROZ***

Claude JARD****,***

Publication Interne n° 326

24 pages

Décembre 1986

Résumé: Estelle est un langage de description formelle des protocoles défini à l'ISO+. Cet article est destiné à présenter ce nouveau langage à la communauté informatique. L'article est découpé en deux parties. La première présente le langage. L'allure générale d'Estelle est d'abord esquissée en donnant les concepts essentiels. Une présentation précise suit avec des exemples complets de descriptions en annexe. La deuxième partie parle des outils associés au langage. Différents outils existant ou en cours de développement sont présentés ainsi que leurs perspectives industrielles.

Abstract: Estelle is a formal description language for protocols and defined at ISO. This paper is aimed to introduce this new language to the community of computer scientists. It is structured in two parts. The first one presents the language. General characteristics of Estelle are sketched through its basis concepts. A detailed presentation follows with full formal descriptions . The second part is devoted to some Estelle tools. Different tools, available or under development, are presented with their industrial prospects.

* LAAS CNRS 7, av. du colonel Roche 31077 Toulouse Cedex. Tél. 61336200
** ADI, projet RHIN, Tour Fiat Cedex 16 92084 Paris La Défense. Tél. (1) 47964321
*** CNET Lannion EVP, route de Trégastel 22301 Lannion Cedex. Tél. 96053790
**** IRISA CNRS, Campus de Beaulieu 35042 Rennes Cedex. Tél. 99362000 p425

+ Organisation Internationale de Normalisation.

Cet article est le résultat d'un travail collectif suscité et animé par Claude Jard.

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)

INTRODUCTION

1. Les motivations

Le développement du logiciel des systèmes distribués est une tâche largement aussi difficile que celui du logiciel classique en général. Les difficultés tiennent principalement en deux points. Premièrement, on doit assurer la compatibilité entre composants logiciels souvent hétérogènes et développés par différents groupes (ou organismes). Deuxièmement, la compréhension des comportements parallèles est encore un exercice malaisé. Le logiciel distribué est généralement structuré en couches. Pour un protocole donné, on considère deux niveaux de description : une description du service qui spécifie le comportement global du système du niveau considéré, et une description du protocole donné en termes d'entités coopérantes, devant réaliser le service.

Ces points montrent l'importance d'une description précise des différents composants du système. Comme pour le logiciel en général, l'emploi du langage naturel (français, anglais ...) conduit à des descriptions longues et souvent ambiguës. Ce qui ne permet pas la vérification de bon fonctionnement et peut induire des mises en oeuvre incompatibles.

Pour résumer, la difficulté de maîtrise des logiciels distribués, leur spécificité et la nécessité de formaliser justifient le développement de techniques de description formelle pour les algorithmes distribués et les protocoles. Les spécifications produites servent de référence pour différentes activités le long du cycle de production du logiciel, comme la validation des algorithmes, la génération d'une mise en oeuvre et le test de celle-ci.

2. Historique

L'ISO et le CCITT (Comité Consultatif International Télégraphique et Téléphonique) ont étudié depuis quelques années des techniques de description formelle pour la spécification de protocoles. Ce travail était motivé par le fait que les ambiguïtés dans la description des normes conduisent souvent à des mises en oeuvre incompatibles, un résultat en contradiction avec la normalisation.

L'ISO a conçu deux types de techniques. Une est appelée Lotos; elle est fondée sur la description de l'ordonnancement temporel des interactions (utilisant un modèle proche de CCS [Milner 80]). L'autre est dénommée Estelle (pour *Extended State Transition Language*); elle est fondée sur des automates communicants, étendus avec le langage de programmation Pascal. Une première proposition de norme pour Estelle et Lotos a été produite en 1985. Cette proposition, après quelques amendements, entre dans le processus de normalisation [ISO86a,86b] qui devrait aboutir dans quelques années. Ces langages sont destinés à être utilisés pour la description formelle des protocoles normalisés.

3. L'effort en France sur Estelle

La contribution francaise à Estelle a été particulièrement importante. Les travaux sur PDIL (*Protocol Development and Implementation Language*) dans le projet RHIN (Agence de l'Informatique) [Ansart 82] et LC/1 au LAAS (Laboratoire d'Automatique et d'Analyse des Systèmes) [Ayache 83] ont préfigurés Estelle. Le CNET, à l'occasion du projet Véda, a oeuvré au CCITT pour la clarification du langage et le rapprochement avec l'ISO. Ces trois équipes émettaient principalement ces dernières années, les contributions françaises. Depuis, beaucoup d'autres groupes de recherche se sont intéressés à Estelle qui joue un rôle fédérateur dans des activités comme la génération de mises en oeuvre, la détermination de séquences de test, la vérification et la simulation des protocoles.

4. Plan de l'article

L'article est découpé en deux parties. La première présente le langage en commençant par les concepts essentiels. Une présentation détaillée suit avec des exemples complets de descriptions formelles en annexe. La deuxième partie est consacrée à des outils Estelle, existants ou en voie de développement, ainsi qu'à leurs perspectives industrielles.

PARTIE I : Le langage Estelle

Le langage Estelle peut être rapidement décrit comme fondé sur un modèle d'automate étendu avec le langage de programmation Pascal. Il inclue tout le langage Pascal, mais l'encapsule dans des éléments qui en font un véritable langage pour l'expression de comportements parallèles. Trois caractéristiques principales sont à noter:

- une spécification est composée de plusieurs modules. Estelle permet de bien spécifier les interfaces de chacun d'entre eux.
- la description du comportement de chaque module est assez fine et précise, afin que le comportement de la spécification ne soit pas ambigü, ou que ses ambiguïtés (non-déterminisme) soient explicites.
- la notion de typage (définition de types, puis création d'exemplaires) du langage Pascal est étendue aux objets parallèles (comme les modules, canaux ...), dont certains sont même paramètrables.

Estelle apporte donc un soin particulier à décrire les interfaces (définition de ce qu'on appelle des canaux, des modules, des interactions....) et toutes les finesses d'un comportement non-déterministe (partie *body*, avec des transitions comportant des gardes variées ...).

I. Les concepts de base.

Ce paragraphe présente les concepts de base spécifiques du langage Estelle (expression des comportements parallèles).

I.1. La notion de module

Le comportement d'un module (tel qu'il est perçu à travers ses points d'interaction) est donné par un modèle à transitions d'état (automate). Un modèle à transitions d'état est défini par un ensemble d'entrées, de sorties, d'états et de transitions. Néanmoins, ce modèle est étendu pour lui donner la puissance et la concision d'un langage de programmation. L'état interne est prolongé par un vecteur de variables Pascal. Dans le principe, l'état principal (celui de l'automate d'état fini sous-jacent) détermine les phases du protocole décrit, c'est à dire l'aspect contrôle, alors que l'aspect données sera représenté dans des variables Pascal. Pour l'aspect contrôle, le langage Estelle définit une syntaxe originale. Pour l'aspect données, Estelle reprend intégralement les constructions Pascal.

I.2. L'interconnexion

L'architecture du système est définie par un ensemble de modules communicants. Ces modules s'échangent des messages (interactions) via des points d'interaction (ou ports). Chacun de ces points est typé par un canal. La spécification d'un canal contient une énumération de toutes les interactions possibles sur les ports se référant à ce type. Pour chaque type d'interaction, on donne une liste de paramètres transportés par l'interaction et leurs types. Les concepts de canaux et points d'interaction servent à partitionner les échanges de signaux dans le système.

A chaque point d'interaction est associé une file dans laquelle sont placées les interactions reçues. Toutes les files sont non bornées et de discipline FIFO. Deux modules peuvent être interconnectés en reliant deux points d'interaction. Seules les interactions spécifiées par le canal associé au points d'interaction sont autorisées. Une liaison est a priori bidirectionnelle.

I.3. La structuration

Une spécification est constituée de plusieurs modules. Chaque module est composé d'une en-tête et d'un corps. Le corps d'un module contient une partie d'initialisation et un ensemble de transitions. Il peut se suffire à lui même ou être structuré en sous modules. Les exemplaires de sous modules sont créés uniquement à l'intérieur du module immédiatement englobant dans les parties initialisation ou transitions. On parle de configuration statique lorsque la création n'a lieu que dans la partie initialisation. A un niveau donné de la hiérarchie statique, deux types de synchronisation entre modules peuvent être considérés, couvrant un nombre varié de problèmes. Il s'agit du mode asynchrone (la synchronisation ne se fait que par échange d'interactions) et du mode synchrone (les transitions tirables au même moment sont tirées simultanément). Le partage de variables est autorisé entre un module parent et ses enfants (dans la hiérarchie). Pour cette raison, une transition d'un module père ne peut s'exécuter simultanément avec une de celles de ses fils.

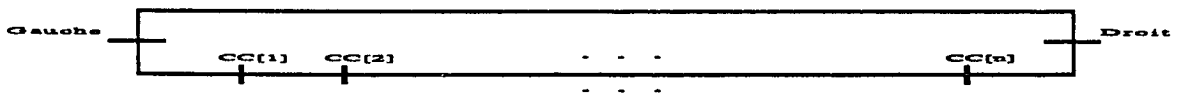
Enfin, un module peut créer dynamiquement (dans les actions des transitions) des sous modules fils, puis les détruire. La partie initialisation d'un module est activée lors de sa création.

