PROJET CESAR

Laboratoire de Génie Informatique de Grenoble
Groupe Spécification et Analyse des Systèmes
BP 68
38402 Saint Martin d'Hères cedex
France
Tél. 76-51-48-32

Utilisation du Système CESAR
pour la Vérification de Protocoles
spécifiés en LOTOS

Hubert Garavel

RT Cesar n° 2 Décembre 86

Utilisation du système CESAR

pour la vérification de protocoles spécifiés en LOTOS

Rapport technique CESAR

Hubert GARAVEL

Institut d'Informatique et de Mathématiques Appliquées de Grenoble
Laboratoire de Génie Informatique
Groupe Spécification et Analyse des Systèmes
BP 68, 38402 St MARTIN d'HERES cedex

ABSTRACT

Ce rapport décrit les diverses approches de la vérification automatique des systèmes parallèles et présente le logiciel CESAR développé au Laboratoire de Génie Informatique.

On montre comment CESAR peut être utilisé pour la validation de protocoles ISO sur deux modélisations, spécifiées dans le langage LOTOS, d'un protocole de la couche transport

---

[1] Ce travail a été effectué dans le cadre du projet SEDOS avec la collaboration d'Elie NAJM

Agence de l'Informatique Tour Fiat cedex 16, 92084 PARIS-LA-DEFENSE

1. Introduction

Actuellement, on assiste à un important effort pour normaliser les communications entre ordinateurs, en particulier dans le cadre de l'ISO. La constitution de réseaux hétérogènes est envisageable si les constructeurs acceptent la normalisation et proposent des systèmes "ouverts". Les enjeux politiques et économiques de cette normalisation sont considérables.

Afin de permettre la définition formelle des protocoles et des services, l'ISO a conçu deux langages de spécification du parallélisme, ESTELLE et LOTOS. ESTELLE est une extension de PASCAL comprenant des primitives de synchronisation fondées sur le modèle des systèmes de transitions. LOTOS ([L 84], [B 85], [NB 86], [EFH 83]) est un langage plus novateur et de plus haut niveau basé sur les algèbres de processus, en particulier CCS [M 80]. Il faut noter que LOTOS et ESTELLE sont des langages exécutables, bien qu'ils soient utilisés dans un but de spécification; la réalisation de compilateurs est en cours.

La communauté européenne encourage cet effort à travers un projet ESPRIT, baptisé SEDOS, qui comporte deux groupes: LOTOS et ESTELLE, et trois thèmes de recherche: définition formelle, compilation, vérification. Le présent travail a été fait dans le cadre du groupe c2 qui étudie et développe des techniques et des outils de vérification adaptés à LOTOS.

L'équipe "Spécification et Analyse des Systèmes" du LGI a conçu le système CESAR qui permet la preuve de systèmes parallèles de processus communicants. L'objectif de ce travail est l'application de CESAR pour la vérification de protocoles et de services exprimés en LOTOS. Il faut préciser ici qu'une seconde version de CESAR destinée au langage ESTELLE est actuellement en cours de réalisation pour un contrat liant le LGI, le CNET et la société CAP SOGETI Innovation.

La mise en oeuvre de CESAR a été faite sur un exemple concret: le protocole de transport. Deux spécifications en LOTOS de ce protocole ont été développées par Elie NAJM (Agence de l'Informatique). Les différentes étapes du travail ont été la compréhension du fonctionnement du protocole, la traduction des descriptions LOTOS dans le langage qui sert à décrire les programmes fournis à CESAR et l'élaboration de formules logiques servant à spécifier le fonctionnement correct du protocole.

2. Techniques de vérification des systèmes réactifs

Vérifier un programme, c'est le comparer avec une "spécification" (c'est-à-dire une description, formelle ou non, des propriétés que doit vérifier le programme) pour s'assurer qu'il la satisfait.

Dans notre cas on s'intéresse aux programmes "réactifs" [P 86] c'est-à-dire aux programmes qui ne se terminent pas forcément et dont la caractéristique principale est d'interagir avec leur environnement (par opposition aux programmes "transformationnels" qui prennent des données en entrée et fournissent des résultats en sortie lorsqu'ils se terminent). Comme exemple de systèmes réactifs on peut citer le contrôle de processus temps-réel, la gestion d'environnements interactifs, les bases de données réparties et les protocoles de télécommunications.

Les systèmes réactifs ne peuvent pas être uniquement spécifiés par leurs états initiaux et terminaux: une spécification adéquate doit porter sur les "comportements", en termes de séquences d'états et/ou de transitions. Suivant la nature des spécifications on distingue deux classes de techniques de preuve:

- **spécifications algébriques:**

  Dans cette approche les spécifications sont exprimées sous la forme d'un programme interprétable par une machine abstraite. Bien souvent les spécifications sont données dans le même langage que les programmes. Le programme et sa spécification constituent alors deux solutions du même problème; la spécification est concise et trivialement correcte alors que le programme est une description plus détaillée qui tient compte des contraintes d'implémentation.

  Le problème de la vérification est alors ramené à celui de savoir si le programme est équivalent à sa spécification ou, du moins, s'il implémente toutes les propriétés de sa spécification. On recherche donc des relations d'équivalence ou d'inclusion entre programmes.

  Mathématiquement il est commode, pour exprimer ces relations, de représenter les programmes par des termes d'une algèbre: [M 80], [HM 85], [BK 85], [BHR 84].

- **spécifications comportementales:**

  La seconde approche utilise un langage adapté aux spécifications, radicalement différent du langage de description des programmes qui, par nature, est algorithmique. Ce langage doit être assez puissant et assez abstrait pour décrire les propriétés des programmes sans indépendamment de l'implémentation [DR 86] [S 86].

  Comme langage de spécifications on utilise généralement le langage des formules d'une logique temporelle. Les logiques temporelles sont des extensions du calcul des prédicats obtenues en rajoutant des opérateurs qui permettent d'exprimer des propriétés d'ordonnancement sur le passé ou le futur, la possibilité, la certitude, ...

  La spécification étant donnée sous la forme d'un ensemble de formules, le problème de la preuve est alors ramené à celui de la recherche d'une relation de satisfaction de la forme "programme satisfait formule".

Si l'on compare ces deux approches, algébrique et logique, on constate que les spécifications logiques possèdent plusieurs avantages marquants:

- elles sont abstraites, c'est-à-dire complètement indépendantes des considérations d'implémentation
- elles sont facilement modifiables, ce qui confère à la démarche de spécification un caractère flexible
- elles permettent d'exprimer simplement des propriétés globales du système, comme l'exclusion mutuelle, la terminaison ou la non-terminaison, l'absence de blocage, l'absence de famine, l'équité, ...

mais le principal problème est celui de la complétude des spécifications: l'ensemble de formules est-il suffisamment contraignant pour n'être satisfait que par des programmes corrects?

D'un autre côté plusieurs arguments plaident en faveur des spécifications algébriques:

- elles sont mieux adaptées à la description des propriétés de séquencement, c'est-à-dire à la caractérisation des situations où il existe une relation d'ordre (précédence) entre événements
- elles sont basées sur des concepts primitifs, les états et les transitions: les descriptions sont concrètes et facilement compréhensibles par l'utilisateur
- il est possible de représenter graphiquement les systèmes et leurs spécifications

En revanche, on doit reconnaître aux spécifications algébriques certains inconvénients:

- elles ne sont guère abstraites, elles tendent à décrire comment le programme doit être implémenté; si on choisit une autre implémentation, la preuve devient difficile
- de plus l'abstraction n'est pas inhérente à l'approche: les spécifications sont souvent établies à partir d'une implémentation en masquant certains détails et non par raffinements successifs
- certaines propriétés globales (dont l'exclusion mutuelle) ne peuvent pas s'exprimer en termes de comportements

3. Présentation informelle du système CESAR

CESAR [Q 82] [S 83] est un outil d'analyse statique destiné à la vérification des systèmes de processus communiquants. Il permet de comparer un programme avec ses spécifications données par un ensemble dé formules de logique temporelle. Il y a différentes versions de CESAR qui diffèrent par le langage de programmation employé. Pour ce travail, la version QUASAR a été utilisée: les processus sont décrits dans un langage de type CSP, appelé ici LDC (Langage de Description de CESAR). Une nouvelle version, baptisée XESAR, est actuellement en cours de développement: elle est destinée à vérifier des programmes écrits en ESTELLE.

Figure 1: schéma fonctionnel de CESAR

Le programme à étudier est formé par la composition en parallèle de processus séquentiels. Avant la vérification, ce programme est transformé en un graphe d'états ("système de transitions") qui décrit le comportement global de l'ensemble des processus. Cette conversion est faite automatiquement par simulation exhaustive.

Les spécifications sont données par un ensemble de formules d'une logique temporelle (baptisée CTL pour "Conditional Temporal Logic"). Il s'agit d'une logique du temps arborescent.

La preuve n'utilise pas de techniques déductives (qui s'appuieraient sur des principes d'inférence tels que la déduction naturelle ou la résolution) ni sur des systèmes de réécriture. Au contraire, une formule est directement et complètement évaluée sur le graphe d'états de la même manière que les équations d'analyse du flux des données sont résolues dans les outils d'analyse statique et d'optimisation des programmes.

Les principes et le fonctionnement de CESAR sont décrits dans [Q 82] et [S 83]. On trouve des exemples d'utilisation de CESAR dans [FFS 84] et [FRV 85].

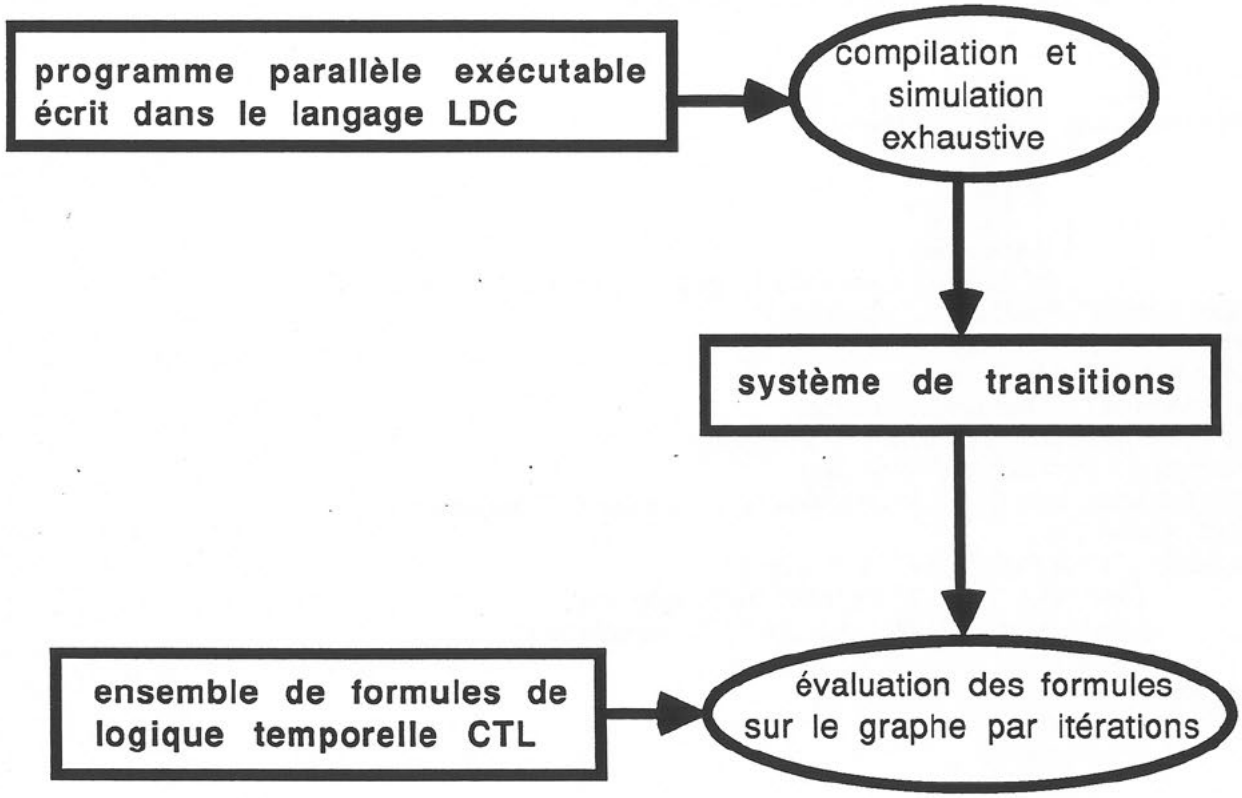


Figure 1: schéma fonctionnel du système CESAR

4. Présentation du langage de description de CESAR (LDC)

La syntaxe de ce langage de type CSP est donnée par les règles grammaticales suivantes [S 83] [FSS 84] où:

les éléments terminaux sont en caractères gras
les éléments non-terminaux sont caractères normaux
: dénote une définition de non-terminal terminée par un point
| dénote l'alternative
[x] dénote zero ou une occurence de x
(x)* dénote zero ou plusieurs occurences de x
(x)+ dénote une ou plusieurs occurences de x

```
program : task "." .
task : elementary_task | composed_task .
elementary_task : "task" identifier ";"
                  [ input ]
                  [ output ]
                  [ declaration ]
                  [ initialization ]
                  "do" guarded_command ( "|" guarded_command )* "od" .
input : "input" identifier ( "," identifier )* .
output : "output" identifier ( "," identifier )* .
declaration : "declare" ( dcl )+ .
dcl : identifier ":" number ".." number .
initialization : "init" assignment ( "," assignment )* .
assignment : identifier ":=" expression .
guarded_command : ( label )* condition ":" [ exchange ( "," assigment )* ] .
label : { identifier } .
exchange : "!" identifier [ ":=" expression ] |
           ( identifier "," )+ "!" identifier ":=" expression |
           [ ( identifier "," )* identifier ":=" ] "?" expression |
           assignment .
composed_task : "cotask" identifier ";"
                [ input ]
                [ output ]
                ( task ";" )+
                [ diffusion_channels ]
                [ communication_channels ]
                "body" task_instance ( "//" task_instance )* .
diffusion_channels : "broad" identifier ( "," identifier )* ";" .
communication_channels : "port" identifier ( "," identifier )* ";" .
task_instance : identifier [ "(" identifier ( "," identifier )* ")" ] .
```

Les éléments non-terminaux suivants sont définis selon la syntaxe de PASCAL

identifiers (le caractère "_" est admis)
number : entier signé
condition : expression booléenne
expression : expression entière

Une tâche élémentaire est un processus sequentiel non-déterministe. On l'introduit par "**task**" suivi du nom de la tâche et de la déclaration optionnelle des portes. Les portes de reception sont déclarées dans la section "**input**" tandis que les portes d'émission le sont dans la section "**output**". Toutes les portes doivent avoir des noms distincts et on suppose qu'elles ne peuvent recevoir ou émettre

que des valeurs entières. L'ordre dans lequel les portes sont déclarées est significatif.

Les variables internes des tâches élémentaires ont pour types des intervalles bornés de l'ensemble des entiers relatifs. Elles sont déclarées dans une section qui commence par "**declare**". Pour chaque variable on précise les bornes de son intervalle de définition. La section de déclaration peut être suivie d'une section (optionnelle) d'initialisations introduite par "**init**". Les variables non initialisées reçoivent une valeur arbitraire de leur intervalle de définition.

Le corps d'une tâche élémentaire se compose d'une boucle non-déterministe "**do**"..."**od**" unique contenant un ensemble de commandes gardées. Chaque commande gardée est formée d'une condition suivie du symbole ":" et d'un ensemble d'affectations simultanées éventuellement précédées par un échange (voir ci-dessous).

Un échange peut être une émission ou une réception. Une émission (resp. une réception) est une affectation dont la partie gauche (resp. droite) est un nom de porte précédé du symbole "**!**" (resp. "**?**"). Cette porte doit appartenir à la liste des portes de sortie (resp. d'entrée). Une valeur lue ou reçue peut être affectée simultanément à plusieurs variables internes. Lorsque la valeur transmise est sans intérêt (cas de la synchronisation "pure") on peut l'ommettre ainsi que d'affectation ":=".

Chaque commande gardée peut être précédée d'un ensemble d'étiquettes qui seront référencées dans les formules logiques.

Exemple

Les automates d'états finis peuvent facilement être décrits comme des tâches élémentaires. L'état courant est mémorisé par une ou plusieurs variables internes. L'exemple suivant illustre ce propos:

```
task Ti;
input
    tdisind1, tconconf, tdataind1;
output
    tconreq, tdisreq1, tdatareq1;
declare
    STATE : 0..2;
init
    STATE := 0;
do
    STATE = 0 : !tconreq, STATE := 1 |
    STATE = 1 : !tdisreq1, STATE := 0 |
    STATE = 1 : ?tdisind1, STATE := 0 |
    STATE = 1 : ?tconconf, STATE := 2 |
    STATE = 2 : !tdisreq1, STATE := 0 |
    STATE = 2 : ?tdisind1, STATE := 0 |
    STATE = 2 : !tdatareq1, STATE := 2
    STATE = 2 : ?tdataind1, STATE := 2
od
```

Une tâche composée est déclarée à l'aide du symbole "**cotask**" suivi du nom de la tâche et des définitions optionnelles de portes. Comme pour les tâches élémentaires, ces portes doivent avoir des noms distincts et ne peuvent transmettre que des valeurs entières.

La déclaration d'une tâche composée est suivie:

- de la déclaration des tâches, élémentaires ou composées, à partir desquelles elle est construite (on les appelera sous-tâches)
- de la déclaration des canaux de diffusion et de communication utilisés pour établir des connexions entre les portes des sous-tâches
- du corps de la tâche

Les canaux de communication sont introduits par "**port**". Ils permettent la communication par rendez-vous entre une tâche émettrice et une tâche réceptrice. Les canaux de diffusion sont déclarés avec "**broad**". Ils autorisent le rendez-vous de N tâches, un processus émettant et les (N-1) autres recevant.

Le corps d'une tâche composée est introduit par "**body**" et il contient un ensemble d'instanciations des sous-tâches séparées par "//". Une instanciation de sous-tâche se compose du nom de la sous-tâche suivi d'une liste de canaux ou portes de la tâche composante. La correspondance entre les paramètres formels (ie les portes de la sous-tâche) et les paramètres effectifs (les canaux et/ou les portes de la tâche composée) est faite par position.

Exemple

L'exemple est une version très schématisée du protocole du transport qui sera étudié plus loin:

```
cotask HIGH;

task Ti;
input
    tdisind1_Ti, tconconf_Ti, tdataind1_Ti;
output
    tconreq_Ti, tdisreq1_Ti, tdatareq1_Ti;
...

task Ta;
input
    tconind_Ta, tdisind2_Ta, tdataind2_Ta;
output
    tdisreq2_Ta, tconresp_Ta, tdatareq2_Ta;
...

task Tia;
input
    tconreq_Tia, tdisreq1_Tia, tdatareq1_Tia;
output
    tdisind1_Tia, tconind_Tia, tdataind2_Tia, tsync_Tia;
...

task Tai;
input
    tconresp_Tai, tdisreq2_Tai, tdatareq2_Tai, tsync_Tai;
output
    tdisind2_Tai, tconconf_Tai, tdataind1_Tai;
...
```

```
port
  tconreq, tconind, tconresp, tconconf,
  tdisreq1, tdisind1, tdisreq2, tdisind2,
  tdatareq1, tdataind1, tdatareq2, tdataind2,
  tsync;

body
  Ti (tdisind1, tconconf, tdataind1, tconreq,
     tdisreq1, tdatareq1) //
  Tia (tconreq, tdisreq1, tdatareq1, tdisind1,
     tconind, tdataind2, tsync) //
  Tai (tconresp, tdisreq2, tdatareq2, tsync,
     tdisind2, tconconf, tdataind1) //
  Ta (tconind, tdisind2, tdataind2, tdisreq2,
     tconresp, tdatareq2).
```

5. Présentation de la logique CTL

[EH 82] décrit un ensemble de logiques du temps arborescent construites autour du système UB ("unified system of branching time") et les classe selon leur pouvoir d'expression. Le système CESAR utilise la plus puissante de ces logiques, la logique CTL ("conditionnal branching time logic") comme langage de spécification [Q 82] [S 83].

On note s |= p le fait qu'un état s du système de transitions satisfasse la formule p. On définit cette relation de satisfaction par récurrence de la manière suivante:

si f est une expression employant les variables d'état du programme
à vérifier et les opérateurs disponibles en PASCAL
s |= f ssi les valeurs des variables d'état pour l'état s satisfont f

On dispose de prédicats prédéfinis portant sur les états et les transitions du système:

init : prédicat vrai pour les états initiaux
sink : prédicat vrai pour les états terminaux (puits)
enable (a) : prédicat vrai pour les états à partir
desquels on peut faire la transition a
after (a) : prédicat vrai pour les états que l'on peut
atteindre par une transition a

On définit ensuite les opérateurs logiques usuels:

s |= not p ssi s |= p est faux
s |= p and q ssi (s |= p) et (s |= q)
s |= p or q ssi (s |= p) ou (s |= q)
s |= p => q ssi s |= (not p or q)
s |= p <> q ssi s |= (p => q) and (q => p)

La logique CTL utilise des opérateurs temporels conditionnels all, pot, some, inev, fair définis comme suit:

s |= all [p] q ssi
pour tout chemin d'exécution s0 = s, s1, ...
pour tout k >= 0
(s0 |= p et ... s(k-1) |= p) => sk |= q

-- pour tout chemin issu de s, q est vrai jusqu'à
-- ce que p devienne faux

s |= pot [p] q ssi
il existe un chemin d'exécution s0 = s, s1, ...
il existe k >= 0
(s0 |= p et ... s(k-1) |= p) et sk |= q

-- il existe un chemin issu de s contenant un état sk
-- vérifiant q et tel que p soit vrai de s à sk

s |= some [p] q ssi
   il existe un chemin d'exécution s0 = s, s1, ...
   pour tout k >= 0
   (s0 |= p et ... s(k-1) |= p) => sk |= q

-- il existe un chemin issu de s tel que q est vrai
-- jusqu'à ce que p devienne faux

s |= inev [p] q ssi
   pour tout chemin d'exécution s0 = s, s1, ...
   il existe k >= 0
   (s0 |= p et ... s(k-1) |= p) et sk |= q

-- tout chemin issu de s contient un état sk
-- vérifiant q et tel que p soit vrai de s à sk

s |= fair [p] q ssi s |= all [not q] pot [p] q

-- fair est analogue à inev mais introduit la notion
-- d'équité dans les choix non-déterministes [QS 83]

On définit des opérateurs dérivés à partir des opérateurs primitifs:

[] p = init => all p
all p = all [true] p
pot p = pot [true] p
some p = some [true] p
inev p = inev [true] p
fair p = fair [true] p

La relation de satisfaction est ainsi définie: le programme P satisfait la formule p ssi tous les états s du système de transitions correspondant à P vérifient:

s |= p

6. Présentation du protocole de transport

La normalisation ISO des protocoles et des services s'appuie sur la définition de "couches" qui définissent 7 niveaux d'abstraction allant du bas niveau (communication physique) jusqu'au plus haut (application logicielle). Le protocole de transport fait partie des fonctionnalités de la 4-ième couche ("transport") et son implémentation utilise les fonctionnalités de la 3-ième couche (réseau ou "network").

La description complète de ce protocole est assez complexe (la description semi-formelle fournie par l'ISO comprend une centaine de pages). Pour plusieurs raisons il est actuellement impossible de prouver automatiquement le protocole complet:

- fondamentalement, le comportement du protocole ne peut pas être décrit par un automate d'états finis car il autorise la création dynamique d'un nombre quelconque de processus. On pallie cette difficulté en bornant le nombre de processus pouvant être créés
- de manière plus contingente, la version initiale du système CESAR (la seule à être disponible lorsque ce travail a été effectué) ne fonctionne pas pour les systèmes qui ont plus de 3000 états

Deux descriptions simplifiées en LOTOS ont été développées pour servir d'exemples à la vérification automatique. Par rapport à la version complète, plusieurs restrictions ont été imposées:

- chacun des deux sites qui communiquent ne peut créer qu'un seul processus
- les deux sites sont particularisés et ne jouent plus un rôle symétrique: on distingue le site "initiateur" qui décide d'établir la communication et le site "accepteur" qui lui répond
- les transferts de valeurs ne sont pas modélisés; pratiquement, un transfert est représenté par un rendez-vous sans échange de valeur. Par conséquent les deux descriptions ne font intervenir que de la synchronisation pure

Le fonctionnement de ce protocole simplifié s'apparente à une communication téléphonique ordinaire. On distingue 3 phases successives:

- la connexion: initialement il n'existe aucune connexion entre les sites au niveau transport. L'initiateur appelle l'accepteur en lui envoyant une "demande de connexion" ("connection request" ou "tconreq"). Le réseau transmet à l'accepteur une "indication de connexion" ("connection indication" ou "tconind"). Ce dernier émet alors une "réponse de connexion" ("connection response" ou "tconresp"). Le réseau renvoie alors à l'initiateur une "confirmation de connexion" ("connection confirm" ou "tconconf"). A partir de ce moment la communication est établie. Il faut noter que le processus de connexion ne réussit pas toujours: la connexion peut être refusée soit par le réseau ou soit par le site accepteur; ce refus prend la forme d'une déconnexion (voir plus loin)

Figure 2: mécanisme de connexion

- le transfert: à partir de ce moment, chacun des deux sites peut émettre ou recevoir des données; la communication est bi-directionnelle et tout se passe comme s'il y avait deux lignes uni-directionnelles montées en duplex. Le mécanisme est le suivant: le site qui veut émettre envoie au réseau une "requête de données" ("data request" ou "tdatareq"). Le réseau transmet alors un message "indication de données" ("data indication" ou "tdataind") à l'autre site.

Figure 3: mécanisme de transfert

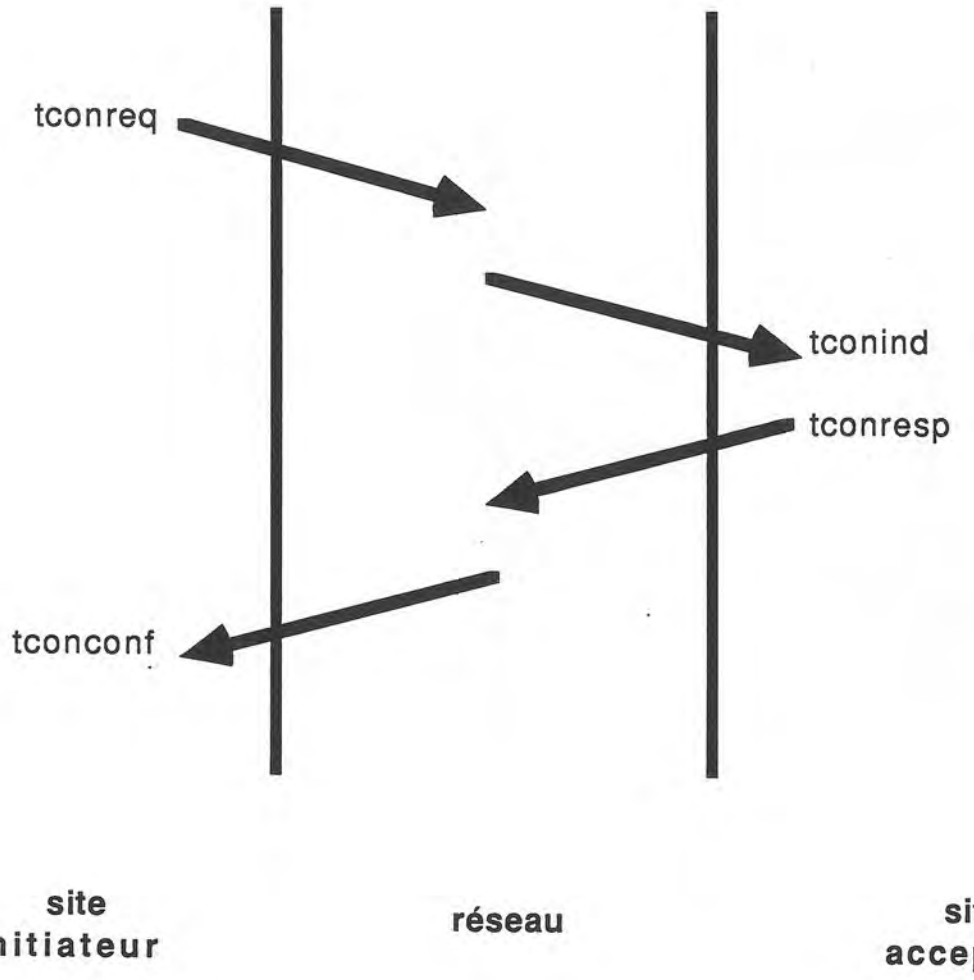


Figure 2: mécanisme de connexion

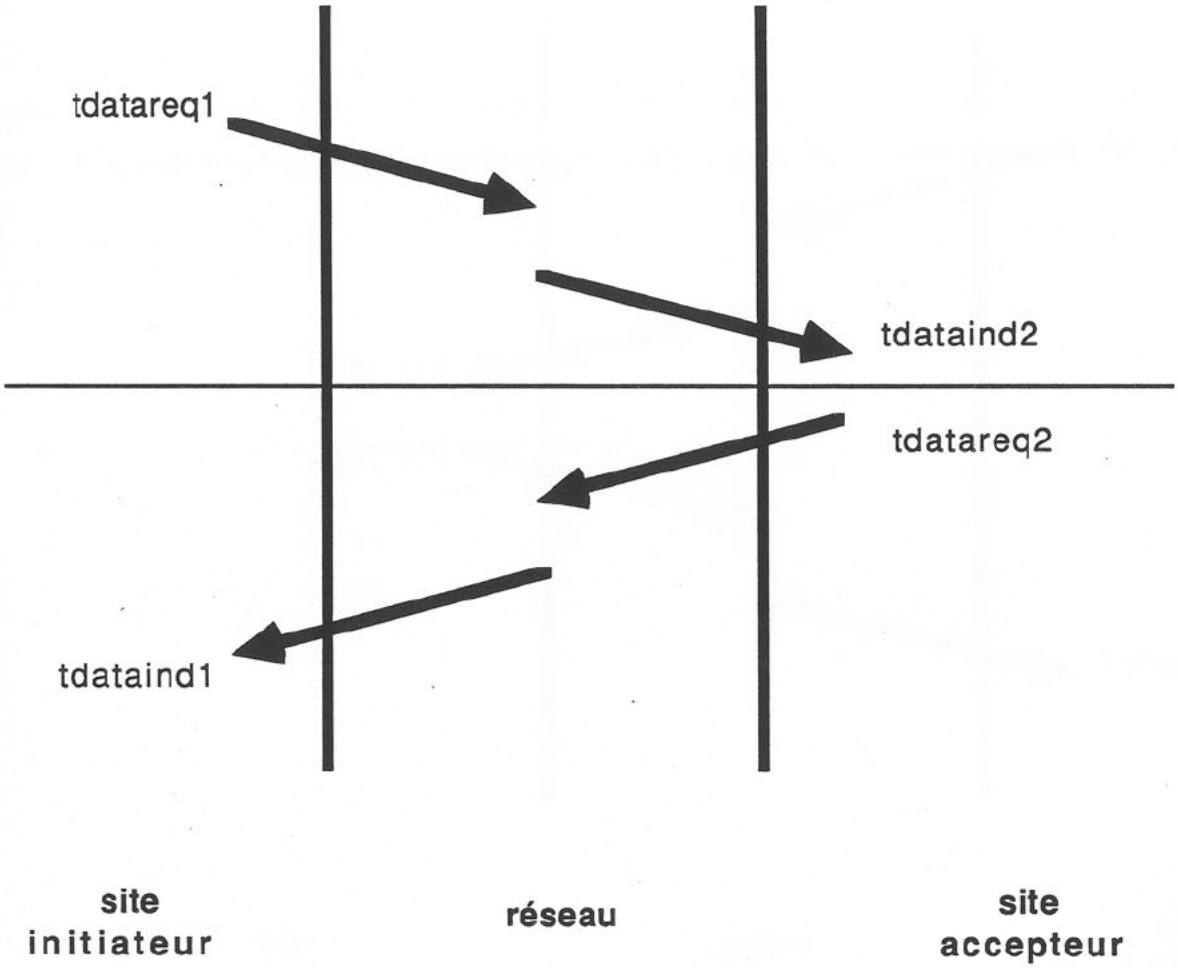


Figure 3: mécanisme de transfert

- la déconnexion: à tout instant la communication peut être interrompue, soit par le réseau lorsque, par suite d'une défaillance, celui-ci n'est plus capable d'assurer un service correct, soit parce que l'un des sites a décidé la rupture. Dans le premier cas le réseau envoie à chacun des sites une "indication de déconnexion" ("disconnection indication" ou "tdisind"). Dans le second cas le site qui veut se déconnecter envoie au réseau une "requête de déconnexion" ("disconnection request" ou "tdisreq").