II. La description des interfaces.

II.1. Module

Dans la découpe fonctionnelle de la spécification d'un système distribué, il faut d'abord avoir isolé des modules. Ces entités peuvent aussi bien correspondre à un site qu'à un groupe de sites ou à une fonction à l'intérieur d'un site. L'important est qu'il s'agisse d'entités aux interfaces bien définies. On peut alors leur associer une déclaration de modules Estelle.

Par exemple, pour l'algorithme de Dijkstra (voir annexe, exemple 1), on peut associer à chaque site le module suivant:



La déclaration Estelle correspondante est :

```
module  Site systemprocess (identite : integer) ;
    ip    Gauche : anneau (arriere) ; Droit : anneau (avant) ;
          CC : array [1..nb_sites] of canal_calcul (role) ; end ;
```

Ceci définit un type de boîte Site à n+2 points d'interaction. Par exemple, par CC[j], ce Site communiquera avec le site d'identité j (la façon de connecter deux sites sera vue plus loin). A ces liaisons CC qui établiront un réseau totalement maillé, par lequel se fera l'activité propre à l'algorithme dont on cherche à détecter la terminaison, se superposera un anneau virtuel établi par les liens Gauche et Droit. NB: il se peut que les messages circulant sur l'anneau empruntent la même ligne (physique ou autre) que ceux passant par les CC. On a fait ici une découpe fonctionnelle de ces lignes en points d'interaction correspondant à des activités différentes.

La syntaxe complète d'une déclaration de module est la suivante:

module <id> [classe] <liste-de-parametres-de-module>;
[*ip* <liste-de-points-d'interaction>]
[*export* <suite-de-déclarations-de-variable>] *end* ;

Les paramètres de module sont des paramètres formels Pascal de type valeur. L'attribut classe est constitué de l'un des mots-clés *systemprocess*, *systemactivity*, *process* ou *activity* dont les rôles sont expliqués au paragraphe V. Les points d'interaction sont déclarés dans la partie *ip*, et leur syntaxe est la suivante:

<liste-id> : [*array* <types-d'indices> *of*] <id-canal> (<id-role>)
[<discipline-de-file>]

<id-canal> et <id-role> sont des identificateurs apparus dans une déclaration de type de canal (cf ci-dessous). La discipline de file permet d'indiquer si la file où s'accumulent les interactions devant être traitée est propre à ce point d'interaction, ou si elle est partagée (ce qu'on indique par les mots-clés optionnels *individual queue* ou *common queue*). Enfin, par la clause *export*, on déclare que le module contient des variables (avec leur type) qui pourront être accédées par d'autres modules. Ce partage de variable est en fait restreint: seul un module "père" peut accéder aux variables de ses "fils" (cf infra la sémantique du parallélisme pour les notions de père et fils).

II.2. Typage des interfaces

En Estelle, chaque point d'interaction est typé. Ce typage définit les types d'interaction qui pourront passer par ce point. Par exemple, les points d'interaction entre deux couches du modèle OSI verront passer des interactions qui devront faire partie de la liste des primitives de service offertes à ce niveau. Dans ce cas, on voit bien que les primitives émises par la couche haute (requêtes en général) et celles émises par la couche basse sont différentes. Estelle permet de distinguer deux rôles, chaque extrémité jouant un rôle (en Estelle, toutes les interactions se font entre deux partenaires). Les déclarations correspondant à l'exemple 1 illustrent la syntaxe.

```
channel anneau (avant, arriere) ;                    (* roles avant et arriere          *)
    by avant : jeton (couleur: type_couleur);
    by arriere : ;                          (* ce role est vide, il ne peut rien emettre  *)
                              (* ceci traduit le caractere unidirectionnel de l'anneau    *)
end ;
channel canal_calcul (role, bidon) ;
    by role, bidon: message ;                  (* on aurait pu ecrire by role : ...      *)
end ;                                            (* puisque bidon ne sert à rien          *)
```

NB On a supposé une déclaration préalable : type_couleur = (blanc,noir)

III. La description du comportement

La définition des types d'interfaces d'un module ne dit pas grand chose sur son comportement local. C'est la déclaration de corps de module (*body*) qui va définir les contraintes d'ordonnancement qui définiront l'algorithme distribué. Notons que pour les mêmes interfaces, donc pour le même type de module, on peut avoir plusieurs comportements possibles, donc plusieurs corps possibles en Estelle. Ceci explique qu'on pourra définir plusieurs *body* pour le même type de module, et associer dans la configuration du système distribué différents corps à différents exemplaires du même type de module.

III.1. Structure générale

Syntaxiquement, ce corps de module ressemble à un programme Pascal pour toute la partie des déclarations: constantes, types (suivis éventuellement de déclarations de canaux et modules), variables (suivies de la déclaration de l'état et éventuellement des exemplaires de modules), procédures et fonctions; la différence essentielle est qu'ensuite, au lieu d'avoir un programme séquentiel, on a un automate qui peut être non-déterministe.

On décrit cet automate par une suite de transitions. Chaque transition est constituée de deux parties: la condition préalable et l'action associée. La condition préalable définit à partir de quel état, ou sur l'arrivée de quelle interaction la transition peut être tirée. L'action associée consiste en un changement d'état, et d'éventuelles émissions d'interactions. L'état de l'automate comprend deux composantes:

- d'une part une variable distinguée, de type énuméré, appelée *state;* si l'automate se limite à cette variable, c'est un automate d'état fini;
- d'autre part les valeurs des variables Pascal, qu'on a pu déclarer dans la partie déclaration.

Ainsi, Estelle définit des automates dont la structure de contrôle est celle d'automates d'état fini, mais qu'on étend (d'où le nom du langage) par des variables et un programme Pascal.

III.2. Les principaux constituants des transitions

Voici un exemple de transition:

```
from passif                                   (* condition: etre dans l'etat passif   *)
    when Gauche.jeton                         (* condition: reception d'un jeton      *)
        provided couleur=noir                 (* condition sur le contenu du jeton    *)
            to passif                         (* action: changement d'etat            *)
                begin output Droit.jeton(noir) ;   (* action: sortie                  *)
                ma_couleur:= blanc end ;      (* action: variable modifiee            *)
```

Toutes les conditions préalables doivent être réalisées pour que la transition puisse être tirée. On a donc une conjonction implicite sur les conditions. L'ordre dans lequel celles-ci sont écrites n'importe pas: on aurait pu commencer par *when*, puis *from*, puis *provided*. Noter cependant que la clause *provided* doit suivre la clause *when*, puisqu'elle fait référence au contenu du jeton: la clause *when* ouvre la visibilité des paramètres de l'interaction (on a un *with* Pascal implicite sur ces paramètres).

On décrit donc l'automate par une suite de transitions, c'est à dire de couples <condition/action> où la condition est une suite de clauses et l'action un énoncé composé Pascal, et éventuellement une clause *to* (celle ci-dessus n'était pas nécessaire puisqu'on ne changeait pas d'état de contrôle). L'exécution de cet automate consiste à évaluer toutes les conditions préalables, puis à tirer une transition dont toutes les clauses sont satisfaites, et à recommencer ainsi en un cycle infini. La sémantique d'exécution est décrite plus précisément au paragraphe V.

Nous avons vu que la condition préalable d'une transition peut contenir trois types de clauses: *from*, *when* et *provided*. Si *when* est absent, la transition n'est conditionnée que par l'état de l'automate, et peut donc être tirée spontanément, en l'absence de tout événement extérieur, dès que l'état voulu (défini par *from* et *provided*) est atteint. Ces transitions spontanées peuvent être assorties d'une clause *delay* (d1,d2) où d1 et d2 sont deux expressions entières positives, avec $d2 \geq d1$. Dans ce cas, la transition ne sera tirable qu'au bout d'un temps t, avec $d2 \geq t \geq d1$, à partir du moment où on a atteint l'état voulu, et à condition qu'on soit resté dans cet état pendant tout ce temps t. Si on est resté pendant un temps d2 dans l'état voulu, la transition devient impérative : elle doit être tirée immédiatement. Cette sémantique

intuitive correspond à celle des réseaux de Petri temporels de Merlin. Elle satisfait aux contraintes précises décrites dans le document [ISO86a] (qui prend de plus en compte le contexte dans lequel se trouve le module).

III.3. Commodités syntaxiques

Estelle permet de faciliter l'écriture des transitions par plusieurs moyens. Il y a d'abord une clause qui abrège l'écriture des transitions: la clause *any* . Par exemple :

```
any i :1..nb_sites do
    when CC[i].message ...
```

évite d'écrire nb_sites transitions de réception d'un message, une pour chaque point d'interaction CC.

L'autre moyen fourni par Estelle pour éviter de répéter des clauses est de les mettre en facteur commun. Ainsi, on pourra écrire :

```
trans    from E1  provided C1a to E5
                      when I.x begin end;
                      when I.y begin end;
                  provided C1b to E4
                      when I.x begin end;
         from E2      when I.z begin end;
trans    when K.a     from E4 to E1 begin end;
                      from E5 to E4 begin end;
         when K.b
                      from E4,E5 to E2 begin end;
```

Le mot-clé *trans* introduit une suite de transitions. L'indentation utilisée sur l'exemple montre la portée des différentes clauses. A chaque bloc *begin end;* correspond une transition. Ainsi, les 3 premières transitions sont conditionnées par *from E1*. Les deux premières le sont par *provided C1a*. Dans la première suite de transitions, on a commencé par factoriser sur les *from*: celà correspond à une description de l'automate état par état. La deuxième suite de transitions correspond plutôt à l'approche : "Quand je reçois telle interaction, que fais-je ?". Entre les deux suites de transition, il a fallu remettre le mot-clé *trans* pour éviter que le *when K.a* se trouve implicitement sous la portée du *from E2* qui précède. Pour plus de détails, on se reportera à la définition complète de la syntaxe d'Estelle.

III.4. L'initialisation

Puisque le corps du module Estelle n'est pas une séquence d'instructions ayant un point de départ, il convient de définir l'état initial. Celà se fait par la ou les transitions d'initialisation. Celles-ci ressemblent à des transitions, mais seules les clauses *to* et *provided* y sont autorisées. Les conditions *provided* peuvent porter, par exemple, sur les paramètres passés dans l'en-tête du module. Le fait d'avoir plusieurs transitions d'initialisation permet une initialisation non-déterministe du module. La partie action permet d'initialiser les variables (Pascal) du module, et on initialise l'état principal par la clause *to*.

III.5. Compléments

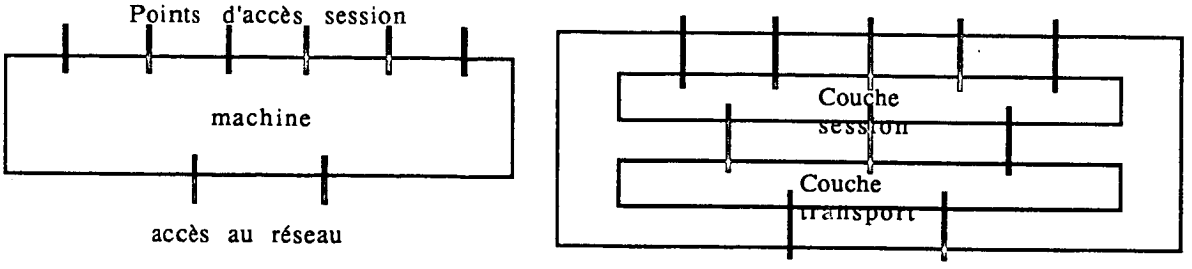
Bien sûr, la description faite ici de la syntaxe Estelle pour les automates n'est pas complète, mais les aspects essentiels y ont été présentés. Mentionnons l'existence d'énoncés ajoutés à Pascal (*all, forone* etc, qui apparaissent dans l'exemple 2 de l'annexe); et la possibilité d'avoir des entrées et sorties sur des canaux internes à un module.

IV. Architecture et configuration

Estelle permet non seulement de décrire séparément chaque module, il offre aussi un moyen de spécifier l'architecture du système distribué. La syntaxe définissant l'architecture se coule dans la syntaxe des modules, grâce à la notion d'emboîtement.

IV.1. Structuration

Considérons le cas d'une machine offrant, dans le cadre du modèle OSI, un service de niveau 5 (session) au dessus d'un réseau X25 (donc au niveau 3). Nous pouvons représenter cette machine par un module Estelle avec une interface haute et une interface basse.



Mais une description plus fine de cette machine mettra en évidence les deux couches (4 et 5: transport et session) qu'elle contient.

A son tour, le module correspondant au transport pourra être découpé en sous-modules, comme celà se fait souvent dans les implantations du protocole de transport ISO, avec un sous-module pour traiter chaque points d'accès haut (il y en a 5 sur la figure) et bas (3). Le nombre de sous-module peut d'ailleurs varier dynamiquement en fonction des connexions demandées par les utilisateurs du service de session: on est ainsi amené à créer des modules, puis à les détruire une fois leur tâche achevée. Le langage Estelle permet ainsi de gérer dynamiquement les modules et leurs connexions.

IV.2. Configuration statique

Pour décrire une architecture statique, il suffit d'écrire dans le corps du module englobant quels sont les exemplaires de (sous-)modules qu'il contient. Ces exemplaires sont décrits dans une partie *modvar* qui est réservée aux variables de type module. Ensuite, on connecte ces exemplaires de modules dans la partie *initialize* du corps de machine. En outre, on affecte à chaque module un corps lui correspondant; et c'est l'occasion d'initialiser ses paramètres. Sur l'exemple précédent, on pourra avoir:

```
type t1_5:1..5; t1_3:1..3; t1_2:1..2;
module machine;
    ip SS:array[t1_5] of PAS_S(serveur);UR:array[t1_2] of PAS_R(utilisateur); end;
body exemple for machine ;
    channel Niv_4 (cinq,quatre); ...
    module session (taille :integer);
        ip H:array[t1_5] of PAS_S(serveur);  B:array[t1_3] of Niv_4(cinq); end;:
    module transport; ...
        ip H:array[t1_3] of Niv_4(quatre); B:array[t1_2] of PAS_R(utilisateur); end;
    body protocole_ISO_5 for session; external;
```

```
body protocole_ISO_4 for transport; external;
    modvar Haut: session; Bas: transport;
    initialize begin
        init Haut with protocole_ISO_5 (256);  init Bas with protocole_ISO_4;
        all i :t1_5 do attach Haut.H[i] to SS[i];
        all i :t1_3 do connect Haut.B[i] to Bas.H[i];
        all i :t1_2 do attach Bas.B[i] to UR[i]
    end;   end; (* body exemple *)
```

L'énoncé *all* spécifie qu'une instruction doit être effectuée pour toutes les valeurs d'un type (ici de 1 à 5 par exemple); contrairement à la boucle *for* de Pascal, l'ordre d'exécution n'est pas spécifié. L'énoncé *connect* relie deux points d'interaction. L'énoncé *attach* a une sémantique différente : il ne signifie pas une connexion entre, par exemple, Haut.H[i] et SS[i], mais que SS[i] n'est que le nom, vu de l'extérieur du module, du point d'interaction H[i] du sous-module Haut.

Avec ce type de construction, on peut décrire toute architecture statique : l'initialisation des modules englobants crée des sous-modules, et cette sous-structuration peut se faire à plusieurs niveaux. Un module englobant est appelé "père" par rapport aux sous-modules qu'il contient directement, et qui sont appelés ses "fils". Les "fils" de ses "fils" sont ses descendants. La structuration à plusieurs niveaux crée ainsi une arborescence dont la racine est le module le plus englobant et les feuilles les modules n'ayant pas de fils, c'est à dire pas de sous-structure.

IV.3. Configuration dynamique

Les constructions introduites ci-dessus permettent de faire évoluer la configuration au cours de l'exécution du système. Une configuration initiale ayant été établie par les parties *initialize*, des créations de modules pourront se faire ensuite dans les parties . En fait, la configuration sera statique si tous les pères ont des parties *trans* vides (ou ne faisant pas de création de modules ou de connexions), et dynamique (évolutive) sinon.

V. Sémantique du parallélisme et de la communication

Ce paragraphe donne un bref aperçu du traitement du parallélisme et de la communication dans Estelle. Le modèle formel général et détaillé du document ISO peut être consulté dans [Dembinski 86]. Un modèle formel correspondant au cas privilégié où l'attribut *systemactivity* est associé à la racine de la spécification est présenté dans [Courtiat 86]. Il permet de faire le lien entre la sémantique d'Estelle et celle des réseaux de Petri temporels [Merlin 76].

V.1. Parallélisme et communication

Comme il a déjà été mentionné, une spécification en Estelle décrit un système hiérarchisé formé de composants séquentiels et non-déterministes (exemplaires de modules) échangeant des messages à travers des liens bidirectionnels entre ports (points d'interaction). A la fois la structure hiérarchique et la structure des liens peuvent changer au cours du temps, transformant alors le système en un système dynamique.

Vu de l'extérieur, un module est une boîte noire, accessible seulement par un ensemble fini de points d'interaction et de variables exportées. L'accès en lecture/écriture depuis l'extérieur aux variables exportées est réservé exclusivement au module père.

Chaque point d'interaction d'un module possède une file FIFO qui mémorise les interactions envoyées à ce module sur ce point d'interaction. Un module peut aussi envoyer des interactions à d'autres modules (et même à lui-même). Ces interactions sont ajoutées à la file du point d'interaction lié à ce moment au point de départ.

Le comportement interne d'un seul module peut être caractérisé en terme de

système à transitions d'états non-déterministe; c'est à dire par un ensemble d'états, un sous-ensemble des états initiaux et la relation de succession d'état. En gros, ces éléments sont déterminés par les parties déclaration, initialisation et transition de la définition du corps du module. Si la partie transition est vide, le module (et ses exemplaires) sont dits inactifs (la relation successeur est vide).