Figure 4: mécanisme de déconnexion

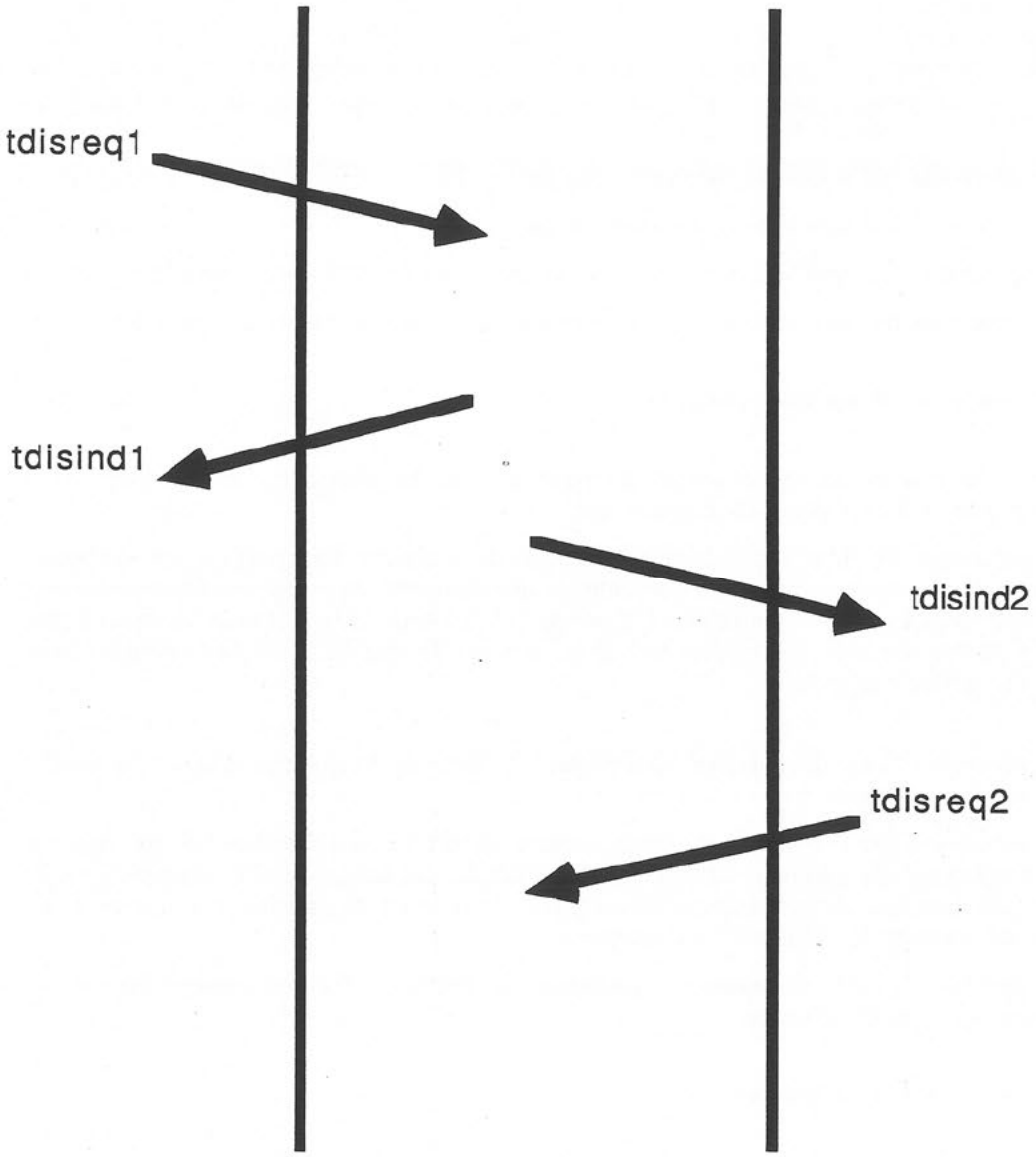


Figure 4: mécanisme de déconnexion

7. Modélisations en LOTOS du protocole

Les deux modélisations du protocole simplifié de transport, développées par Elie Najm, diffèrent par les niveau des couches ISO dont elles utilisent les fonctionnalités. La première modélisation, dite "de haut niveau", décrit le fonctionnement du protocole en utilisant les primitives de la 4-ième couche ("transport"). Cette description est faite suivant les principes de la modélisation "par contraintes" ("constraint oriented") développée par l'ISO et qui veut qu'on spécifie une communication entre deux sites par la donnée de 4 processus:

- un processus qui simule le comportement du site 1
- un processus qui simule le comportement du site 2
- un processus qui simule le service de communication allant du site 1 vers le site 2
- un processus qui simule le service de communication allant du site 2 vers le site 1

Figure 5: modélisation par contraintes

Dans le cas de la modélisation de haut niveau du protocole de transport on a effectivement 4 processus LOTOS définis comme suit:

- le processus "Ti": il modélise le comportement de la couche transport du site initiateur. Par exemple il exprime que l'initiateur doit émettre une demande de connexion ("tconreq") puis attendre une confirmation de connexion ("tconconf"), qu'à tout instant il peut se déconnecter ("tdisreq") ou être déconnecté ("tdisind"),... Précisons que Ti spécifie la totalité des utilisations correctes du service transport.

- le processus "Ta": de manière symétrique, il décrit le fonctionnement de la couche transport du site accepteur
- le processus "Tia": il simule le comportement du service uni-directionnel qui va de la couche transport du site initiateur à la couche transport du site accepteur. Par exemple lorsque Tia reçoit une demande de connexion ("tconreq") en provenance de l'initiateur il transmet une indication de connexion ("tconind") à l'accepteur
- le processus "Tai": de manière symétrique il décrit le fonctionnement du service uni-directionnel de sens contraire

Figure 6: modélisation de haut niveau

La seconde modélisation est dite "de bas niveau". Elle décrit le même protocole de transport que la modélisation de haut niveau mais elle le fait de manière différente. La description de haut niveau utilise les primitives du niveau transport (4-ième couche ISO) alors que la description de bas niveau est basée sur les primitives du niveau réseau (3-ième couche) et met en oeuvre des processus d'interfaçage entre les 3-ième et 4-ième couches. Cette modélisation comprend 10 processus:

- le processus "Ti", décrit ci-dessus
- le processus "Ta", décrit ci-dessus
- les processus "TNia" et "TNai", qui constituent des interfaces entre la couche transport et la couche transport du site initiateur
- les processus "NTia" et "NTai", qui jouent un rôle identique sur le site accepteur

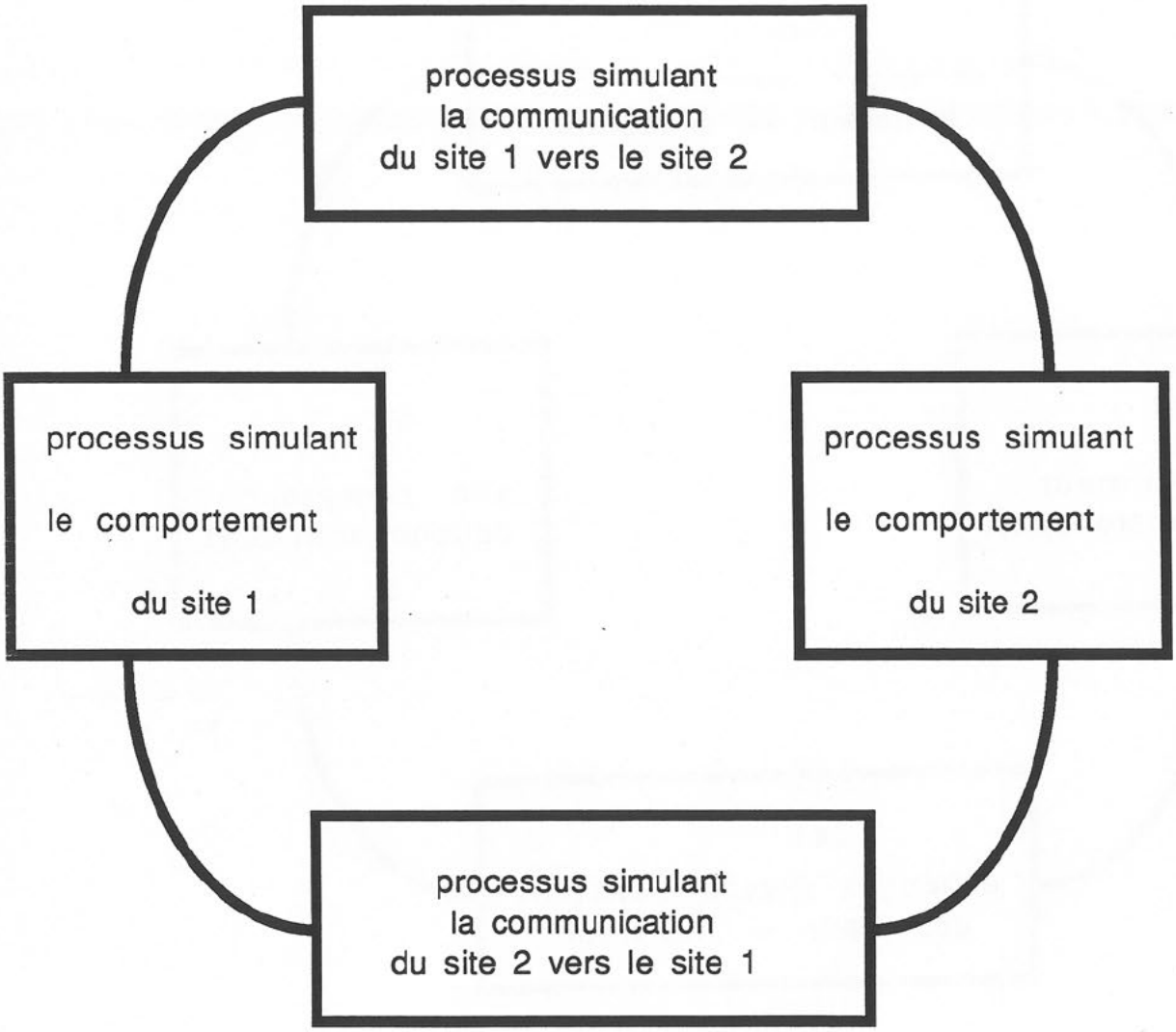


Figure 5: principes de la modélisation par contraintes

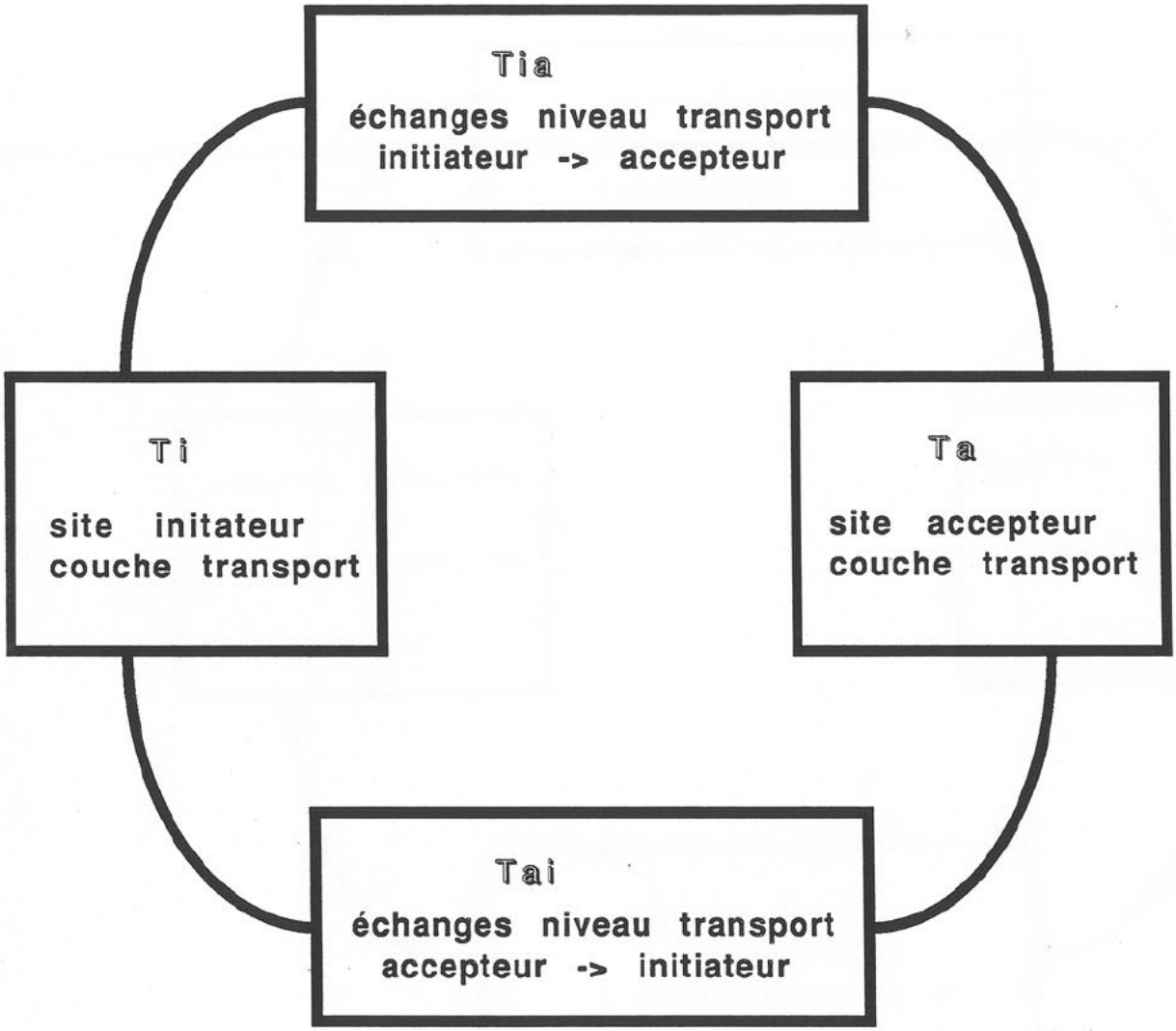


Figure 6: modélisation de haut niveau

- le processus "TNi" qui modélise le comportement de la couche réseau du site initiateur lorsqu'elle travaille pour le compte du protocole de transport
- le processus "NTa" qui modélise le comportement de la couche réseau du site accepteur lorsqu'elle travaille pour le compte du protocole de transport
- le protocole "Nia" qui simule le comportement du service uni-directionnel allant de la couche réseau du site initiateur à la couche réseau du site accepteur.
- le protocole "Nai" qui simule le comportement du service uni-directionnel fonctionnant en sens inverse

Figure 7: modélisation de bas niveau

Figure 8: comparaison des deux modélisations

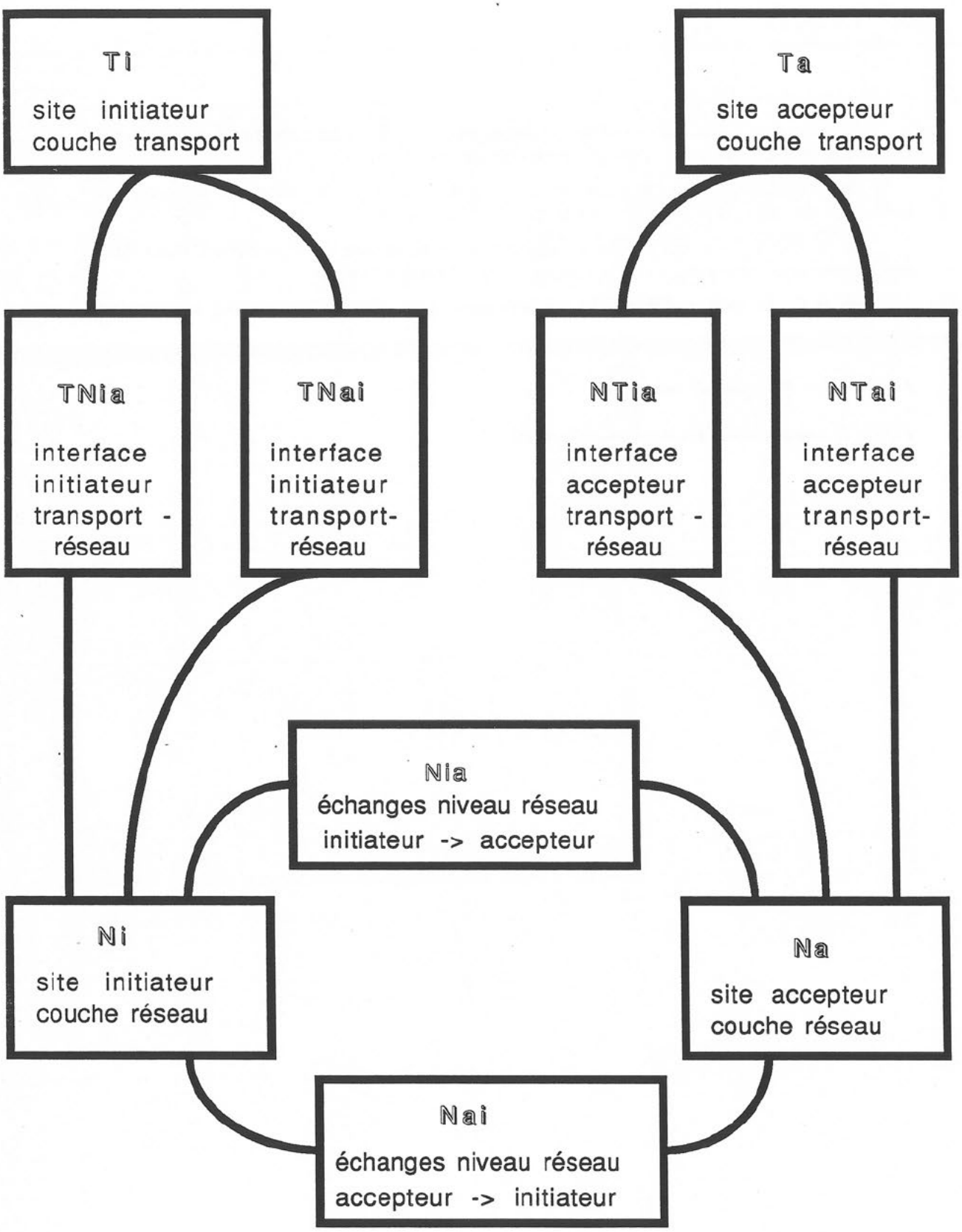
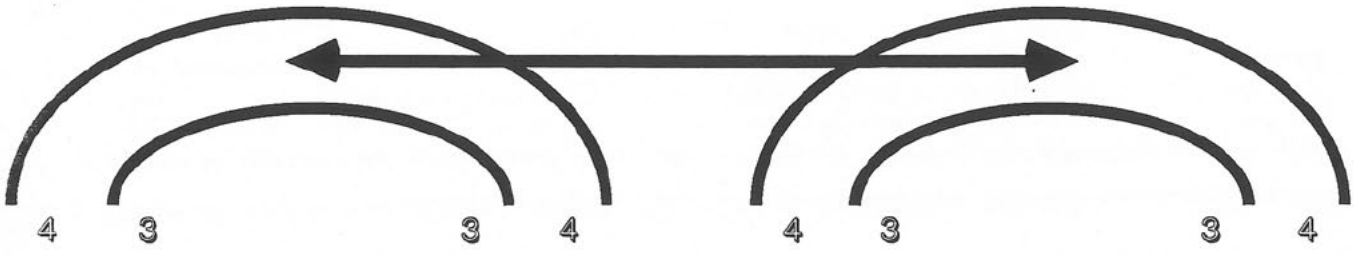
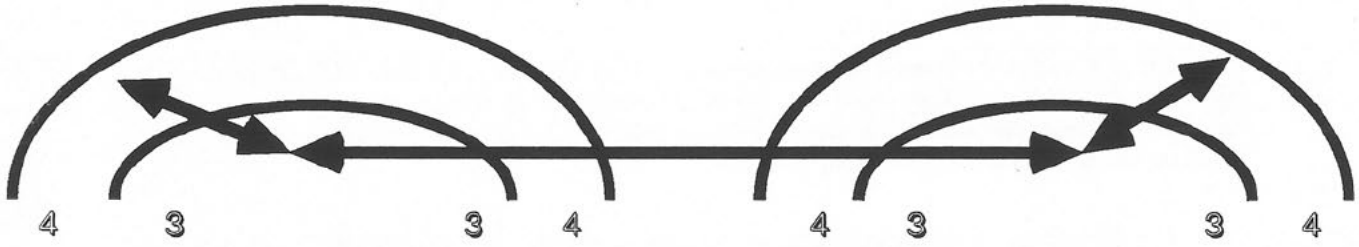


Figure 7: modélisation de bas niveau



site initiateur site accepteur

modélisation de haut niveau: communication au niveau transport



site initiateur site accepteur

modélisation de bas niveau: communication au niveau réseau

Figure 8: comparaison des deux modélisations

8. Traduction des programmes LOTOS en LDC

Pour vérifier à l'aide du système CESAR les deux programmes LOTOS décrivant le protocole, il a fallu les traduire dans le langage de description de CESAR (LDC). Précisons tout de suite que LDC ne constitue qu'un interface provisoire de CESAR et que les principes de CESAR peuvent s'appliquer à des langages autres que LDC, notamment ESTELLE. Toutefois ce problème de traduction est intéressant car il permet de comparer deux langages parallèles représentatifs de deux grandes approches: CSP (dont dérive LDC) et CCS (dont s'inspire LOTOS).

- LDC maintient une différence très nette entre les parties de programmes séquentielles ("task") et parallèles ("cotask"). Une "task" est un automate implémenté par une boucle non-déterministe "do...od". Une "cotask" est obtenue par composition parallèle de "tasks" ou de "cotasks". En revanche, LOTOS n'établit pas de distinction syntaxique de cette nature: la composition séquentielle et la composition parallèle sont deux opérateurs homogènes.

- Il faut donc isoler les sous-expressions séquentielles de LOTOS (celles qui n'utilisant que les opérateurs ";", "[]", ">>" et "[>") et les convertir en automates que l'on implémente comme des "tasks" à l'aide d'une variable d'état. Un générateur automatique de "tasks" LDC à partir du tableau des transitions d'un automate a été développé et a rendu de grands services.

- En LDC, tous les noms d'objets doivent être distincts. En revanche, LOTOS dispose de règles de visibilité par blocs pour les noms de variables et d'événements. Le générateur mentionné ci-dessus produit des noms uniques en préfixant les noms des objets par le nom de la "task" ou de la "cotask" dont ils font partie.

Les primitives de synchronisation et de communication de LOTOS sont beaucoup plus puissantes que celles de LDC et diffèrent sur beaucoup de points:

- En LOTOS il n'y a pas de "canaux": pour que plusieurs processus puissent se synchroniser sur une porte "p" il suffit qu'ils emploient le même nom "p" dans une demande de rendez-vous. Au contraire, en LDC, tous les noms de portes utilisés par un processus sont des paramètres formels que l'on doit instancier par un nom de canal lorsque ces processus sont composés en parallèle. Il y a deux types de canaux, ceux de type "port" qui permettant une communication par rendez-vous entre deux processus, l'un émettant et l'autre recevant, et ceux de type "broad" qui permettent le rendez-vous de N processus en mode "diffusion", l'un émettant et les (N-1) autres recevant.

- Il ne suffit pas de créer un canal ("port" ou "broad") pour chaque porte LOTOS et de relier, au moyen de ce canal, tous les processus qui se synchronisent sur cette porte. En effet LOTOS permet le rendez-vous par filtrage. Par exemple, si deux processus LOTOS proposent respectivement les rendez-vous "p!x" et "p!y" sur la porte "p", il n'y aura synchronisation que si x=y. LDC ne dispose pas d'un mécanisme aussi élaboré. La solution s'apparente à la résolution des surcharges d'opérateurs dans les langages de programmation. Dans l'exemple précédent la porte LOTOS "p" permet deux rendez-vous possibles, "p!x" et "p!y" et doit donc être scindée en deux portes LDC "p_x" et "p_y". Dans le cas du protocole, la porte LOTOS "ti", employée dans les contextes suivants:

ti ! tcon ! itoa
ti ! tcon ! atoi
ti ! tdis ! itoa
ti ! tdis ! atoi
ti ! tdata ! itoa
ti ! tdata ! atoi

a été décomposée en six portes LDC:

tconreq
tconconf
tdisreq1
tdisind1
tdatareq1
tdataind1

- En LOTOS le mécanisme de rendez-vous est symétrique: il n'y a pas de processus émetteur ni de processus récepteur. Cette caractéristique est séduisante lorsque l'on traite de la synchronisation "pure" (sans transmission de valeurs) car les deux processus qui se synchronisent sont traités de la même manière. En LDC, au contraire, les rendez-vous sont orientés même dans le cas de la synchronisation "pure". Quand on effectue la traduction on doit imposer un sens arbitraire aux rendez-vous LOTOS, choisir un processus émetteur et un (ou plusieurs) processus récepteurs.

- LOTOS et LDC permettent les rendez-vous à plus de 2 processus (facilité dont ne dispose pas CSP). Le rendez-vous de 3 processus LOTOS sur une porte "p" s'écrit ainsi:

P (p) |[p]| Q (p) |[p]| R (p)

En LDC on utilise un canal de type "broad" pour relier les trois processus. Dans le cas de la modélisation de bas niveau du protocole de transport, les portes suivantes permettent un rendez-vous à 3 processus:

ni ! cr ! itoa	(ndatareq1_cr)
na ! cr ! itoa	(ndataind2_cr)
ni ! cc ! atoi	(ndataind1_cc)
na ! cc ! atoi	(ndatareq2_cc)
ni ! dr ! atoi	(ndataind1_dr)
na ! dr ! atoi	(ndatareq2_dr)
ni ! ntdata ! itoa	(ndatareq1_ntdata)
na ! ntdata ! itoa	(ndataind2_ntdata)
ni ! ntdata ! atoi	(ndataind1_ntdata)
na ! ntdata ! atoi	(ndatareq2_ntdata)

- En LDC, le non-déterminisme prend place à l'intérieur des processus séquentiels, dans les boucles "do...od". Ceci est aussi possible en LOTOS grâce à l'opérateur "[]". La conversion est donc aisée. Mais LOTOS permet aussi d'introduire le non-déterminisme au niveau même de la composition parallèle et des mécanismes de synchronisation. En effet, on peut utiliser l'opérateur "|||" qui met en parallèle deux processus sans aucune synchronisation. Lorsque ces processus ont une porte "p" en commun:

P (p) ||| Q (p)

le processus résultant possède aussi une porte "p" qui représente soit celle de P soit celle de Q. La modélisation de bas niveau emploie des expressions de la forme

(P (p) ||| Q (p)) |[p]| R (p)

et que l'on peut transformer ainsi:

(P (p) ||| Q (p_bis)) |[p, p_bis]| R' (p, p_bis)

où "p_bis" est un nouveau nom de porte. Dans Q on remplace toutes les occurences de "p" par

"p_bis". R' est obtenue à partir de R en dédoublant toutes les transitions "p", c'est-à-dire en offrant la possibilité non-déterministe de faire la transition "p_bis" partout où l'on peut faire la transition "p". Les portes suivantes ont subi cette transformation:

```
np1 ! ndis ! itoa      (ndisreq1)
    renommé en ndisreq1_bis dans TNai
    dédouble dans Nia
np1 ! ndis ! atoi      (ndisind1)
    renommé en ndisind1_bis dans TNai
    dédoublé dans Nia
np2 ! ndis ! atoi      (ndisreq2)
    renommé en ndisreq2_bis dans NTia
    dédoublé en Nai
np2 ! ndis ! itoa      (ndisind2)
    renommé en ndisind2_bis dans NTia
    dédoublé dans Nai
```

CESAR ne s'applique qu'aux systèmes "fermés", c'est-à-dire aux systèmes qui n'ont pas de portes ouvertes sur l'extérieur: toutes les portes des processus doivent être attachés à des canaux. Au contraire, une spécification LOTOS peut être "ouverte", c'est à dire paramétrée par des portes non instanciées. Tel est le cas de la modélisation de bas niveau. Elle était composée de 3 processus comprenant chacun 4 sous-processus, suivant la description "par contraintes" de l'ISO:

```
TN = Ti // TNi // TNia // TNai
NN = Ni // Na // Nia // Nai
NT = Ta // NTa // NTai // NTia
```

On a supprimé les processus "Ni" et "Na" qui modélisent le comportement d'utilisateurs idéaux de la couche "réseau" (qui en exploiteraient toutes les fonctionnalités en respectant les règles d'utilisation). Les 10 processus restants ont été interconnectés. "TNi" et "NTa" sont venus tout naturellement prendre la place de "Ni" et "Na" puisqu'ils jouent le rôle d'utilisateurs de la couche réseau pour le compte du protocole de transport.

La suppression de "Ni" et "Na" a soulevé le problème suivant: la description de bas niveau contenait six portes dont chacune n'était présente que dans un seul processus ("Nia" ou "Nai"):

```
ni ! cr ! atoi     (ndataind1_cr)
na ! cr ! atoi     (ndatareq2_cr)
ni ! dr ! itoa     (ndatareq1_dr)
na ! dr ! itoa     (ndataind2_dr)
ni ! cc ! itoa     (ndatareq1_cc)
na ! cc ! itoa     (ndataind2_cc)
```

Cela s'explique par le fait que l'on a remplacé "Ni" et "Na" par "NTi" et "TNa" qui n'emploient qu'une partie des fonctionnalités du réseau; en particulier ces 6 transitions correspondent à des services non utilisés. Pour traduire cela en LDC, on a associé à chacun de ces 6 ports un canal dont l'autre extrémité n'est reliée à aucun port. LDC interprète cette situation en admettant qu'aucun rendez-vous n'est possible sur ces canaux, ce que l'on voulait exprimer.

Cette phase de traduction de LOTOS en LDC est intéressante car elle permet la comparaison fructueuse de deux façons de spécifier le parallélisme; toutefois elle n'a été faite que pour des raisons contingentes: la seule version de CESAR disponible fonctionnait avec LDC.

Une autre solution aurait été la traduction directe des programmes LOTOS en systèmes de transitions. Toutefois cette conversion fastidieuse aurait du être effectuée manuellement car on ne dispose pas encore d'un compilateur de programmes LOTOS en systèmes de transitions.

9. Spécifications logiques du protocole

Les modélisations de haut et de bas niveau, après avoir été traduites de LOTOS en LDC, ont été fournies au système CESAR qui a produit les systèmes de transitions correspondants. Le système de transitions de haut niveau comporte 47 états (4 processus, 3 pages de LDC) et celui de bas niveau en comprend 2549 (10 processus, 10 pages de LDC). Cette complexité rend très difficile sinon impossible la preuve "à la main". Encore ne s'agit-il que d'une modélisation simplifiée du protocole!

Les transitions que l'on veut manipuler dans les formules de logique temporelle, c'est-à-dire les transitions "observables" au niveau de la spécification, doivent être explicitement étiquetées dans le programme LDC.

Les formules expriment des propriétés sur les états mais il n'est pas possible, pour des raisons d'abstraction, de désigner nommément les états (par leurs numéros, par exemple) puisque l'utilisateur n'est pas supposé connaître le système de transitions. On ne peut accéder à un état qu'en donnant une propriété qui le caractérise. Les propriétés peuvent porter sur le fait qu'un état soit initial ou terminal, sur les valeurs des variables, sur la position par rapport à une transition étiquetée (avant ou après).

Il est classique de distinguer deux sortes de propriétés: les propriétés de sûreté et les propriétés de vivacité.

9.1. Propriétés de sûreté

Elles sont utilisées pour s'assurer que certaines situations anormales ne peuvent jamais se produire. Elles expriment des contraintes statiques ("invariants") sur les états du système.

De manière générale si la situation dangereuse est caractérisée par le prédicat ANORMAL (prédicat portant sur les valeurs des variables et/ou sur les transitions) on écrit:

init => not pot ANORMAL

ce qui signifie qu'il n'est pas possible d'atteindre, en partant des états initiaux, un état qui vérifie le prédicat ANORMAL, ou encore, de manière équivalente:

init => all not ANORMAL

que l'on peut abréger en:

[] not ANORMAL

et qui traduit que tous les états accessibles à partir des états initiaux vérifient le prédicat (not ANORMAL)

Remarque:

Le système CESAR supprime les états inaccessibles à partir de l'état initial. Par conséquent, les deux formules [] P et P sont équivalentes.

Les propriétés d'exclusion mutuelle sont un cas particulier important des propriétés invariantes. Soit une ressource partagée entre deux processus et deux prédicats ACCES_1 et ACCES_2 exprimant respectivement que les processus 1 et 2 accèdent à la ressource commune. La propriété d'exclusion mutuelle traduit qu'il n'est jamais possible d'avoir ces deux prédicats valides simultanément:

not (ACCES_1 and ACCES_2)

Dans le cas du protocole de transport, les propriétés de sûreté portent sur une utilisation conforme du service. Donnons quelques exemples:

- il n'est pas possible de transmettre des données avant d'avoir établi la connexion (et sans avoir été déconnecté entre temps, bien entendu)
- l'initiateur ne doit pas envoyer deux requêtes de connexion successives à moins d'avoir été déconnecté entre temps ou d'avoir reçu une confirmation de connexion.
- un site ne peut pas être déconnecté avant d'avoir émis une requête de déconnexion ou d'avoir reçu une indication de déconnexion

Dans le cas du protocole étudié on constate qu'il n'est pas besoin d'utiliser CESAR pour être certain que ces propriétés sont vérifiées. Ceci est une conséquence de la modélisation "par contraintes": les processus "Ti" et "Ta" placés aux deux bouts de la chaîne de communication expriment le comportement d'un utilisateur qui respecte les règles d'utilisation du protocole. Par construction "Ti" et "Ta" vérifient toutes les propriétés de sûreté. La nature du système et de la communication par rendez-vous fait que, si ces propriétés sont vraies pour "Ti" et "Ta", elles le seront pour la totalité du système.

On voit ici apparaître un défaut de la modélisation "par contraintes": elle ne décrit que les utilisations correctes et le système de télécommunications n'est pas placé dans un environnement hostile. Il n'est donc pas possible de savoir si le protocole est robuste, c'est-à-dire capable de résister à de mauvaises utilisations. Une solution consisterait à remplacer "Ti" et "Ta" par des processus "chaos" (au sens de CSP, c'est-à-dire des processus qui tentent d'effectuer toutes les transitions possibles dans n'importe quel ordre) et de voir ensuite si les propriétés de sûreté sont vérifiées.