Le comportement du système (donné par la spécification Estelle) est défini de façon opérationnelle en termes de transformations d'une situation globale du système; celle-ci inclut la hiérarchie instantanée des modules avec leur état local et la structure des liens établis entre eux.

Un "pas de calcul", dans cette approche opérationnelle de la sémantique d'Estelle, consiste à exécuter une transition; le calcul commence à partir de la situation initiale. L'exécution d'une transition par un module provoque le changement de l'état local, qui est un changement observable dans la situation globale. En outre, le contenu d'une file d'un autre module peut être affecté par une sortie de la transition. Le principe central d'Estelle est que l'exécution d'une transition est atomique. Une fois que l'exécution est commencée, elle ne peut être interrompue, et conceptuellement, on ne peut pas visualiser les résultats intermédiaires aussi longue que soit la transition. L'atomicité des transitions est une hypothèse importante pour exprimer à un niveau abstrait le comportement d'une spécification, indépendamment de la manière précise dont est mise en oeuvre l'atomicité. L'atomicité a aussi un impact sur l'exécution parallèle des modules.

La façon dont les actions sont entrelacées dans le temps caractérise les séquences possibles de calcul du système. Ces calculs, à leur tour, modélisent les résultats des exécutions réelles admissibles.

V.2. Structuration et parallélisme

Une définition de corps de module (*body*) peut inclure d'autres définitions de modules (appelés fils) et ainsi de suite. La hiérarchie dynamique des modules possède un "squelette" textuel fourni par la spécification Estelle. Bien que le nombre d'exemplaires d'un module donné puisse varier au cours du temps, la position hiérarchique de chaque exemplaire correspond à la position de la définition du module dans le squelette (comme l'illustre l'exemple 2). La façon dont les exemplaires de modules existants se comportent les uns par rapport aux autres (c'est à dire, le degré et la forme du parallélisme et du non déterminisme) est strictement déterminé par :

- le squelette (hiérarchie) textuel des définitions de modules,
- la qualification de leur en-tête: *systemprocess*, *systemactivity*, *process* et *activity*.
- un principe d'Estelle, qui est l'exécution prioritaire des transitions du père par rapport à celles de ses fils.

La combinaison de ces trois aspects offre au concepteur les possibilités suivantes:

- au niveau "haut", une spécification peut être structurée en une configuration fixe (architecture) de systèmes indépendants et s'exécutant de façon asynchrone (*systemprocess* ou *systemactivity*). Ces modules ne peuvent pas être inclus dans des modules actifs puisque cela violerait leur indépendance (par le principe de priorité parent/enfants). En conséquence ces systèmes sont inclus dans des modules non actifs qui ne servent pratiquement qu'au moment de l'initialisation. C'est pourquoi toute initialisation détermine une configuration fixe de modules et établit des liens entre les points d'interaction. Cette structure ne changera pas durant la vie du système; elle est invariante. Dans le cas de l'exemple 1, c'est la structure des exemplaires du module Site, créés pendant une partie initialisation qui n'est pas donnée, qui est invariante. Pour l'exemple 2, c'est la structure formée d'un exemplaire du service Session, de "site_num" exemplaires de l'entité RP et de "cl_num+se_num" exemplaires de clients qui est créée une fois pour toutes.

- Tout module *systemprocess* et par conséquent tout module *process* peut être structuré en sous-modules qui doivent être qualifiés *process* ou *activity*. Leur parallélisme est régulé par le principe de priorité parent/enfants. Seul un module actif peut dynamiquement créer, détruire et changer les connexions de ses exemplaires fils. Ceci est une des raisons du principe de priorité. La priorité joue sur toute la descendance (relation transitive). Ceci exclut tout parallélisme entre modules ayant une relation ancestrale. Intuitivement parlant, chaque fois qu'une transition est tirable à l'intérieur d'un système (pour tous les descendants d'un module), une permission doit être demandée. En outre, cette permission ne peut être accordée que si les transitions permises antérieurement sont déjà terminées (pour être capable à nouveau de résoudre les conflits de priorité). Le parallélisme interne à un *systemprocess* revêt donc un caractère synchrone. Dans l'exemple 2, ce sont les exemplaires des clients et serveurs, fils des sites RP qui sont sous la priorité. Notons que la structure interne d'un site RP est dynamique : les exemplaires de modules traitant les clients et les serveurs sont créés et détruits au gré du père.

- En dernier lieu, on peut décider, au début de chaque niveau de sous structure, de l'exécution non-synchronisée et non-déterministe des modules descendants (tout en préservant la priorité parent/enfants). Ceci s'accomplit en désignant explicitement les modules *activity* ou *systemactivity*. Tous les descendants sont alors forcément aussi du type *activity*.

Pour modéliser le comportement du système complet (ensemble de modules structuré hiérarchiquement dans un arbre de racine exemplaire du module système) dans notre style opérationnel, il est supposé que les systèmes s'exécutent en une succession "d'instantanés" ("snapshots"). Chaque instantané débute par la sélection de toutes les transitions (ou d'une seule pour les modules *activity*) tirables du système qui ne sont pas en conflit ancêtre/descendant. Les transitions sélectionnées sont alors franchies en parallèle et le prochain instantané démarre lorsqu'elles sont toutes terminées. Puisqu'aucune hypothèse n'est faite sur les vitesses relatives d'exécution des transitions et que les résultats peuvent dépendre de ces relations (files communes), tous les entrelacements possibles des transitions doivent être considérés pour modéliser correctement ce parallélisme "synchronisé".

Cet entrelacement "interne" des transitions d'un système est combiné avec ceux des autres afin de modéliser à son tour, le parallélisme asynchrone entre systèmes. L'atomicité des transitions assure que les transitions d'un système sélectionnées dans un instantané ne dépendent que de transitions déjà terminées dans un autre système. Un entrelacement modélise donc l'activité indépendante des systèmes en dépit des messages échangés. Toutes les séquences possibles de transitions produites de cette façon définissent exactement le comportement global de la spécification Estelle.

PARTIE II : Des outils Estelle

L'utilisation d'Estelle, dont l'avenir semble prometteur, dépend de l'existence d'un ensemble cohérent d'outils autour de ce langage. Un environnement logiciel pour la spécification, la validation et la mise en oeuvre de protocoles complexes est nécessaire et parfaitement envisageable. Son domaine d'application déborde le cadre des protocoles OSI pour toucher celui plus général de l'algorithmique distribuée et de la conception des systèmes répartis et des réseaux.

Un certain nombre de méthodes actuelles issues de la recherche sur les problèmes de :

- manipulation des descriptions formelles,
- vérification des algorithmes,
- mise au point et simulation,

- génération de code pour les systèmes distribués,

peuvent s'appliquer au langage Estelle ou un sous-ensemble de celui-ci. Il est alors bien tentant de fédérer les prototypes de recherche autour d'un langage international normalisé pour offrir de bons outils de développement aux concepteurs de systèmes et d'applications réparties.

Nous présentons dans cette partie l'initiative européenne (dans le cadre du projet Esprit) de réalisation industrielle de la version de base d'un tel poste de travail Estelle. Nous décrivons ensuite deux outils (Véda et César), résultats d'une recherche récente sur la validation des algorithmes distribués. A l'occasion d'un transfert industriel promu par le CNET, ces outils acceptent un sous-ensemble du langage Estelle et devraient s'intégrer dans les années à venir au poste de travail Estelle.

I. Les projets Sedos [Diaz 85b] et Sedos-Estelle/Demonstrator

Le but du projet Sedos (*Software Environment for the Design of Open distributed Systems*), soutenu par la Communauté Européenne dans le cadre du programme Esprit, est de définir un ensemble d'outils qui puissent être utilisés dans les différentes phases de la conception et de la mise en oeuvre de systèmes répartis. Ce projet, qui a débuté en Novembre 84 pour une durée de 3 ans, prend pour point de départ les techniques de description formelle développées au sein de l'ISO, Estelle et Lotos. Les partenaires impliqués sont : LAAS, Univ. Twente(Pays-bas), ADI, ICL(Angleterre), Bull, sous contractants: Crei (Milan), Univ. Berlin, HMI(Berlin), Univ. Catania(Italie) et PUM(polyt. Madrid).

Le projet Sedos est structuré en six tâches qui ont pour vocations respectives de:

- de participer au développement des langages Estelle et Lotos et de les appliquer à la description formelle de protocoles et services normalisés,
- d'étudier des techniques permettant la vérification formelle de descriptions Estelle et Lotos,
- de définir des prototypes d'outils traitant les langages Estelle et Lotos, en particulier un éditeur orienté syntaxe, un noyau de simulateur et un aide à l'implantation.