9.2. Propriétés de vivacité

Elles complètent les propriétés de sûreté en permettant de s'assurer qu'un fonctionnement normal est possible. Il ne suffit pas de savoir que certaines évolutions critiques ne peuvent pas se produire (on la sait grâce aux propriétés de sûreté) il faut encore montrer que le comportement du système est, une fois les situations exceptionnelles écartées, conforme à ce qu'on en attend.

Parmi les propriétés de vivacité que l'on emploie fréquemment, on peut citer:

- **l'absence de terminaison** (aucun des états accessibles à partir de l'état initial n'est un état puits, c'est-à-dire sans successeur):

  not sink,

- **l'absence de blocage** d'une transition T (une transition devient bloquée à partir d'un certain moment lorsqu'il n'existe aucune évolution permettant d'exécuter cette transition)

  pot enable (T)

- **l'absence de famine** d'une transition T (une transition souffre de famine à partir d'un certain moment lorsqu'il existe une évolution permettant indéfiniment de ne pas exécuter cette transition)

  inev enable (T)

- **l'absence de famine avec équité** (on suppose que le non-déterministe est implémenté par un choix aléatoire équitable: une composante d'un choix ne pourra indéfiniment être négligée)

  fair enable (T)

Les propriétés ci-dessus sont générales, indépendantes de la sémantique du système considéré. Elles ont été fournies à CESAR (pour T décrivant l'ensemble de toutes les transitions étiquetées) pour les deux modélisations et déclarées valides. Dans le cas du protocole de transport des propriétés plus spécifiques ont été prouvées:

- **le comportement cyclique**: on a déjà montré l'absence de terminaison mais on peut exprimer une propriété plus forte qui dit qu'à chaque instant (et sous l'hypothèse d'équité) on retournera inévitablement dans l'état initial après un nombre fini de transitions:

  fair init

  Il faut noter que la formule:

  pot init

  exprime qu'il est toujours possible de revenir dans l'état initial.

- **propriétés de connexion**: on veut garantir le fonctionnement correct du mécanisme de connexion qui donne lieu à la succession des 4 événements suivants, observables au niveau de la couche transport:

  tconreq: requête de connexion émise par l'initiateur
  tconind: indication de connexion reçue par l'accepteur
  tconresp: réponse de connexion émise par l'accepteur
  tconconf: confirmation de connexion reçue par l'initiateur

Les propriétés que l'on veut prouver sont de la forme: après émission de "tconreq" il y a inévitablement réception de "tconind". En première approximation on peut les formuler ainsi:

after (tconreq) => inev after (tconind)
after (tconind) => inev after (tconresp)
after (tconresp) => inev after (tconconf)

Mais ces formules sont incorrectes puisque le processus de connexion peut être abandonné suite à une requête de déconnexion qui ramène le système dans son état initial (dénoté par le prédicat prédéfini "init"). On corrige donc les formules:

after (tconreq) => inev (after (tconind) or init)
after (tconind) => inev (after (tconresp) or init)
after (tconresp) => inev (after (tconconf) or init)

afin d'exprimer qu'après une action, on obtient inévitablement soit la réponse correspondante soit le retour à l'état initial de la totalité du système (c'est-à-dire l'ensemble des processus).

Il faut affiner ces formules en remplaçant l'opérateur "inev" par "fair":

after (tconreq) => fair (after (tconind) or init)

after (tconind) => fair (after (tconresp) or init)
after (tconresp) => fair (after (tconconf) or init)

En effet la définition de "inev" est telle que la formule:

PRECONDITION => inev POSTCONDITION

signifie: sur tous les chemins qui partent d'un état qui satisfait PRECONDITION il existe un état qui satisfait POSTCONDITION. Or ces chemins peuvent contenir des circuits "parasites" dûs au fait que le système de transitions sur lequel on travaillle est un graphe composé. En particulier il suffit qu'il existe un processus cyclique (une horloge par exemple) complètement indépendant du reste du système pour que de tels circuits se créent. En présence de tels circuits, CESAR évalue à faux une formule intuitivement vraie. Il est toujours possible de remplacer "inev" par "fair" ("fair" ne tenant pas compte des circuits, "parasites" ou non) mais d'autres solutions sont recherchées.

Une nouvelle difficulté surgit alors: ces formules sont vraies mais elles ne caractérisent pas un fonctionnement correct du système. En effet, on a vu que:

fair init

et d'autre part, l'opérateur "fair" est croissant par rapport à l'implication. On en déduit alors que, pour toute formule P:

si (fair init) est vraie alors (fair (P or init)) est vraie

On remédie à ce problème en conditionnant l'opérateur "fair":

after (tconreq) => fair [not enable (tconreq)] (after (tconind) or init)

after (tconind) => fair [not enable (tconind)] (after (tconresp) or init)

after (tconresp) => fair [not enable (tconresp)] (after (tconconf) or init)

ce qui signifie (pour la première) que sur tous les chemins équitables issus d'un état situé après une transition étiquetée "tconreq", et avant d'avoir rencontré un état à partir duquel on peut exécuter la transition "tconreq", il existe soit un état qui est placé après une transition étiquetée "tconind", soit un état initial. On s'assure par cette restriction, qu'il n'y a pas eu plusieurs transitions "tconreq" exécutées avant de d'avoir eu la réponse, c'est-à-dire "tconind".

- **propriétés de transfert**: chaque site peut émettre ou recevoir des données; les transitions correspondantes sont étiquetées "tdatareq_x" et "tdataind_x" où x décrit {1, 2}.

  Les propriétés à vérifier portent sur le bon acheminement des données. Il faut s'assurer que les données émises par un site sont effectivement reçues par l'autre site et que le signal ne se perd pas à travers la chaîne des processus (parce qu'un processus serait bloqué dans un état incapable de le recevoir, par exemple).

  Il s'agit de formules analogues à celles données pour la connexion:

  after (tdatareq_x) => fair [not enable (tdatareq_x)] (after (tdataind_y) or init)

  où x décrit {1, 2} et y décrit {1, 2} - {x}.

- déconnexion: on a vu que la communication pouvait être rompue soit par un des deux sites, soit par le réseau qui assure le transport. Les transitions correspondantes sont étiquetées "tdisreq_xy" et "tdisind_xy" où x et y prennent leurs valeurs dans {1, 2}. La question qui se pose est la suivante: les déconnexions sont-elles correctement traitées? Plus précisément, après une reqûete ou une indication de déconnexion, quelle que soit sa cause, est-on le système revient dans son état initial? Il s'agit d'une propriété forte puisque chacun des processus doit être réinitialisé. Pour s'en assurer on utilise les formules suivantes:

after (tdisreq_1x) => fair [not enable (tdisreq_11, tdisreq_12, tdisind_11, tdisind_12)] init
after (tdisind_1x) => fair [not enable (tdisreq_11, tdisreq_12, tdisind_11, tdisind_12)] init

after (tdisreq_2x) => fair [not enable (tdisreq_21, tdisreq_22, tdisind_21, tdisind_22)] init
after (tdisind_2x) => fair [not enable (tdisreq_21, tdisreq_22, tdisind_21, tdisind_22)] init
où x décrit {1, 2}

10. Résultats obtenus

Voici la liste des formules évaluées par le système CESAR et les résultats obtenus.

10.1. Formules valides pour les deux modélisations

Absence de terminaison

not sink

Comportement cyclique

fair init

Absence de famine (donc de blocage) pour toutes les transitions

fair enable (tconreq)
fair enable (tconind)
fair enable (tconresp)
fair enable (tconconf)

fair enable (tdisreq11)
fair enable (tdisreq12)
fair enable (tdisreq21)
fair enable (tdisreq22)
fair enable (tdisind11)
fair enable (tdisind12)
fair enable (tdisind21)
fair enable (tdisind22)

fair enable (tdatareq1)
fair enable (tdatareq2)
fair enable (tdataind1)
fair enable (tdataind2)

10.2. Formules valides uniquement pour la modélisation de haut niveau

Les formules suivantes sont vraies pour la modélisation de haut niveau et fausses pour celle de bas niveau.

Connexion

after (tconreq) => fair [not enable (tconreq)] (after (tconind) or init)
after (tconind) => fair [not enable (tconind)] (after (tconresp) or init)
after (tconresp) => fair [not enable (tconresp)] (after (tconconf) or init)

Déconnexion

after (tdisreq11) => fair [not enable (tdisreq_11, tdisreq_12, tdisind_11, tdisind_12)] init
after (tdisreq12) => fair [not enable (tdisreq_11, tdisreq_12, tdisind_11, tdisind_12)] init
after (tdisind11) => fair [not enable (tdisreq_11, tdisreq_12, tdisind_11, tdisind_12)] init
after (tdisind12) => fair [not enable (tdisreq_11, tdisreq_12, tdisind_11, tdisind_12)] init
after (tdisreq21) => fair [not enable (tdisreq_21, tdisreq_22, tdisind_21, tdisind_22)] init
after (tdisreq22) => fair [not enable (tdisreq_21, tdisreq_22, tdisind_21, tdisind_22)] init
after (tdisind21) => fair [not enable (tdisreq_21, tdisreq_22, tdisind_21, tdisind_22)] init
after (tdisind22) => fair [not enable (tdisreq_21, tdisreq_22, tdisind_21, tdisind_22)] init

Transmission

after (tdatareq_1) => fair [not enable (tdatareq_1)] (after (tdataind_2) or init)
after (tdatareq_2) => fair [not enable (tdatareq_2)] (after (tdataind_1) or init)

10.3. Analyse des résultats

On peut donc conclure que le protocole de bas niveau est incorrect, car il ne vérifie pas les spécifications de la connexion, de la déconnexion et du transfert de données. En revanche, le protocole de haut niveau - qui sert de spécification - est correct.

Ces résultats sont confirmés par une étude d'Amélia Soriano qui a soumis cette modélisation au système VENUS qu'elle a développé [S 86]. Les automates correspondants ont été construits, projetés (on n'observe que certaines transitions) et minimisés, en appliquant les principes de l'équivalence observationnelle de Milner [M 80].

Les deux automates ne sont pas équivalents observationnellement. En effet les formes normales, après réduction et minimisation, n'ont pas le même nombre d'états et on a pu mettre en évidence des chemins pour lesquels les formules sont fausses. Précisons ici que la logique CTL n'est pas compatible avec l'équivalence observationnelle: deux programmes vérifiant la même formule logique peuvent ne pas être observationnellement équivalents et deux programmes observationnellement équivalents peuvent ne pas vérifier les mêmes formules logiques.

11. Conclusion

Au terme de ce travail qui se situe au confluent de trois thèmes de recherche: les langages de spécification du parallélisme, les protocoles de télécommunication et la vérification automatique des systèmes réactifs, plusieurs conclusions peuvent être formulées.

A l'usage, le système CESAR se révèle être un logiciel utile et réaliste:

- utile parce que la complexité combinatoire des systèmes réactifs ne peut être maîtrisée sans l'aide d'outils automatiques; la conception et le développement de tels moyens de vérification connaît actuellement un essor considérable dans la recherche et l'industrie.

  A l'aide de CESAR on a ainsi pu montrer que la modélisation de bas niveau du protocole de transport ne respectait pas ses spécifications. Il faut signaler que l'on n'a pas utilisé toutes les possibilités de CESAR: en particulier la modélisation du protocole n'explicitait pas les transferts de valeurs.

- réaliste ensuite parce qu'il s'appuie sur une logique décidable et que son implémentation utilise des algorithmes et non pas des techniques plus complexes (inférence, résolution, réécriture). On n'a donc pas à espérer l'arrivée de nouvelles machines à architecture spécialisée pour avoir des performances convenables. On trouvera dans l'annexe C une évaluation des temps d'exécution nécessaires.

  Par ailleurs il faut noter que la nouvelle version XESAR est encore plus efficace, tant par la taille des automates acceptés que par la rapidité d'exécution. La nouvelle version accepte au moins 30000 états (contre 3000 pour l'ancienne) et utilise des algorithmes linéaires (alors que l'ancienne version employait des techniques itératives dont la complexité était quadratique ou cubique par rapport au nombre d'états). Enfin il faut signaler qu'il sera possible de produire un graphe réduit en vue de l'évaluation d'une formule donnée, ce qui permettra de traiter les systèmes ayant plus de 30000 états.

L'utilisation intensive de l'ancienne version de CESAR a permis la détection de plusieurs erreurs résiduelles qui ont été corrigées et qui seront absentes de la nouvelle version. Par ailleurs l'interface utilisateur a été considérablement amélioré (la version XESAR propose un système de menus et une interface graphique évoluée, avec fenêtres et souris, est en cours de réalisation).

En ce qui concerne les logiques, leur emploi dans les spécifications présente des avantages décisifs:

- abstraction: indépendance vis-à-vis de toute implémentation, ce qui élimine les risques de sur-spécification toujours présents lorsqu'on choisit une implémentation particulière comme spécification
- expression de propriétés globales: exclusion mutuelle, absence de blocage et/ou de famine, propriétés de réponse à une action, ...
- décidabilité: CESAR emploie des logiques décidables. On dispose ainsi d'un démonstrateur automatique (très rapide) qui permet de décider si une formule est un théorème (un théorème est une formule valide quelque soit le système de transitions)

La nouvelle version XESAR utilise une logique plus puissante que CTL qui remédie aux insuffisances de cette dernière. En particulier on dispose d'opérateurs temporels par rapport au passé (alors que les opérateurs temporels de CTL ne caractérisent que les évolutions futures du système) qui permettent d'exprimer des propriétés nouvelles: existence d'états inaccessibles depuis l'état initial ("il

est possible, en remontant dans le passé, d'arriver à l'état initial"), existence d'une cause unique pour une situation donnée ("il est inévitable, en remontant dans le passé, d'atteindre la cause").

Tout en étant plus simples et plus lisibles que d'autres logiques temporelles [P 86], les logiques de type CTL restent d'un emploi délicat. Une phase d'apprentissage est nécessaire pour maîtriser le sens des formules que l'on écrit. C'est pourquoi un environnement permettant d'assister et de contrôler la démarche l'utiliseur pendant la phase de spécification semble indispensable. Il devrait offrir les fonctionnalités suivantes:

Remarque:

Par la suite, on note F la formule de logique temporelle dont l'évaluation a été demandée par l'utilisateur. On analyse F pour détecter diverses situations où F n'a pas les propriétés d'une "bonne spécification"

- on signale les sous-expressions G de F telles que G soit valide (G est satisfaite par tous les états) ou (not G) soit valide (G n'est satisfaite par aucun état).
- si F est valide on vérifie (et on signale, le cas échéant) si F est un théorème (on utilise pour celà le démonstrateur automatique)
- lorsque (not F) est valide on vérifie (et on signale, le cas échéant) si (not F) est un théorème
- les deux points précédents s'appliquent à toutes les sous-formules G de F telles que G ou (not G) sont valides
- lorsque F est valide, on recherche s'il existe une formule G telle que: G est valide, G implique F (autrement dit, si G valide alors F valide, à distinguer de (G => F) valide) et G est plus simple que F (G a moins ou autant d'opérateurs que F). Autrement dit F n'est valide que comme conséquence d'une autre formule G qui traduit une propriété plus forte
- lorsque (not F) est valide, on recherche s'il existe une formule G telle que: (not G) est valide, (not G) implique (not F) et G est plus simple que F
- lorsque F n'est pas valide, on recherche s'il existe une formule G telle que: F implique G et G est plus simple que F. Si G n'est pas valide, cela suffit à expliquer pourquoi F n'est pas valide. Si G est valide, on a trouvé une propriété valide plus faible que F

Remarque:

Un prototype a été développé, qui permet de déterminer un ensemble de formules G qui impliquent F (resp. impliquées par F) et plus simples que F.

Enfin, la réponse binaire ("vrai" ou "faux") fournie par CESAR est en général insuffisante pour la mise au point. Lorsque le système a déclaré qu'une formule était fausse, il doit fournir un contre-exemple qui justifie sa réponse car les recherches manuelles sont extrêmement fastidieuses. Les contre-exemples ne doivent pas se résumer à l'ensemble des états qui ne vérifient pas la formule mais doivent être donnés sous forme de chemins étiquetés par des transitions, dans le cas des opérateurs temporels.

Dans le futur, il est prévu d'adapter XESAR à LOTOS en déterminant un sous-langage de LOTOS (comprenant les primitives d'expression du parallélisme de LOTOS et un sous ensemble des types de données ACT ONE) et en concevant un simulateur qui transforme les programmes LOTOS en systèmes de transitions, dans le format interne de XESAR.

Références

A propos de la spécification et de la vérification des systèmes

[M 80]
R. MILNER
A Calculus of Communicating Systems
Springer-Verlag, Berlin, 1980

[QS 83]
J.P. QUEILLE, J. SIFAKIS
Fairness and related properties in transition systems
A temporal logic to deal with fairness
Acta Informatica 19 (1983) 195-220

[BHR 84]
S.D. BROOKES, C.A.R. HOARE, A.W. ROSCOE
A theory of Communicating Sequential Processes
JACM 31, 3 (1985) 560-599

[HM 85]
M. HENESSY, R. MILNER
Algebraic Laws for nondeterminism and concurrency
JACM 32, 1 (1985) 137-161

[BK 85]
J.A. BERGSTRA, J.W. KLOP
Algebra of communicating processes with abstraction
TCS 37, 1 (1985)

[DR 86]
W.P. DE ROEVER
Questions to Robin Milner - a responder's commentary
IFIP Information Processing, 1986, North Holland

[P 86]
A.PNUELI
Specification and development of reactive systems
IFIP Information Processing, 1986, North Holland

[S 86]
J. SIFAKIS
A response to A. PNUELI
Specification and development of reactive systems
IFIP Information Processing, 1986, North Holland

A propos du langage LOTOS et de sa définition formelle

[L 84]
ISO
Information Processing Systems
Open Systems Interconnection
LOTOS: a formal description technique
ISO/TC97/WG16-1 - ad hoc group FDT/ subgroup C

[B 85]

E. BRINKSMA
A tutorial on LOTOS
note technique
Université technologique de Twente

[EFH 83]
H. EHRIG, W.FEY, H. HANSEN
ACT ONE: an algebraic specification language with two levels of semantics
Bericht-Nr 83-03
Technical University of Berlin

[NB 86]
E. NAJM, J. BARRE
A proposal for a simplified presentation of the dynamic semantics of LOTOS
note technique
ADI, INRIA

A propos du système CESAR

[EH 82]
E.A. EMERSON, J.Y. HALPERN
Decision procedure and expressiveness
in the temporal logic of branching time
14th symposium on theory of computing (1982)

[Q 82]
J.P. QUEILLE
Le système CESAR: description, spécification et analyse des
applications réparties
Thèse de Docteur-Ingénieur - Université de Grenoble

[S 83]
J.P. SCHWARTZ
QUASAR: une réalisation du système CESAR
Thèse de Docteur-Ingénieur - Université de Grenoble

[FSS 84]
J.C. FERNANDEZ, J.P. SCHWARTZ, J. SIFAKIS
Description, specification and analysis in CESAR (4 examples)
RR 435 LGI-IMAG

[FRV 85]
J.C. FERNANDEZ, J.L. RICHIER, J. VOIRON
Verification of protocol specifications using the CESAR system
Proc. of the IFIP WG 6.1 fith international workshop on
protocol specification, testing and verification
Toulouse-Moissac, jun 1985, North-Holland

A propos du système VENUS

[S 86]
A. SORIANO
VENUS: un outil d'aide à la vérification des systèmes communicants
Thèse de 3ème cycle, Université de Grenoble

Annexe A: description en LOTOS du protocole de transport

Définition des notations et des sortes employées

Par la suite, les lettres "N", "T", "i" et "a" seront fréquemment employées dans les identificateurs. Leur signification est la suivante:

- "T" : transport
- "N" : network (réseau)
- "i" : initiateur
- "a" : accepteur

Les programmes LOTOS qui suivent utilisent deux types énumérés:

- ndata_sort = { cr, cc, dr, ntdata }
- form_to_sort = { itoa, atoi }

On introduit un opérateur de renommage, qui a la syntaxe suivante:

B [y\x]

où B est une expression de comportement et x, y des noms d'événements. Le résultat est l'expression B dans laquelle les occurences de x ont été remplacées par y.

Spécification du niveau transport

```
specification TT [ti, ta] :=

(Ti [ti] ||| Ta [ta])
|[ti, ta]|
((Tia[ti, ta, sync] |[sync]| Tai [ti, ta, sync]) \ sync)

where

process Ti [ti] :=
ti ! tcon ! itoa;
  ((ti ! tcon ! atoi; Tdt_trans_t [ti])
   [>
   (ti ! tdis ? x : from_to_sort; Ti [ti]))
endproc

process Ta [ta] :=
ta ! tcon ! itoa;
  ((ta ! tcon ! atoi; Tdt_trans_t [ta])
   [>
   (ti ! tdis ? x : from_to_sort; Ta [ta]))
endproc

process Tdt_trans_t [t] :=
t ! tdata ? x : from_to_sort; Tdt_trans_t [t]
endproc

process Tia [ti, ta, sync] :=
ti ! tcon ! itoa;
  ((ti ! tdis ? x : from_to_sort; Tia [ti, ta, sync])
```

```
  []
  (ta ! tcon ! itoa;
    (Tdt_trans_ia [ti, ta]
     [>
     (ti ! tdis ? x: from_to_sort; sync ! eoc; Tia [ ti, ta, sync])))
endproc

process Tai [ti, ta, sync] :=
(ta ! tcon! atoi;
  ((Tdt_trans_ai [ti, ta] ||| (ti ! tcon ! atoi; stop)))
[>
(ta ! tdis ? x : from_to_sort; sync ! eoc; Tai [ti, ta, sync])
endproc

process Tdt_trans_ia [ti, ta] :=
ti ! tdata ! itoa; ta ! tdata ! itoa; Tdt_trans_ia [ti, ta]
endproc

process Tdt_trans_ai [ti, ta] :=
ta ! tdata ! atoi; ti ! tdata ! atoi; Tdt_trans_ai [ti, ta]
endproc

endspec
```

Spécification du niveau réseau

```
specification NN [ni, na] :=
(Ni [ni] ||| Na [na])
|[ni, na]|
((Nia [ni, na, sync] |[sync]| Nai [ni, na, sync]) \ sync)

where

process Ni [ni] :=
ni ! ncon ! itoa;
  ((ni ! ncon ! atoi; Ndt_trans_n [ni]
   [>
   (ni ! ndis ? x : from_to_sort; Ni [ni]))
endproc

process Ndt_trans_n [ni] :=
n ? y : ndata_sort ? x : from_to_sort; Ndt_trans_n [ni]
endproc

process Na [na] :=
na ! ncon ! itoa;
  ((na ! ncon ! atoi; Ndt_trans_n [na])
   [>
   (na ! ndis ? x : from_to_sort; Na [na]))
endproc

process Nai [ni, na, sync] :=
(na ! ncon ! atoi;
  (Ndt_trans_ai [ni, na] ||| (ni ! ncon ! atoi; stop)))
```

```
[>
(na ! ndis ? x : from_to_sort; sync ! eoc; Nai [ni, na, sync])
endproc

process Ndt_trans_ai [ni, na] :=
na ? y : ndata_sort ! atoi; ni ! y ! atoi; Ndt_trans_ai [ni, na]
endproc

process Nia [ni, na, sync] :=
ni ! ncon ! itoa;
  ((ni ! ndis ? x : form_to_sort; Nia [ni, na, sync])
   []
   (na ! ncon ! itoa;
     (Ndt_trans_ia [ni, na]
     [>
     (ni ! ndis ? x : from_to_sort; sync ! eoc; Nia [ni, na, sync]))))
endproc

process Ndt_trans_ia [ni, na] :=
ni ? y : ndata_sort ! itoa; na ! y ! itoa; Ndt_trans_ia [ni, na]
endproc

endspec
```

Spécification de l'entité transport/réseau pour l'initiateur

```
specification TN [ti, ni] :=
((Ti [ti] ||| TNi [np, ni])
|[np, ti]|
((TNia [ti, np, sync] [sync]|| TNai [ti, np, sync]) \ [sync]))
[ni\np]

where

process TNi [np, ni] :=
ni ! ncon ! itoa;
  ((ni ! ndis ? x : from_to_sort; TNi [np, ni])
   []
   (ni ! ncon ! atoi;
     ((ni ! ndis ? x : from_to_sort; TNi [np, ni])
      []
      TNPi [np, ni])))
endproc

process TNPi [np, ni] :=
np ! cr ! itoa;
  (((np ! dr ! atoi; TNPi [np])
   []
   (np ! cc ! atoi; TNdt_trans_i [np]))
  [>
    (np ! ndis ? x : from_to_sort; TNi [np, ni]))
endproc

process TNdt_trans_i [np] :=
```

```
np ! ntdata ? x : from_to_sort; TNdt_trans_i [np]
endproc

process TNai [ti, np, sync] :=
((np ! cc ! atoi;
  (TNdt_trans_ai [ti, np] ||| (ti ! tcon ! atoi; stop)))
 []
 (np ! dr ! atoi; sync ! eoc; TYNai [ti, np, sync]))
[>
 (np ! ndis ? x : from_to_sort; sync ! eoc; TNai [ti, np, sync]))
endproc

process TNdt_trans_ai [ti, np] :=
np ! ntdata ! atoi; ti ! tdata ! atoi; TNdt_trans_ai [ti, np]
endproc

process TNia [ti, np, sync] :=
ti ! tcon ! itoa;
  ((ti ! tdis ? x : from_to_sort; TNia [ti, np, sync])
   []
   (np ! cr ! itoa;
     (TNdt_trans_ia [ti, np]
      [>
      (ti ! tdis ? x : from_to_sort; sync ! eoc; TNia [ti, np, sync])))))
endproc

process TNdt_trans_ia [ti, np] :=
ti ! tdata ! itoa; np ! ntdata ! itoa; TNdt_trans_ia [ti, np]
endproc
```

Spécification de l'entité transport/réseau pour l'accepteur

```
specification NT [ ta, na] :=
((Ta [ta] ||| NTa [np, na])
|[np, ta]|
((NTia [ta, np, sync] |[sync]| NTai [ta, np, sync])\[sync]))
[na\np]

where

process NTa [np, na] :=
na ! ncon ! itoa;
  ((na ! ndis ? x : from_to_sort; NTa [np, na])
   []
  (na ! ncon ! atoi;
    ((na ! ndis ? x : from_to_sort; NTa [np, na])
     []
     NTPa [np, na])))
endproc

process NTPa [np, na] :=
np ! cr ! itoa;
  (((np ! dr ! atoi; NTPa [np])
    []
```

```
    (np ! cc ! atoi; NTdt_trans_a [np]))
   [>
   (np ! ndis ? x : from_to_sort; NTa [np, na]))
endproc

process NTdt_trans_a [np] :=
np ! ntdata ? x : from_to_sort; NTdt_trans_a [np]
endproc

process NTai [ta, np, sync] :=
(ta ! tcon ! atoi;
  (NTdt_trans_ai [ta, np] ||| (np ! cc ! atoi; stop)))
[>
(ta ! tdis ? x : from_to_sort; sync ! eoc; NTai [ta, np, sync])
endproc

process NTdt_trans_ai [ta, np] :=
ta ! tdata ! atoi; np ! ntdata ! atoi; NTdt_trans_ai [ta, np]
endproc

process NTia [ta, np, sync] :=
np ! ct ! itoa;
((np ! dr ! atoi; NTia [ta, np, sync])
 []
 (np ! ndis ? x : from_to_sort; NTia [ta, np, sync]))
 []
  (ta ! tcon ! itoa;
    (NT_dt_trans_ia [ta, np]
     [>
     (np ! ndis ? x : from_to_sort; sync ! eoc; NTia [ta, np, sync]))))
endproc

process NT_dt_trans_ia [ta, np] :=
np ! ntdata ! itoa; ta ! tdata ! itoa; NT_dt_trans_ia [ta, np]
endproc

endspec
```

Annexe B: description en CESAR du protocole de transport

Introduction

Donnons tout d'abord la table des conversions entre événements structurés LOTOS et canaux de CESAR:

ti ! tcon ! itoa	tconreq
ta ! tcon ! itoa	tconind
ta ! tcon ! atoi	tconresp
ti ! tcon ! atoi	tconconf
ti ! tdis ! itoa	tdisreq1
ti ! tdis ! atoi	tdisind1
ta ! tdis ! atoi	tdisreq2
ta ! tdis ! itoa	tdisind2
ti ! tdata ! itoa	tdatareq1
ti ! tdata ! atoi	tdataind1
ta ! tdata ! atoi	tdatareq2
ta ! tdata ! itoa	tdataind2
ni ! ncon ! itoa	nconreq
na ! ncon ! itoa	nconind
na ! ncon ! atoi	nconresp
ni ! ncon ! atoi	nconconf
ni ! ndis ! itoa	ndisreq1
ni ! ndis ! atoi	ndisind1
na ! ndis ! atoi	ndisreq2
na ! ndis ! itoa	ndisind2
ni ! cr ! itoa	ndatareq1_cr
ni ! ntdata ! itoa	ndatareq1_ntdata
ni ! cc ! itoa	ndatareq1_cc (*)
ni ! dr ! itoa	ndatareq1_dr (*)
ni ! cr ! atoi	ndataind1_cr (*)
ni ! ntdata ! atoi	ndataind1_ntdata
ni ! cc ! atoi	ndataind1_cc
ni ! dr ! atoi	ndataind1_dr
na ! cr ! atoi	ndatareq2_cr (*)
na ! ntdata ! atoi	ndatareq2_ntdata
na ! cc ! atoi	ndatareq2_cc
na ! dr ! atoi	ndatareq2_dr
na ! cr ! itoa	ndataind2_cr
na ! ntdata ! itoa	ndataind2_ntdata
na ! cc ! itoa	ndataind2_cc (*)
na ! dr ! itoa	ndataind2_dr (*)

(*) ports connectés à un canal dont l'autre extrémité n'est pas reliée

Spécification de haut niveau

```
cotask HIGH;

task Ti;
input
    tdisind1_Ti,
    tconconf_Ti,
    tdataind1_Ti;
output
    tconreq_Ti,
    tdisreq1_Ti,
    tdatareq1_Ti;
declare
    STATE_Ti : 0..2;
init
    STATE_Ti := 0;
do
{ TCONREQ      }   STATE_Ti = 0 : !tconreq_Ti, STATE_Ti := 1 |
{ TDISREQ11    }   STATE_Ti = 1 : !tdisreq1_Ti, STATE_Ti := 0 |
{ TDISIND11    }   STATE_Ti = 1 : ?tdisind1_Ti, STATE_Ti := 0 |
{ TCONCONF     }   STATE_Ti = 1 : ?tconconf_Ti, STATE_Ti := 2 |
{ TDISREQ12    }   STATE_Ti = 2 : !tdisreq1_Ti, STATE_Ti := 0 |
{ TDISIND12    }   STATE_Ti = 2 : ?tdisind1_Ti, STATE_Ti := 0 |
{ TDATAREQ1    }   STATE_Ti = 2 : !tdatareq1_Ti, STATE_Ti := 2 |
{ TDATAIND1    }   STATE_Ti = 2 : ?tdataind1_Ti, STATE_Ti := 2.
od;

task Ta;
input
    tconind_Ta,
    tdisind2_Ta,
    tdataind2_Ta;
output
    tdisreq2_Ta,
    tconresp_Ta,
    tdatareq2_Ta;
declare
    STATE_Ta : 0..2;
init
    STATE_Ta := 0;
do
{ TCONIND      }   STATE_Ta = 0 : ?tconind_Ta, STATE_Ta := 1 |
{ TDISREQ21    }   STATE_Ta = 1 : !tdisreq2_Ta, STATE_Ta := 0 |
{ TDISIND21    }   STATE_Ta = 1 : ?tdisind2_Ta, STATE_Ta := 0 |
{ TCONRESP     }   STATE_Ta = 1 : !tconresp_Ta, STATE_Ta := 2 |
{ TDISREQ22    }   STATE_Ta = 2 : !tdisreq2_Ta, STATE_Ta := 0 |
{ TDISIND22    }   STATE_Ta = 2 : ?tdisind2_Ta, STATE_Ta := 0 |
{ TDATAREQ2    }   STATE_Ta = 2 : !tdatareq2_Ta, STATE_Ta := 2 |
{ TDATAIND2    }   STATE_Ta = 2 : ?tdataind2_Ta, STATE_Ta := 2
od;

task Tia;
input
    tconreq_Tia,
```

```
    tdisreq1_Tia,
    tdatareq1_Tia;
output
    tdisind1_Tia,
    tconind_Tia,
    tdataind2_Tia,
    tsync_Tia;
declare
    STATE_Tia : 0..4;
init
    STATE_Tia := 0;
do
    STATE_Tia = 0 : ?tconreq_Tia, STATE_Tia := 1 |
    STATE_Tia = 1 : ?tdisreq1_Tia, STATE_Tia := 0 |
    STATE_Tia = 1 : !tdisind1_Tia, STATE_Tia := 0 |
    STATE_Tia = 1 : !tconind_Tia, STATE_Tia := 2 |
    STATE_Tia = 2 : ?tdatareq1_Tia, STATE_Tia := 3 |
    STATE_Tia = 2 : ?tdisreq1_Tia, STATE_Tia := 4 |
    STATE_Tia = 2 : !tdisind1_Tia, STATE_Tia := 4 |
    STATE_Tia = 3 : !tdataind2_Tia, STATE_Tia := 2 |
    STATE_Tia = 3 : ?tdisreq1_Tia, STATE_Tia := 4 |
    STATE_Tia = 3 : !tdisind1_Tia, STATE_Tia := 4 |
    STATE_Tia = 4 : !tsync_Tia, STATE_Tia := 0
od;

task Tai;
input
    tconresp_Tai,
    tdisreq2_Tai,
    tdatareq2_Tai,
    tsync_Tai;
output
    tdisind2_Tai,
    tconconf_Tai,
    tdataind1_Tai;
declare
    STATE_Tai : 0..5;
init
    STATE_Tai := 0;
do
    STATE_Tai = 0 : ?tconresp_Tai, STATE_Tai := 1 |
    STATE_Tai = 0 : ?tdisreq2_Tai, STATE_Tai := 5 |
    STATE_Tai = 0 : !tdisind2_Tai, STATE_Tai := 5 |
    STATE_Tai = 1 : !tconconf_Tai, STATE_Tai := 2 |
    STATE_Tai = 1 : ?tdatareq2_Tai, STATE_Tai := 3 |
    STATE_Tai = 1 : ?tdisreq2_Tai, STATE_Tai := 5 |
    STATE_Tai = 1 : !tdisind2_Tai, STATE_Tai := 5 |
    STATE_Tai = 2 : ?tdatareq2_Tai, STATE_Tai := 4 |
    STATE_Tai = 2 : ?tdisreq2_Tai, STATE_Tai := 5 |
    STATE_Tai = 2 : !tdisind2_Tai, STATE_Tai := 5 |
    STATE_Tai = 3 : !tdataind1_Tai, STATE_Tai := 1 |
    STATE_Tai = 3 : !tconconf_Tai, STATE_Tai := 4 |
    STATE_Tai = 3 : ?tdisreq2_Tai, STATE_Tai := 5 |
    STATE_Tai = 3 : !tdisind2_Tai, STATE_Tai := 5 |
    STATE_Tai = 4 : !tdataind1_Tai, STATE_Tai := 2 |
```

```
    STATE_Tai = 4 : ?tdisreq2_Tai, STATE_Tai := 5
    STATE_Tai = 4 : !tdisind2_Tai, STATE_Tai := 5
    STATE_Tai = 5 : ?tsync_Tai, STATE_Tai := 0
od;

port
    HIGH_tconreq, HIGH_tconind, HIGH_tconresp, HIGH_tconconf,
    HIGH_tdisreq1, HIGH_tdisind1, HIGH_tdisreq2, HIGH_tdisind2,
    HIGH_tdatareq1, HIGH_tdataind1, HIGH_tdatareq2, HIGH_tdataind2,
    HIGH_tsync;

body
    Ti (HIGH_tdisind1,
        HIGH_tconconf,
        HIGH_tdataind1,
        HIGH_tconreq,
        HIGH_tdisreq1,
        HIGH_tdatareq1) //

    Tia (HIGH_tconreq,
         HIGH_tdisreq1,
         HIGH_tdatareq1,
         HIGH_tdisind1,
         HIGH_tconind,
         HIGH_tdataind2,
         HIGH_tsync) //

    Tai (HIGH_tconresp,
         HIGH_tdisreq2,
         HIGH_tdatareq2,
         HIGH_tsync,
         HIGH_tdisind2,
         HIGH_tconconf,
         HIGH_tdataind1) //

    Ta (HIGH_tconind,
        HIGH_tdisind2,
        HIGH_tdataind2,
        HIGH_tdisreq2,
        HIGH_tconresp,
        HIGH_tdatareq2).
```