Un premier objectif a ainsi en particulier conduit d'une part à un ensemble de travaux sur la définition du langage Estelle et de sa sémantique formelle. Ces travaux ont contribué de manière significative à l'évolution d'Estelle à l'ISO (en particulier la rédaction du 2ième *Draft Proposal*- Juin 1986) et ont permis une séparation très nette entre la partie proprement Estelle (architecture des modules, comportement des modules sous la forme d'automates étendus, primitives Estelle) et la partie PASCAL qui sert uniquement à représenter la manipulation des variables de données. Le langage Estelle a d'autre part été appliqué à la description de plusieurs protocoles et services normalisés parmi lesquels Réseau, Transport et Session.

En ce qui concerne l'approche Estelle (la seule que nous considérons dans cet article), et compte tenu des résultats intermédiaires obtenus, le projet Sedos est accompagné du projet Sedos-Estelle/Demonstrator, démarré en Juillet 1986, qui a d'abord pour but d'industrialiser les prototypes d'outils développés dans le cadre de Sedos et de les intégrer dans une station de travail Estelle et ensuite d'évaluer cette station de travail dans les quatre domaines d'application suivants : les réseaux d'ordinateurs, les communications spatiales, les protocoles de télécommunication et les réseaux industriels. Les partenaires impliqués sont : Vérilog(Toulouse), Marben(Paris), E2S, sous-contractants : CNET, LAAS, ADI, Bull, ENTEL(Espagne), ETSIT(Univ. Madrid).

Le but du projet Sedos-Estelle/Demonstrator est de démontrer l'applicabilité dans le milieu industriel des prototypes d'outils développés au sein du projet Sedos. Deux objectifs principaux sont ainsi assignés au projet Sedos-Estelle/Demonstrator :

- le développement d'une station de travail Estelle destinée à permettre l'édition, la compilation et la simulation d'un logiciel réparti;
- l'évaluation de cet environnement Estelle sur un ensemble d'applications réalistes de plusieurs algorithmes distribués et protocoles complexes. Le développement de la station de travail Estelle consiste à intégrer dans une même station de travail quatre modules qui sont respectivement :
- un éditeur orienté syntaxe; contrairement au prototype développé dans le cadre de Sedos qui utilise l'environnement Mentor, cet outil (disponibilité envisagée pour le troisième trimestre 1987) utilisera l'environnement Mira pour effectuer l'analyse syntaxique d'une description Estelle ce qui permettra d'offrir de nouvelles fonctionnalités qui sont en particulier :
  -- une interface utilisateur plus sophistiquée (écran haute résolution et multifenêtrage);
  -- une analyse interactive de la description Estelle;
  -- une plus grande portabilité de l'outil grâce à la portabilité de Mira qui est disponible sur toute une gamme de machines.
- un compilateur, destiné à engendrer une forme intermédiaire associée à une description Estelle, dérivé de celui de Sedos, mais plus général afin de garantir la compatibilité de ses interfaces avec tout autre outil aval. Une vérification syntaxique et sémantique poussée sera effectuée dans cet outil, qui devrait être disponible aux environs du quatrième trimestre 1987;
- un générateur et un noyau d'exécution destinés à exécuter une description Estelle sur un système d'exploitation cible; ce noyau d'exécution se chargera en particulier de l'exécution des primitives Estelle et de leur mise en correspondance avec les primitives du système d'exploitation considéré; dans le cas où la description Estelle devra communiquer avec le monde extérieur en utilisant le service Réseau normalisé par l'ISO, la mise en correspondance avec X25 sera effectuée de manière automatique en utilisant le service Réseau défini dans le cadre du projet ROSE;
- un générateur et un noyau de simulation destiné à offrir à l'utilisateur un environnement privilégié d'exécution et de mise au point d'une description Estelle; cet environnement est privilégié dans le sens ou il doit fournir à l'utilisateur de nombreuses facilités interactives et ergonomiques permettant une mise au point rapide de cette description (points d'arrêt, traces, maîtrise des objets Estelle,...).

Un vérificateur, prototype actuellement en cours de développement dans Sedos, prend en compte comme modèle sous-jacent des réseaux de Petri à prédicats [Papapanagiotakis 86]; ce prototype est écrit dans un environment Prolog et permet d'exprimer les propriétés que l'on veut vérifier sur la description Estelle sous la forme d'assertions exprimées en logique temporelle; ce vérificateur pourrait plus tard être intégré à la station Estelle.

La station de travail sera réalisée sur le système UNIX V, choisi par la CEE, et dans l'environnement PCTE (*Portable Common Tool Environment*) promu par le programme Esprit. En ce qui concerne l'évaluation de la station de travail Estelle plusieurs champs d'application sont envisagés, en raison de leur importance :

- réseaux d'ordinateurs: description et implantation des
  -- protocole de TRANSPORT ISO (en particulier la classe 4) et SESSION ISO
  -- protocole FTAM (File Transfer Access and Management) ISO
  -- une passerelle spécifique non ISO (*IBM UNIVAC Protocol Converter* - conversion entre un protocole de la famille IBM 3270 et un protocole spécifique UNIVAC)
- communications spatiales: description et simulation des protocoles de transfert de données normalisés dans le cadre du CCSDS (*Consultative Committee for Space*

Data Systems).

- protocoles de télécommunication: description et simulation des protocoles destinés à offrir des services RNIS (Réseau Numérique à Intégration de Services). Un autre exemple envisagé est celui des protocoles de courrier électronique développés dans le cadre des normes X400 du CCITT.

- réseaux industriels: description et simulation de deux exemples relatifs à la commande répartie d'un champ d'héliostats d'une centrale solaire et a la commande d'une cellule flexible d'assemblage qui paraissent être significatifs de deux grands domaines : les processus continus et les processus discontinus.

II. Le simulateur Véda

II.1. Présentation

Véda est un outil logiciel développé au CNET par le département "Evaluation et Validation de Protocoles". L'objectif était de valider par simulation des spécifications de protocoles décrits dans un langage d'assez haut niveau. Véda est fondé sur les idées développées dans la thèse de Claude Jard. Une présentation générale est publiée dans [Jard 85].

Il est ou a été utilisé au CNET (par une demi-douzaine de départements), à l'ENST, à l'INT, au CNAM, et par les universités de Rennes, Montpellier, Grenoble, Paris VI, Besançon... Les usages varient de l'enseignement à la validation de protocoles, en passant par la modélisation et la conception.

Enfin, mentionnons que Véda est écrit en Prolog (Prolog-CNET, commercialisé sous le nom de Prolog/P) et en Pascal. C'est à notre connaissance le premier compilateur réel d'un langage réaliste écrit en Prolog (par Jean-François Monin). Le programme comporte environ 6500 lignes de Prolog et 2000 lignes de Pascal, structurés en une vingtaine de fichiers ou modules. Pascal est utilisé pour le noyau de simulation et pour optimiser certains prédicats Prolog, comme l'analyse lexicale.

II.2. Fonctionnalités

Véda est essentiellement destiné à la simulation de spécifications de systèmes distribués. Dans la version diffusée, le langage pris en entrée est une version un peu ancienne d'Estelle, qui correspond à la syntaxe qu'avait retenue le CCITT dans l'avis X250.

Après invocation du logiciel, on est en présence d'un interpréteur de commandes (l'approche par "menus", simpliste et lourde, a été délibérément évitée). Une vingtaine de commandes sont disponibles et offrent, avec un choix d'options, les fonctions suivantes.

Aide en ligne : par les commandes "aide", "info" et "listinfo", une documentation en ligne très complète est disponible (environ 5500 lignes d'information, groupées en 42 sujets).

Compilation: Véda compile des spécifications de protocoles. On dispose d'une véritable compilation séparée. Le code produit est un pré-Pascal, qu'un éditeur de liens (commande "code") assemble ensuite, et compile en utilisant le compilateur Pascal du système hôte. NB: les mots-clés sont en français, sauf si on choisit l'option "cles_anglaises" dans la commande "comp".

Simulation de protocoles: Véda permet de simuler l'exécution parallèle d'un programme distribué. Deux modes de simulation sont disponibles: automatique (ou tous les choix non-déterministes sont déterminés de façon aléatoire par le simulateur) et interactif. Le noyau de simulation, qui simule le parallélisme et contient toutes les limites implicites (en nombre de processus, degré de parallélisme etc) est entièrement accessible à l'utilisateur de Véda qui peut ainsi le modifier à sa guise (et en particulier ajuster toutes les limites). De ce fait, Véda n'impose pas de

contraintes de dimensionnement des applications traitées.

Interface et confort: un ensemble de commandes permettent la manipulation de fichiers (en gérant les applications suivant une notion de "cas"), d'accéder au système sous-jacent ou à Prolog, de gérer sa session et d'utiliser des abréviations...

Pour la validation des simulations, Véda offre un moyen souple et puissant de "tracer" les événements auxquels on s'intéresse, ou de faire vérifier en cours de simulation des propriétés logiques sur le comportement du système distribué. Pour celà, on définit dans un des fichiers du cas des programmes "observateurs", décrits dans un langage syntaxiquement proche d'Estelle. Ces observateurs auront accès, au cours de la simulation, aux variables internes de tous les modules et aux interactions échangées [Groz 86].