Spécification de bas niveau

```
cotask LOW;

task Ti;
input
    tdisind1_Ti,
    tconconf_Ti,
    tdataind1_Ti;
output
    tconreq_Ti,
    tdisreq1_Ti,
    tdatareq1_Ti;
```

```
declare
    STATE_Ti : 0..2;
init
    STATE_Ti := 0;
do
{ TCONREQ       }   STATE_Ti = 0 : !tconreq_Ti, STATE_Ti := 1 |
{ TDISREQ11     }   STATE_Ti = 1 : !tdisreq1_Ti, STATE_Ti := 0 |
{ TDISIND11     }   STATE_Ti = 1 : ?tdisind1_Ti, STATE_Ti := 0 |
{ TCONCONF      }   STATE_Ti = 1 : ?tconconf_Ti, STATE_Ti := 2 |
{ TDISREQ12     }   STATE_Ti = 2 : !tdisreq1_Ti, STATE_Ti := 0 |
{ TDISIND12     }   STATE_Ti = 2 : ?tdisind1_Ti, STATE_Ti := 0 |
{ TDATAREQ1     }   STATE_Ti = 2 : !tdatareq1_Ti, STATE_Ti := 2 |
{ TDATAIND1     }   STATE_Ti = 2 : ?tdataind1_Ti, STATE_Ti := 2
od;

task TNia;
input
    tconreq_TNia,
    tdisreq1_TNia,
    tdatareq1_TNia;
output
    tdisind1_TNia,
    ndatareq1_cr_TNia,
    ndatareq1_ntdata_TNia,
    tnsync_TNia;
declare
    STATE_TNia : 0..4;
init
    STATE_TNia := 0;
do
    STATE_TNia = 0 : ?tconreq_TNia, STATE_TNia := 1 |
    STATE_TNia = 1 : ?tdisreq1_TNia, STATE_TNia := 0 |
    STATE_TNia = 1 : !tdisind1_TNia, STATE_TNia := 0 |
    STATE_TNia = 1 : !ndatareq1_cr_TNia, STATE_TNia := 2 |
    STATE_TNia = 2 : ?tdatareq1_TNia, STATE_TNia := 3 |
    STATE_TNia = 2 : ?tdisreq1_TNia, STATE_TNia := 4 |
    STATE_TNia = 2 : !tdisind1_TNia, STATE_TNia := 4 |
    STATE_TNia = 3 : !ndatareq1_ntdata_TNia, STATE_TNia := 2 |
    STATE_TNia = 3 : ?tdisreq1_TNia, STATE_TNia := 4 |
    STATE_TNia = 3 : !tdisind1_TNia, STATE_TNia := 4 |
    STATE_TNia = 4 : !tnsync_TNia, STATE_TNia := 0
od;

task TNai;
input
    ndataind1_cc_TNai,
    ndataind1_dr_TNai,
    ndisind1_bis_TNai,
    ndataind1_ntdata_TNai,
    tnsync_TNai;
output
    ndisreq1_bis_TNai,
    tconconf_TNai,
    tdataind1_TNai;
declare
```

```
    STATE_TNai : 0..5;
init
    STATE_TNai := 0;
do
    STATE_TNai = 0 : ?ndataind1_cc_TNai, STATE_TNai := 1 |
    STATE_TNai = 0 : ?ndataind1_dr_TNai, STATE_TNai := 5 |
    STATE_TNai = 0 : ?ndisind1_bis_TNai, STATE_TNai := 5 |
    STATE_TNai = 0 : !ndisreq1_bis_TNai, STATE_TNai := 5 |
    STATE_TNai = 1 : !tconconf_TNai, STATE_TNai := 2 |
    STATE_TNai = 1 : ?ndisind1_bis_TNai, STATE_TNai := 5 |
    STATE_TNai = 1 : !ndisreq1_bis_TNai, STATE_TNai := 5 |
    STATE_TNai = 1 : ?ndataind1_ntdata_TNai, STATE_TNai := 3 |
    STATE_TNai = 2 : ?ndataind1_ntdata_TNai, STATE_TNai := 4 |
    STATE_TNai = 2 : ?ndisind1_bis_TNai, STATE_TNai := 5 |
    STATE_TNai = 2 : !ndisreq1_bis_TNai, STATE_TNai := 5 |
    STATE_TNai = 3 : !tdataind1_TNai, STATE_TNai := 1 |
    STATE_TNai = 3 : !tconconf_TNai, STATE_TNai := 4 |
    STATE_TNai = 3 : ?ndisind1_bis_TNai, STATE_TNai := 5 |
    STATE_TNai = 3 : !ndisreq1_bis_TNai, STATE_TNai := 5 |
    STATE_TNai = 4 : !tdataind1_TNai, STATE_TNai := 2 |
    STATE_TNai = 4 : ?ndisind1_bis_TNai, STATE_TNai := 5 |
    STATE_TNai = 4 : !ndisreq1_bis_TNai, STATE_TNai := 5 |
    STATE_TNai = 5 : ?tnsync_TNai, STATE_TNai := 0
od;

task TNi;
input
    ndisind1_TNi,
    nconconf_TNi,
    ndatareq1_cr_TNi,
    ndisind1_bis_TNi,
    ndisreq1_bis_TNi,
    ndataind1_dr_TNi,
    ndataind1_cc_TNi,
    ndataind1_ntdata_TNi,
    ndatareq1_ntdata_TNi;
output
    nconreq_TNi,
    ndisreq1_TNi;
declare
    STATE_TNi : 0..4;
init
    STATE_TNi := 0;
do
    STATE_TNi = 0 : !nconreq_TNi, STATE_TNi := 1 |
    STATE_TNi = 1 : !ndisreq1_TNi, STATE_TNi := 0 |
    STATE_TNi = 1 : ?ndisind1_TNi, STATE_TNi := 0 |
    STATE_TNi = 1 : ?nconconf_TNi, STATE_TNi := 2 |
    STATE_TNi = 2 : !ndisreq1_TNi, STATE_TNi := 0 |
    STATE_TNi = 2 : ?ndisind1_TNi, STATE_TNi := 0 |
    STATE_TNi = 2 : ?ndatareq1_cr_TNi, STATE_TNi := 3 |
    STATE_TNi = 3 : ?ndisind1_bis_TNi, STATE_TNi := 0 |
    STATE_TNi = 3 : ?ndisreq1_bis_TNi, STATE_TNi := 0 |
    STATE_TNi = 3 : ?ndataind1_dr_TNi, STATE_TNi := 2 |
    STATE_TNi = 3 : ?ndataind1_cc_TNi, STATE_TNi := 4 |
```

```
    STATE_TNi = 4 : ?ndisind1_bis_TNi, STATE_TNi := 0 |
    STATE_TNi = 4 : ?ndisreq1_bis_TNi, STATE_TNi := 0 |
    STATE_TNi = 4 : ?ndataind1_ntdata_TNi, STATE_TNi := 4 |
    STATE_TNi = 4 : ?ndatareq1_ntdata_TNi, STATE_TNi := 4
od;

task Nia;
input
    nconreq_Nia,
    ndisreq1_Nia,
    ndisreq1_bis_Nia,
    ndatareq1_cr_Nia,
    ndatareq1_ntdata_Nia,
    ndatareq1_cc_Nia,
    ndatareq1_dr_Nia;
output
    ndisind1_Nia,
    ndisind1_bis_Nia,
    nconind_Nia,
    ndataind2_cr_Nia,
    ndataind2_ntdata_Nia,
    ndataind2_cc_Nia,
    ndataind2_dr_Nia,
    nsync_Nia;
declare
    STATE_Nia : 0..7;
init
    STATE_Nia := 0;
do
    STATE_Nia = 0 : ?nconreq_Nia, STATE_Nia := 1 |
    STATE_Nia = 1 : ?ndisreq1_Nia, STATE_Nia := 0 |
    STATE_Nia = 1 : ?ndisreq1_bis_Nia, STATE_Nia := 0 |
    STATE_Nia = 1 : !ndisind1_Nia, STATE_Nia := 0 |
    STATE_Nia = 1 : !ndisind1_bis_Nia, STATE_Nia := 0 |
    STATE_Nia = 1 : !nconind_Nia, STATE_Nia := 2 |
    STATE_Nia = 2 : ?ndatareq1_cr_Nia, STATE_Nia := 3 |
    STATE_Nia = 2 : ?ndatareq1_ntdata_Nia, STATE_Nia := 4 |
    STATE_Nia = 2 : ?ndatareq1_cc_Nia, STATE_Nia := 5 |
    STATE_Nia = 2 : ?ndatareq1_dr_Nia, STATE_Nia := 6 |
    STATE_Nia = 2 : ?ndisreq1_Nia, STATE_Nia := 7 |
    STATE_Nia = 2 : ?ndisreq1_bis_Nia, STATE_Nia := 7 |
    STATE_Nia = 2 : !ndisind1_Nia, STATE_Nia := 7 |
    STATE_Nia = 2 : !ndisind1_bis_Nia, STATE_Nia := 7 |
    STATE_Nia = 3 : !ndataind2_cr_Nia, STATE_Nia := 2 |
    STATE_Nia = 3 : ?ndisreq1_Nia, STATE_Nia := 7 |
    STATE_Nia = 3 : ?ndisreq1_bis_Nia, STATE_Nia := 7 |
    STATE_Nia = 3 : !ndisind1_Nia, STATE_Nia := 7 |
    STATE_Nia = 3 : !ndisind1_bis_Nia, STATE_Nia := 7
    STATE_Nia = 4 : !ndataind2_ntdata_Nia, STATE_Nia := 2 |
    STATE_Nia = 4 : ?ndisreq1_Nia, STATE_Nia := 7 |
    STATE_Nia = 4 : ?ndisreq1_bis_Nia, STATE_Nia := 7 |
    STATE_Nia = 4 : !ndisind1_Nia, STATE_Nia := 7 |
    STATE_Nia = 4 : !ndisind1_bis_Nia, STATE_Nia := 7 |
    STATE_Nia = 5 : !ndataind2_cc_Nia, STATE_Nia := 2 |
    STATE_Nia = 5 : ?ndisreq1_Nia, STATE_Nia := 7 |
```

```
    STATE_Nia = 5 : ?ndisreq1_bis_Nia, STATE_Nia := 7 |
    STATE_Nia = 5 : !ndisind1_Nia, STATE_Nia := 7 |
    STATE_Nia = 5 : !ndisind1_bis_Nia, STATE_Nia := 7 |
    STATE_Nia = 6 : !ndataind2_dr_Nia, STATE_Nia := 2 |
    STATE_Nia = 6 : ?ndisreq1_Nia, STATE_Nia := 7 |
    STATE_Nia = 6 : ?ndisreq1_bis_Nia, STATE_Nia := 7 |
    STATE_Nia = 6 : !ndisind1_Nia, STATE_Nia := 7 |
    STATE_Nia = 6 : !ndisind1_bis_Nia, STATE_Nia := 7 |
    STATE_Nia = 7 : !nsync_Nia, STATE_Nia := 0
od;

task Nai;
input
    nconresp_Nai,
    ndisreq2_Nai,
    ndisreq2_bis_Nai,
    ndatareq2_cr_Nai,
    ndatareq2_ntdata_Nai,
    ndatareq2_cc_Nai,
    ndatareq2_dr_Nai,
    nsync_Nai;
output
    ndisind2_Nai,
    ndisind2_bis_Nai,
    nconconf_Nai,
    ndataind1_cr_Nai,
    ndataind1_ntdata_Nai,
    ndataind1_cc_Nai,
    ndataind1_dr_Nai;
declare
    STATE_Nai : 0..11;
init
    STATE_Nai := 0;
do
    STATE_Nai = 0 : ?nconresp_Nai, STATE_Nai := 1 |
    STATE_Nai = 0 : ?ndisreq2_Nai, STATE_Nai := 11 |
    STATE_Nai = 0 : ?ndisreq2_bis_Nai, STATE_Nai := 11 |
    STATE_Nai = 0 : !ndisind2_Nai, STATE_Nai := 11 |
    STATE_Nai = 0 : !ndisind2_bis_Nai, STATE_Nai := 11 |
    STATE_Nai = 1 : !nconconf_Nai, STATE_Nai := 2 |
    STATE_Nai = 1 : ?ndatareq2_cr_Nai, STATE_Nai := 3 |
    STATE_Nai = 1 : ?ndatareq2_ntdata_Nai, STATE_Nai := 4 |
    STATE_Nai = 1 : ?ndatareq2_cc_Nai, STATE_Nai := 5 |
    STATE_Nai = 1 : ?ndatareq2_dr_Nai, STATE_Nai := 6 |
    STATE_Nai = 1 : ?ndisreq2_Nai, STATE_Nai := 11 |
    STATE_Nai = 1 : ?ndisreq2_bis_Nai, STATE_Nai := 11 |
    STATE_Nai = 1 : !ndisind2_Nai, STATE_Nai := 11 |
    STATE_Nai = 1 : !ndisind2_bis_Nai, STATE_Nai := 11 |
    STATE_Nai = 2 : ?ndatareq2_cr_Nai, STATE_Nai := 7 |
    STATE_Nai = 2 : ?ndatareq2_ntdata_Nai, STATE_Nai := 8 |
    STATE_Nai = 2 : ?ndatareq2_cc_Nai, STATE_Nai := 9 |
    STATE_Nai = 2 : ?ndatareq2_dr_Nai, STATE_Nai := 10 |
    STATE_Nai = 2 : ?ndisreq2_Nai, STATE_Nai := 11 |
    STATE_Nai = 2 : ?ndisreq2_bis_Nai, STATE_Nai := 11 |
    STATE_Nai = 2 : !ndisind2_Nai, STATE_Nai := 11 |
```

```
    STATE_Nai = 2 : !ndisind2_bis_Nai, STATE_Nai := 11 |
    STATE_Nai = 3 : !ndataind1_cr_Nai, STATE_Nai := 1 |
    STATE_Nai = 3 : !nconconf_Nai, STATE_Nai := 7 |
    STATE_Nai = 3 : ?ndisreq2_Nai, STATE_Nai := 11 |
    STATE_Nai = 3 : ?ndisreq2_bis_Nai, STATE_Nai := 11 |
    STATE_Nai = 3 : !ndisind2_Nai, STATE_Nai := 11 |
    STATE_Nai = 3 : !ndisind2_bis_Nai, STATE_Nai := 11 |
    STATE_Nai = 4 : !ndataind1_ntdata_Nai, STATE_Nai := 1 |
    STATE_Nai = 4 : !nconconf_Nai, STATE_Nai := 8 |
    STATE_Nai = 4 : ?ndisreq2_Nai, STATE_Nai := 11 |
    STATE_Nai = 4 : ?ndisreq2_bis_Nai, STATE_Nai := 11 |
    STATE_Nai = 4 : !ndisind2_Nai, STATE_Nai := 11 |
    STATE_Nai = 4 : !ndisind2_bis_Nai, STATE_Nai := 11 |
    STATE_Nai = 5 : !ndataind1_cc_Nai, STATE_Nai := 1 |
    STATE_Nai = 5 : !nconconf_Nai, STATE_Nai := 9 |
    STATE_Nai = 5 : ?ndisreq2_Nai, STATE_Nai := 11 |
    STATE_Nai = 5 : ?ndisreq2_bis_Nai, STATE_Nai := 11 |
    STATE_Nai = 5 : !ndisind2_Nai, STATE_Nai := 11 |
    STATE_Nai = 5 : !ndisind2_bis_Nai, STATE_Nai := 11 |
    STATE_Nai = 6 : !ndataind1_dr_Nai, STATE_Nai := 1 |
    STATE_Nai = 6 : !nconconf_Nai, STATE_Nai := 10 |
    STATE_Nai = 6 : ?ndisreq2_Nai, STATE_Nai := 11 |
    STATE_Nai = 6 : ?ndisreq2_bis_Nai, STATE_Nai := 11 |
    STATE_Nai = 6 : !ndisind2_Nai, STATE_Nai := 11 |
    STATE_Nai = 6 : !ndisind2_bis_Nai, STATE_Nai := 11 |
    STATE_Nai = 7 : !ndataind1_cr_Nai, STATE_Nai := 2 |