II.3. Extensions et autres outils

Le CNET poursuit avec Bull une recherche sur les moyens d'utiliser au mieux une simulation: on essaiera de guider la simulation pour explorer plus finement les comportements "limites" ou litigieux du système distribué. Les résultats de cette recherche seront d'abord intégrés à Véda. A terme, les travaux menés sur Véda pourraient s'intégrer au produit que fournira dans trois ans le projet Sedos-Estelle-Demonstrator (auquel le CNET participe). La validation par simulation est complémentaire d'autres méthodes de validation, comme celle proposée par XESAR. L'utilisation de ces outils peut se faire successivement sur des parties du même protocole. Certains aspects, comme la spécification en Estelle, et la spécification des propriétés à vérifier, peuvent être repris par plusieurs outils. Il suffit alors d'avoir les passerelles permettant de passer d'un outil à l'autre...

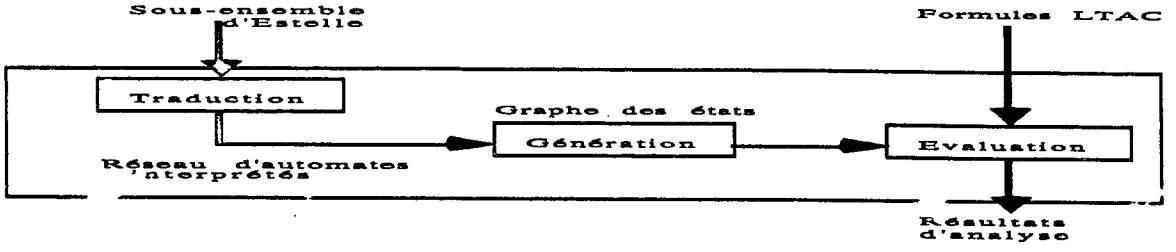
Exemple d'exécution avec Véda de l'algorithme de Dijkstra (cf annexe)
Executeur Veda : version 1_000
Lecture des parametres (entiers ou reels) Temps max de simu et germe (entier) sont les deux premiers :10 5675.
-- debut de la simulation -- noyau VEDA 04/85

```
0.000 : S.SI[0].Droite          --> jeton (blanc)
0.000 : S.SI[3].Gauche          <-- jeton (blanc)
1.049 : S.SI[0].C[ 2]           <== message ()
1.049 : S.SI[2].C[ 0]           ==> message ()
1.702 : S.SI[1].C[ 3]           ==> message ()
1.702 : S.SI[3].C[ 1]           <== message ()
1.801 : S.SI[0].C[ 1]           ==> message ()
1.801 : S.SI[1].C[ 0]           <== message ()
1.847 : S.SI[2].C[ 3]           <== message ()
1.847 : S.SI[3].C[ 2]           ==> message ()
3.047 : S.SI[2].Gauche          <-- jeton (noir)
3.049 : S.SI[2].Droite          --> jeton (noir)
3.049 : S.SI[1].Gauche          <-- jeton (noir)
3.049 : S.SI[1].Droite          --> jeton (noir)
3.049 : S.SI[0].Gauche          <-- jeton (noir)
3.049 : S.SI[0].Droite          --> jeton (blanc)
3.049 : S.SI[3].Gauche          <-- jeton (blanc)
3.049 : S.SI[3].Droite          --> jeton (blanc)
3.049 : S.SI[2].Gauche          <-- jeton (blanc)
3.049 : S.SI[2].Droite          --> jeton (blanc)
3.049 : S.SI[1].Gauche          <-- jeton (blanc)
3.049 : S.SI[1].Droite          --> jeton (blanc)
3.049 : S.SI[0].Gauche          <-- jeton (blanc)
termination detected
Fin normale de la simulation Temps courant : 10
```

III. Le vérificateur César

César est un système de vérification d'applications réparties qui permet de comparer une application décrite par un programme parallèle avec ses spécifications (service) données par un ensemble de formules d'une logique temporelle. Le principe de vérification consiste à traduire la description algorithmique du système étudié vers un graphe d'état fonctionnellement équivalent au programme initial. La conformité du programme aux spécifications est vérifiée par évaluation des formules sur le graphe d'état obtenu. Le principe de César a été présenté dans [Queille 82] et un prototype réalisé sous Multics. Sous l'impulsion d'un contrat CNET, l'IMAG/LGI (Grenoble) et Cap Sogeti Innovation réalisent une version industrielle et performante de César prenant en compte un sous-ensemble du langage Estelle. Cet outil, appelé Xésar, sera disponible (sur SPS7) fin 1986. Quelques principes de la traduction peuvent être trouvés dans [Fernandez 85].

Plus précisément, l'architecture fonctionnelle de Xésar se présente ainsi :



III.1 La génération

Le sous-ensemble d'Estelle considéré est celui défini par l'outil Véda (cf ci-dessus) : c'est à dire qu'il ne prend en compte que l'aspect statique d'Estelle. Comme le processus de vérification travaille sur un graphe d'état fini, diverses restrictions sont imposées par Xésar, comme des bornes raisonnables sur les domaines de valeur des variables d'état.

La fonction traduction produit un programme abstrait qui est ensuite "configuré" (exécution des parties initialisation) pour former un ensemble de processus séquentiels interconnectés (nommé ici réseau d'automates interprétés). La fonction génération simule exhaustivement le réseau d'automates pour produire un graphe d'état.

III.2 L'analyse

La spécification d'un système en Xésar est un ensemble de propriétés de son comportement décrites par des formules d'une logique temporelle arborescente (LTAC). Cette logique est obtenue en étendant le calcul propositionnel avec des opérateurs temporels exprimant des relations sur les séquences de comportement. Elle permet d'exprimer l'atteignabilité possible ou inévitable d'une situation suivant telle ou telle condition. La correspondance logique-programme se fait par des prédicats sur les variables du programme ou des prédicats de position (caractérisant la position de l'état de contrôle du programme par rapport à une transition nommée par une étiquette). Le principe de la méthode de vérification est fondé sur la caractérisation des opérateurs temporels comme points fixes de fonctions monotones que l'on peut évaluer sur le graphe d'état. Le calcul a lieu par itérations successives sur des ensembles d'états. Diverses recherches sont en cours pour offrir aux concepteurs de protocoles un langage de spécification mieux adapté (plus simple à utiliser) à leurs besoins, et une interface homme-machine plus performante.

CONCLUSION

Nous avons essayé dans cet article, de présenter assez complètement, et de façon relativement illustrée, le langage Estelle. Ce langage a montré son aptitude à décrire aussi complètement que possible et sans ambiguïtés, les protocoles de l'ISO et diverses sortes d'algorithmes distribués. Le point fort d'Estelle, c'est d'être conçu dans une perspective de normalisation internationale : de fait, aujourd'hui, il est utilisé par de nombreuses équipes qui ont besoin d'un langage parallèle reconnu et à longue durée de vie. Toutefois, et c'est le cas pour Estelle, la collaboration internationale rend difficile l'émergence d'un "beau" langage, avec une sémantique parfaitement claire et se prêtant bien à la preuve. Nous avons aussi montré qu'un effort important en France et en Europe est engagé sur la réalisation d'outils Estelle et ceci avec des perspectives industrielles encourageantes. Nous espérons enfin que cet article servira de présentation et de référence du langage.

Remerciements

Nous remercions P. Mondain-Monval (Laas) pour la programmation de l'exemple d'appel de procédures distantes et J. Sifakis (Imag) qui nous a permis de présenter une partie du travail réalisé dans son équipe (César). La rédaction de la version révisée doit beaucoup aux différents relecteurs M. Diaz (Laas), JP. Ansart (Adi), JF. Monin (Cnet) et S. Budkowski (Bull).

Bibliographie

[Ansart 82] JP. Ansart, O. Rafiq, V. Chari, *PDIL - Protocol Description and Implementation Language*, Proc. 2nd Ifip workshop on Protocol Specification, Testing and Verification, Idylwild, CA, USA, May 1982, North-Holland ed.

[Ansart 86] JP. Ansart et al, *Software tools for Estelle*, 6th Ifip workshop, Gray Rocks, Montréal, Juin 1986.

[Ayache 83] JM. Ayache, JP. Courtiat, *LC/1, a Specification and Implementation Language for Protocols*, Proc. 3rd Ifip workshop on Protocol Specification, Testing and Verification, Ruschlikon, Suisse, Juin 1983, North-Holland ed.

[Courtiat 86] JP. Courtiat, *Une vue réseau de Petri du langage Estelle*, Congrés ENST-MASI, Nouvelles Architectures pour les Communications, Paris Octobre 1986.

[Dembinski 86] P. Dembinski, *Estelle semantics*, Rapport Sedos O54, Juin 1986.

[Diaz 85a] M. Diaz, P. Mondain-monval, *Appel de Procédures à Distance dans les Réseaux Hétérogènes*, Actes du premier colloque C3 - Angoulême 16-18 Septembre 1985

[Diaz 85b] M. Diaz, C. Vissers, JP. Ansart, Sedos : *Software Environment for the Design of Open distributed Systems*, Proceedings of the Esprit'85 week, North-Holland

[Dijkstra 83] E.W. Dijsktra, W.H.J. Feijen, A.J.M. van Gasteren, *Derivation of a termination detection algorithm for distributed computations*, IPL 16 (1983) pp 217-219

[Fernandez 85] JC. Fernandez, JL. Richier, J. Voiron, *Verification of Protocol Specification using the Cesar system*, 5th Ifip WG6.1 workshop, Moissac, France, juin 1985, north-holland ed.

[Groz 86] R. Groz, *Unrestricted verification of protocol properties on a simulation using an observer approach*, 6th Ifip WG6.1 workshop, Gray Rocks, Montréal, Juin 1986.

[ISO 86a] ISO/TC97/SC21/WG16-1 DP9074, *Estelle : a formal Description Technique based on an Extended State Transition Model*, Juillet 1986.

[ISO 86b] ISO/TC97/SC21/WG16-1 DP8807, *Lotos: a formal Description Technique*, Juillet 1986.

[Jard 85] C. Jard, R. Groz, JF. Monin, *Véda : a sofware simulator for the validation of protocol specifications*, COMNET'85, Budapest, Oct. 85, North-Holland ed.

[Merlin 76] P. Merlin, D. Farber, *Recoverability of Communication Protocols-Implications of a Theoritical Study*, IEEE trans. on Comm., Septembre 1976.

[Milner 80] R. Milner, *A Calculus of Communicating Systems*, LNCS 92, Springer Verlag 1980.

[Papapanagiotakis 86] G. Papapanagiotakis, P. Azema, B. Pradin, *On a Prolog Environment for Protocol Analysis*, Proc. of the 6th International Conference of Distributed Computing Systems - Cambridge (Massachussetts) - May 1986

[Queille 82] JP. Queille, J. Sifakis, *Specification and verification of concurrent systems in Cesar*, Int. symp. of Programming, LNCS 137, 1982.

ANNEXE : Les exemples choisis pour illustrer la présentation

Nous avons choisi d'illustrer la présentation du langage par deux exemples réels, chacun montrant un aspect particulier d'Estelle. Le premier est un exemple typique d'algorithme distribué : il nécessite l'expression d'une algorithmique complexe dans le corps d'un module Estelle et de nombreuses synchronisations. Le second exemple se situe plus dans le domaine "protocole", l'accent est mis particulièrement ici sur le dynamisme et la gestion des modules et connexions.

1. L'algorithme de Dijkstra

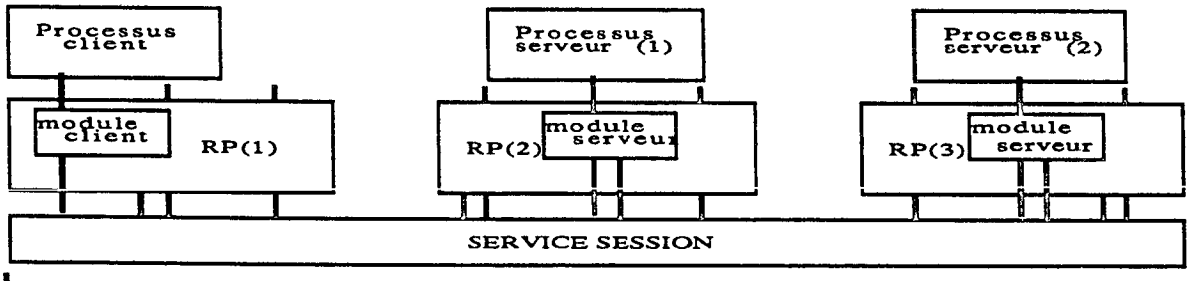
(* Présentation il est difficile d'être plus concis que Dijkstra, aussi le lecteur intéressé pourra se reporter à [Dijkstra 83]. En quelques mots, il s'agit de détecter la terminaison d'un calcul distribué impliquant N machines communiquant par messages. Lorsqu'une machine a fini sa part de calcul, elle passe de l'état actif à l'état passif. Cependant l'arrivée d'un message la réactive. Le problème consiste à faire détecter par un site la terminaison complète du calcul. Pour celà, on suppose qu'on peut faire passer un jeton successivement à toutes les machines (indépendamment de la topologie effective du réseau par lequel passent les messages). On établit ainsi un anneau virtuel sur les N sites qu'on peut numéroter de 0 à N-1, le site 0 étant celui chargé de détecter la terminaison. Pour une détection correcte, Dijkstra montre qu'il faut deux états du jeton, ce qu'il représente par les couleurs blanche et noire. *)

```
body Dijkstra_83 for Site;
    state   passif,actif;
    var ma_couleur: type_couleur; terminaison: boolean; jeton_recu : boolean;
        couleur_recue : type_couleur;
    initialize to actif begin ma_couleur:= blanc; terminaison:= false; couleur_recue := blanc;
        jeton_recu := identite=0 end;
    trans   when Gauche.jeton  (couleur) from passif to same
              begin terminaison:= (identite=0) and (couleur=blanc) and (ma_couleur=blanc) ;
                  if (identite=0) then if not terminaison  (* c'est reparti pour un tour  *)
                      then output Droite.jeton(blanc) else
                       (* TERMINAISON DETECTEE *)
                  else if ma_couleur=noir then output Droite.jeton(noir) (*je le noircis*)
              else output Droite.jeton(couleur); (* je le transmets sans changement*)
          ma_couleur := blanc end;
                                        from actif                (* je conserve le jeton          *)
              begin jeton_recu :=true; couleur_recue :=couleur end;
    trans   from actif to passif   (* jusqu'a passer a l'etat passif                                *)
                                        (* transition supposee spontanee : on ne cherche pas      *)
                                        (* a representer le calcul distribue sous-jacent          *)
              delay (0,4)                       (* par exemple                                    *)
              begin if jeton_recu  then if ma_couleur=noir then output Droite.jeton(noir)
                        else output Droite.jeton(couleur_recue);
                        jeton_recu := false; ma_couleur:= blanc end;
              delay (1,2)                       (* Cette transition represente l'envoi           *)
                                                (* d'un message dans le cadre du calcul           *)
            function un_autre_site :type_sites; primitive;  (* non represente ici *)
                begin ma_couleur:= noir; output CC[un_autre_site].message  end;
          from actif,passif to actif  any autre :type_sites do
              when CC[autre].message
                  begin                 (* pas d'autre action que le changement d'etat          *)
                  end ;                 (*  on ne represente pas le calcul associe             *)
    end;                                                          (* body Dijkstra_83           *)
```

2. L'exemple d'un protocole d'appel de procédures à distance

(* L'exemple considéré décrit une architecture hétérogène fournissant un service fiable d'appel de procédures à distance [Diaz 85a], composée d'un ensemble de sites interconnectés par un service Session ISO. Sur chaque site réside un certain nombre de processus agissant soit en tant que client, soit en tant que serveur. Un processus client invoque des procédures distantes implantées dans les processus serveurs. Un processus serveur peut également se comporter en tant que client, s'il a besoin des services rendus par un autre processus serveur, afin de satisfaire la requête qui lui a été faite. Ceci signifie que les appels imbriqués à des procédures distantes sont pris en compte.

Le mécanisme décrit ici est le suivant : une requête d'appel (RP_CALL_req) soumise par un processus client à une entité locale (RP[i]) provoque la création au sein de cette entité d'un exemplaire de module représentant le client; cet exemplaire de module va engendrer, à travers une connexion session établie préalablement avec l'entité sur le site du serveur invoqué (RP[j]), la création au sein de cette entité distante, d'un exemplaire de module représentant le processus serveur en question. Ces deux exemplaires de module mettent en oeuvre un protocole d'appel synchrone, réalisé par un échange de messages via la connexion Session sous-jacente. Le représentant distant va provoquer, vers le processus serveur concerné après les vérifications nécessaires, l'émission de l'indication d'appel (RP_CALL_ind). Ce mécanisme peut se répéter plusieurs fois selon le nombre d'imbrications successives considérées. La spécification en Estelle présentée décrit essentiellement l'architecture dynamique des entités RP[i] lors de nouvelles requêtes d'appel. Ceci permet d'illustrer par un exemple significatif l'utilisation des primitives *init*, *connect* et *attach*, ainsi que la primitive release utilisée une fois la primitive d'appel de procédure distante complètement satisfaite. Le schéma suivant explicite l'architecture décrite.