    STATE_Nai = 7 : ?ndisreq2_Nai, STATE_Nai := 11 |
    STATE_Nai = 7 : ?ndisreq2_bis_Nai, STATE_Nai := 11 |
    STATE_Nai = 7 : !ndisind2_Nai, STATE_Nai := 11 |
    STATE_Nai = 7 : !ndisind2_bis_Nai, STATE_Nai := 11 |
    STATE_Nai = 8 : !ndataind1_ntdata_Nai, STATE_Nai := 2 |
    STATE_Nai = 8 : ?ndisreq2_Nai, STATE_Nai := 11 |
    STATE_Nai = 8 : ?ndisreq2_bis_Nai, STATE_Nai := 11 |
    STATE_Nai = 8 : !ndisind2_Nai, STATE_Nai := 11 |
    STATE_Nai = 8 : !ndisind2_bis_Nai, STATE_Nai := 11 |
    STATE_Nai = 9 : !ndataind1_cc_Nai, STATE_Nai := 2 |
    STATE_Nai = 9 : ?ndisreq2_Nai, STATE_Nai := 11 |
    STATE_Nai = 9 : ?ndisreq2_bis_Nai, STATE_Nai := 11 |
    STATE_Nai = 9 : !ndisind2_Nai, STATE_Nai := 11 |
    STATE_Nai = 9 : !ndisind2_bis_Nai, STATE_Nai := 11 |
    STATE_Nai = 10 : !ndataind1_dr_Nai, STATE_Nai := 2
    STATE_Nai = 10 : ?ndisreq2_Nai, STATE_Nai := 11
    STATE_Nai = 10 : ?ndisreq2_bis_Nai, STATE_Nai := 11 |
    STATE_Nai = 10 : !ndisind2_Nai, STATE_Nai := 11
    STATE_Nai = 10 : !ndisind2_bis_Nai, STATE_Nai := 11
    STATE_Nai = 11 : ?nsync_Nai, STATE_Nai := 0
od;

task NTa;
input
    nconind_NTa,
    ndisind2_NTa,
    ndataind2_cr_NTa,
    ndisind2_bis_NTa,
    ndisreq2_bis_NTa,
```

```
    ndatareq2_dr_NTa,
    ndatareq2_cc_NTa,
    ndatareq2_ntdata_NTa,
    ndataind2_ntdata_NTa;
output
    ndisreq2_NTa,
    nconresp_NTa;
declare
    STATE_NTa : 0..4;
init
    STATE_NTa := 0;
do
    STATE_NTa = 0 : ?nconind_NTa, STATE_NTa := 1 |
    STATE_NTa = 1 : !ndisreq2_NTa, STATE_NTa := 0 |
    STATE_NTa = 1 : ?ndisind2_NTa, STATE_NTa := 0
    STATE_NTa = 1 : !nconresp_NTa, STATE_NTa := 2
    STATE_NTa = 2 : !ndisreq2_NTa, STATE_NTa := 0 |
    STATE_NTa = 2 : ?ndisind2_NTa, STATE_NTa := 0 |
    STATE_NTa = 2 : ?ndataind2_cr_NTa, STATE_NTa := 3 |
    STATE_NTa = 3 : ?ndisind2_bis_NTa, STATE_NTa := 0 |
    STATE_NTa = 3 : ?ndisreq2_bis_NTa, STATE_NTa := 0 |
    STATE_NTa = 3 : ?ndatareq2_dr_NTa, STATE_NTa := 2 |
    STATE_NTa = 3 : ?ndatareq2_cc_NTa, STATE_NTa := 4 |
    STATE_NTa = 4 : ?ndisind2_bis_NTa, STATE_NTa := 0 |
    STATE_NTa = 4 : ?ndisreq2_bis_NTa, STATE_NTa := 0 |
    STATE_NTa = 4 : ?ndatareq2_ntdata_NTa, STATE_NTa := 4 |
    STATE_NTa = 4 : ?ndataind2_ntdata_NTa, STATE_NTa := 4
od;

task NTai;
input
    tconresp_NTai,
    tdisreq2_NTai,
    tdatareq2_NTai;
output
    tdisind2_NTai,
    ndatareq2_cc_NTai,
    ndatareq2_ntdata_NTai,
    ntsync_NTai;
declare
    STATE_NTai : 0..5;
init
    STATE_NTai := 0;
do
    STATE_NTai = 0 : ?tconresp_NTai, STATE_NTai := 1 |
    STATE_NTai = 0 : ?tdisreq2_NTai, STATE_NTai := 5 |
    STATE_NTai = 0 : !tdisind2_NTai, STATE_NTai := 5 |
    STATE_NTai = 1 : ?tdatareq2_NTai, STATE_NTai := 2
    STATE_NTai = 1 : !ndatareq2_cc_NTai, STATE_NTai := 3
    STATE_NTai = 1 : ?tdisreq2_NTai, STATE_NTai := 5 |
    STATE_NTai = 1 : !tdisind2_NTai, STATE_NTai := 5 |
    STATE_NTai = 2 : !ndatareq2_ntdata_NTai, STATE_NTai := 1 |
    STATE_NTai = 2 : !ndatareq2_cc_NTai, STATE_NTai := 4 |
    STATE_NTai = 2 : ?tdisreq2_NTai, STATE_NTai := 5 |
    STATE_NTai = 2 : !tdisind2_NTai, STATE_NTai := 5 |
```

```
    STATE_NTai = 3 : ?tdatareq2_NTai, STATE_NTai := 4 |
    STATE_NTai = 3 : ?tdisreq2_NTai, STATE_NTai := 5 |
    STATE_NTai = 3 : !tdisind2_NTai, STATE_NTai := 5 |
    STATE_NTai = 4 : !ndatareq2_ntdata_NTai, STATE_NTai := 3 |
    STATE_NTai = 4 : ?tdisreq2_NTai, STATE_NTai := 5 |
    STATE_NTai = 4 : !tdisind2_NTai, STATE_NTai := 5 |
    STATE_NTai = 5 : !ntsync_NTai, STATE_NTai := 0
od;

task NTia;
input
    ndataind2_cr_NTia,
    ndisind2_bis_NTia,
    ndataind2_ntdata_NTia,
    ntsync_NTia;
output
    ndatareq2_dr_NTia,
    ndisreq2_bis_NTia,
    tconind_NTia,
    tdataind2_NTia;
declare
    STATE_NTia : 0..4;
init
    STATE_NTia := 0;
do
    STATE_NTia = 0 : ?ndataind2_cr_NTia, STATE_NTia := 1 |
    STATE_NTia = 1 : !ndatareq2_dr_NTia, STATE_NTia := 0 |
    STATE_NTia = 1 : !ndisreq2_bis_NTia, STATE_NTia := 0 |
    STATE_NTia = 1 : ?ndisind2_bis_NTia, STATE_NTia := 0 |
    STATE_NTia = 1 : !tconind_NTia, STATE_NTia := 2 |
    STATE_NTia = 2 : ?ndataind2_ntdata_NTia, STATE_NTia := 3 |
    STATE_NTia = 2 : !ndisreq2_bis_NTia, STATE_NTia := 4 |
    STATE_NTia = 2 : ?ndisind2_bis_NTia, STATE_NTia := 4 |
    STATE_NTia = 3 : !tdataind2_NTia, STATE_NTia := 2 |
    STATE_NTia = 3 : !ndisreq2_bis_NTia, STATE_NTia := 4 |
    STATE_NTia = 3 : ?ndisind2_bis_NTia, STATE_NTia := 4 |
    STATE_NTia = 4 : ?ntsync_NTia, STATE_NTia := 0
od;

task Ta;
input
    tconind_Ta,
    tdisind2_Ta,
    tdataind2_Ta;
output
    tdisreq2_Ta,
    tconresp_Ta,
    tdatareq2_Ta;
declare
    STATE_Ta : 0..2;
init
    STATE_Ta := 0;
do
{ TCONIND       }   STATE_Ta = 0 : ?tconind_Ta, STATE_Ta := 1 |
{ TDISREQ21     }   STATE_Ta = 1 : !tdisreq2_Ta, STATE_Ta := 0 |
```

```
{ TDISIND21      }   STATE_Ta = 1 : ?tdisind2_Ta, STATE_Ta := 0 |
{ TCONRESP       }   STATE_Ta = 1 : !tconresp_Ta, STATE_Ta := 2 |
{ TDISREQ22      }   STATE_Ta = 2 : !tdisreq2_Ta, STATE_Ta := 0 |
{ TDISIND22      }   STATE_Ta = 2 : ?tdisind2_Ta, STATE_Ta := 0 |
{ TDATAREQ2      }   STATE_Ta = 2 : !tdatareq2_Ta, STATE_Ta := 2
{ TDATAIND2      }   STATE_Ta = 2 : ?tdataind2_Ta, STATE_Ta := 2
od;

broad
   LOW_ndatareq1_cr, LOW_ndatareq1_ntdata, LOW_ndatareq1_cc,
   LOW_ndatareq1_dr,
   LOW_ndataind1_cr, LOW_ndataind1_ntdata, LOW_ndataind1_cc,
   LOW_ndataind1_dr,

   LOW_ndatareq2_cr, LOW_ndatareq2_ntdata, LOW_ndatareq2_cc,
   LOW_ndatareq2_dr,
   LOW_ndataind2_cr, LOW_ndataind2_ntdata, LOW_ndataind2_cc,
   LOW_ndataind2_dr,

   LOW_ndisreq1_bis, LOW_ndisind1_bis, LOW_ndisreq2_bis,
   LOW_ndisind2_bis;

port
   LOW_tconreq, LOW_tconind, LOW_tconresp, LOW_tconconf,
   LOW_tdisreq1, LOW_tdisind1, LOW_tdisreq2, LOW_tdisind2,
   LOW_tdatareq1 LOW_tdataind1, LOW_tdatareq2, LOW_tdataind2,

   LOW_nconreq, LOW_nconind, LOW_nconresp, LOW_nconconf,
   LOW_ndisreq1, LOW_ndisind1, LOW_ndisreq2, LOW_ndisind2,
   LOW_nsync,

   LOW_tnsync, LOW_ntsync;

body
   Ti (LOW_tdisind1,
      LOW_tconconf,
      LOW_tdataind1,
      LOW_tconreq,
      LOW_tdisreq1,
      LOW_tdatareq1) //

   TNia (LOW_tconreq,
        LOW_tdisreq1,
        LOW_tdatareq1,
        LOW_tdisind1,
        LOW_ndatareq1_cr,
        LOW_ndatareq1_ntdata,
        LOW_tnsync) //

   TNai (LOW_ndataind1_cc,
        LOW_ndataind1_dr,
        LOW_ndisind1_bis,
        LOW_ndataind1_ntdata,
        LOW_tnsync,
        LOW_ndisreq1_bis,
```

```
        LOW_tconconf,
        LOW_tdataind1) //

    TNi (LOW_ndisind1,
        LOW_nconconf,
        LOW_ndatareq1_cr,
        LOW_ndisind1_bis,
        LOW_ndisreq1_bis,
        LOW_ndataind1_dr,
        LOW_ndataind1_cc,
        LOW_ndataind1_ntdata,
        LOW_ndatareq1_ntdata,
        LOW_nconreq,
        LOW_ndisreq1) //

    Nia (LOW_nconreq,
        LOW_ndisreq1,
        LOW_ndisreq1_bis,
        LOW_ndatareq1_cr,
        LOW_ndatareq1_ntdata,
        LOW_ndatareq1_cc,
        LOW_ndatareq1_dr,
        LOW_ndisind1,
        LOW_ndisind1_bis,
        LOW_nconind,
        LOW_ndataind2_cr,
        LOW_ndataind2_ntdata,
        LOW_ndataind2_cc,
        LOW_ndataind2_dr,
        LOW_nsync) //

    Nai (LOW_nconresp,
        LOW_ndisreq2,
        LOW_ndisreq2_bis,
        LOW_ndatareq2_cr,
        LOW_ndatareq2_ntdata,
        LOW_ndatareq2_cc,
        LOW_ndatareq2_dr,
        LOW_nsync,
        LOW_ndisind2,
        LOW_ndisind2_bis,
        LOW_nconconf,
        LOW_ndataind1_cr,
        LOW_ndataind1_ntdata,
        LOW_ndataind1_cc,
        LOW_ndataind1_dr) //

    NTa (LOW_nconind,
        LOW_ndisind2,
        LOW_ndataind2_cr,
        LOW_ndisind2_bis,
        LOW_ndisreq2_bis,
        LOW_ndatareq2_dr,
        LOW_ndatareq2_cc,
        LOW_ndatareq2_ntdata,
```

```
        LOW_ndataind2_ntdata,
        LOW_ndisreq2,
        LOW_nconresp) //

    NTai (LOW_tconresp,
        LOW_tdisreq2,
        LOW_tdatareq2,
        LOW_tdisind2,
        LOW_ndatareq2_cc,
        LOW_ndatareq2_ntdata,
        LOW_ntsync) //

    NTia (LOW_ndataind2_cr,
        LOW_ndisind2_bis,
        LOW_ndataind2_ntdata,
        LOW_ntsync,
        LOW_ndatareq2_dr,
        LOW_ndisreq2_bis,
        LOW_tconind,
        LOW_tdataind2) //

    Ta (LOW_tconind,
       LOW_tdisind2,
       LOW_tdataind2,
       LOW_tdisreq2,
       LOW_tconresp,
       LOW_tdatareq2).
```

Annexe C: évaluation des performances

Description de haut niveau
- 4 automates simples
- 4 variables d'état
- automate composé ayant 47 états

Description de bas niveau
- 10 automates simples
- 10 variables d'état
- automate composé ayant 2549 états

Le système CESAR est disponible sur calculateur HB-68 fonctionannt sous le système d'exploitation MULTICS. Pour les deux descriptions, le temps CPU virtuel nécessaire pour évaluer un ensemble de formules logiques en fonction du nombre de formules a été relevé. Une étude de régression linéaire donne les conclusions suivantes:

pour la description de haut niveau:
- pente: 0.06
- ordonnée à l'origine: 2.45
- variance: 0.79

pour la description de bas niveau:
- pente: 36.97
- ordonnée à l'origine: 13.03
- variance: 0.94

On peut interpréter ces résultats de la façon suivante: la pente correspond au nombre de secondes de temps CPU virtuel nécessaires pour évaluer une formule, une fois que le système a été initialisé. L'ordonnée à l'origine est la durée de la phase d'initialisation. La valeur élevée de la variance montre que l'approximation linéaire est une bonne modélisation.