I : *point d'interaction (ip)*

*)

```
specification RP_ARCHITECTURE; default individual queue;
const site_num=...;                                   (* nombre de sites                                  *)
    cl_num=...; se_num=...;                           (* nombre maximum de clients et serveurs par site *)
    lip_num=cl_num+2*se_num;                          (* ips RP/SESSION : 1 par client et 2 par serveur *)
    hip_num=cl_num+se_num;                            (* ips RP/USERS   : 1 par client et 1 par serveur *)
                                                          (* lip: Low Interaction Point : vers SESSION   *)
                                                          (* hip: High Interaction Point: vers USERS     *)
                                                            (* ixip: Internal Interaction point, x= l ou h *)
type diag_type=...;
                                                                  (* points d'interactions USERS/entité RP *)
channel RP_ACCESS_TYPE (user,provider);
    by user: RP_CALL_req(calling_user,called_user: data_type; parameters: data_type);
        RP_CALL_resp(results: data_type; diag: diag_type);
    by provider: RP_CALL_ind(calling_user,called_user: data_type; parameters: data_type);
        RP_CALL_conf(results: data_type; diag: diag_type); end;              (* RP_ACCESS_TYPE *)
                                        (* points d'interactions entité RP/service SESSION *)
```

```
channel SESSION_ACCESS_TYPE (user, provider);
    by user: S_CONNECT_req(...); S_CONNECT_resp(...);
    by provider: S_CONNECT_ind(...); S_CONNECT_conf(...); end;      (* SESSION_ACCESS_TYPE *)

module RP_USER_TYPE (us_num, site_num: integer, ...);                    (* en-tête usager RP *)
    ip RP: RP_ACCESS_TYPE (user); end;                                  (* RP_USER_TYPE *)
body RP_USER_BODY for RP_USER_TYPE; external;                           (* corps non spécifié ici *)

module RP_ENTITY_TYPE systemprocess (site_num: integer);                 (* en-tête entité RP *)
    ip U: array[1..hip_num] of RP_ACCESS_TYPE(provider);
       S: array[1..lip_num] of SESSION_ACCESS_TYPE(user); end;          (* RP_ENTITY_TYPE *)
body RP_ENTITY_BODY for RP_ENTITY_TYPE;                                 (* corps entité RP *)
    type cl_hip_range=...; out_lip_range=...;                           (* partition des lips et hips *)
       se_hip_range=...;    in_lip_range=...;
    module CLIENT_TYPE process (num: integer);                          (* en-tête module client *)
       ip U:RP_ACCESS_TYPE(provider); S:SESSION_ACCESS_TYPE(user);
       export release_me : boolean; end;                                (* CLIENT_TYPE *)
    body CLIENT_BODY for CLIENT_TYPE; external;                         (* corps non spécifié ici *)

    module SERVER_TYPE process (num: integer);                          (* en-tête module serveur *)
       ip U:RP_ACCESS_TYPE(provider); S1,S2:SESSION_ACCESS_TYPE(user);
       export release_me : boolean; num:integer; end;                   (* SERVER_TYPE *)
    body SERVER_BODY for SERVER_TYPE; external;                         (* corps non spécifié ici *)
    ip                          (* points d'interactions internes: dialogues entité RP/modules internes *)
       IDLE_RP: array[1..hip_num] of RP_ACCESS_TYPE(user);
       IDLE_S: array[1..se_num] of SESSION_ACCESS_TYPE(provider);
                                        (* variables de types module client et serveur *)
    modvar SERVER: SERVER_TYPE; CLIENT:CLIENT_TYPE;
    trans any hip: cl_hip_range do                                      (* pour tout hip, quand on y *)
       when U[hip].RP_CALL_req                                          (* reçoit une requête d'appel *)
       var lip: out_lip_range begin    lip:=get_lip_from(called_user);  (* déterminer le lip *)
          init CLIENT with CLIENT_BODY(hip);                            (* créer un module client *)
          connect CLIENT.U to IDLE_RP[hip];                             (* le connecter à un hip interne *)
          output IDLE_RP[hip].RP_CALL_req;                              (* lui passer la requête *)
          disconnect CLIENT.U;                                          (* déconnecter l'hip interne *)
          attach CLIENT.U to U[hip];                                    (* attacher le module au hip *)
          attach CLIENT.S to S[lip];                                    (* attacher le module au lip *)
          end;
       any lip: in_lip_range do                                         (* pour tout lip, quand on y reçoit *)
          when S[lip].S_CONNECT_ind                                     (* une indication de connexion *)
             provided is_correct(S_CONNECT_ind)                         (* si elle est correcte *)
             begin  init SERVER with SERVER_BODY(lip);                  (* créer un module serveur *)
                connect SERVER.U to IDLE_RP[lip];                       (* le connecter a un hip interne *)
                attach SERVER.S1 to S[lip];                             (* l'attacher au lip *)
             end;
             provided otherwise                                         (* si l'indication est incorrecte *)
             begin                                                      (* refuser la connexion session *)
                output S[lip].S_CONNECT_resp(...,reject,...);           end;
       any ihip: se_hip_range do                                        (* pour tout hip interne, quand on *)
          when IDLE_RP[ihip].RP_CALL_ind                                (* y reçoit une indication d'appel *)
             provided is_correct(RP_CALL_ind)                           (* si elle est correcte *)
             var hip :se_hip_range begin
                hip:=get_hip_from(called_user);                         (* déterminer le hip, et y *)
                output U[hip].RP_CALL_ind;                              (* émettre l'indication d'appel *)
                disconnect IDLE_RP[ihip];                               (* déconnecter l'hip interne *)
                forone SERVER:SERVER_TYPE                               (* déterminer le module serveur *)
                suchthat (SERVER.num=ihip) do                           (* attaché à IDLE_RP[ihip] *)
                   begin attach SERVER[ihip].U to U[hip];               (* attacher le module au *)
```

```
                    attach SERVER[ihip].S2 to IDLE_S[ihip]; (* hip et à un lip interne *)
                end;    end;
            provided otherwise                          (* si l'indication d'appel est incorrecte *)
            var diag: diag_type; begin
                diag:= diagnostic(...);                                  (* diagnostic d'erreur *)

                output IDLE_RP[ihip].RP_CALL_resp(dnull,diag); (* envoyer la réponse  *)
                disconnect IDLE_RP[ihip];                                (* détacher l'hip interne *)
                end;
        any ilip: out_lip_range do                          (* pour tout lip interne, quand on y *)
            when IDLE_S[ilip].S_CONNECT_req                 (* reçoit une requête de connexion  *)
                provided is_correct(S_CONNECT_req)          (* si elle est correcte             *)
                var lip: out_lip_range;   begin
                    disconnect IDLE_S[ilip];                (* déconnecter le lip interne       *)
                    lip:=get_ip_from(called_ssap_ad);       (* déterminer le lip                *)
                    output S[lip].S_CONNECT_req;            (* y passer la requête              *)
                    attach SERVER[ilip].S2 to S[lip];       (* y attacher le module serveur *)
                    end;
                provided otherwise                                       (* si elle est incorrecte *)
                    begin                                                (* refuserla connexion  *)
                        output IDLE_S[ilip].S_CONNECT_conf(...,reject,...);      end;

    trans                                                                (* gestion des modules *)
        provided exist CLIENT: CLIENT_TYPE          (* si il existe un module *)
            suchthat (CLIENT.r  se_me)              (* client ayant ter  né  *)
            begin  forone CLIENT: CLIENT_TYPE suchthat (CLIENT.release_me) do
                begin                   (* la primitive forone donne l'identificateur réel du module *)
                                                                         (* détacher le client de tous les ips *)
                    release CLIENT;                                      (* détruire ce module client        *)
                end;   end;
        provided exist SERVER: SERVER_TYPE                               (* si il existe un module *)
            suchthat (SERVER.release_me)                                 (* serveur ayant terminé  *)
            begin  forone SERVER: SERVER_TYPE suchthat (SERVER.release_me) do
                begin                           (* détacher le serveur de tous les ips et détruire ce *)
                    release SERVER;     (* module serveur, ce qui déconnecte les ips internes *)
                end;   end;
    end;                                                                 (* RP_ENTITY_BODY *)
    module SESSION_SERVICE_TYPE systemprocess (...);                     (* en-tête module SESSION *)
        ip S: array[1..site_num,1..lip_num] of SESSION_ACCESS_TYPE(provider); ...;
    end ;                                   (* SESSION_SERVICE_TYPE *) (* corps non spécifié ici *)
    body SESSION_SERVICE_BODY for SESSION_SERVICE_TYPE; external;

modvar  US: array[1..user_num,1..site_num] of RP_USER_TYPE;
        RP: array[1..site_num] of RP_ENTITY_TYPE;   SS: SESSION_SERVICE_TYPE;
initialize  (* création et connexion des modules US,RP et SESSION *)
    var k : integer ;  begin  init SS with SESSION_SERVICE_BODY(...      (* créer service session  *)
        all i : site_num    do begin                                     (* sur tous les sites, faire :  *)
        init RP[i] with RP_BODY_TYPE(i,...);                             (* créer l'entité RP[i] et la  *)
        all j : lip_num                                                  (* connecter au service session *)
        do connect RP[i].S[j] to SS.S[i,j];
        for k:=1 to (cl_num + se_num)                                    (* pour les usagers du site, qu' *)
        do begin                                                         (* ils soient clients ou *)
        init US[k,i] with US_BODY_TYPE(i,k,...);                         (* serveurs: les créer,  *)
        connect US[k,i].RP to RP[i].U[k];                                (* les connecter à RP[i] *)
        end;    end;   end;
end.                                                                     (* RP_ARCHITECTURE *)
```


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