

# Un support méthodologique pour la spécification formelle de systèmes “mixtes”

Pascal POIZAT, Christine CHOPPY & Jean-Claude ROYER  
IRIN, Université de Nantes & Ecole Centrale  
2 rue de la Houssinière, B.P. 92208  
44322 Nantes cedex 3  
tél: 02 51 12 58 22 - fax: 02 51 12 58 12  
{Prénom.Nom}@irin.univ-nantes.fr

2 septembre 1998  
révisé le 16 février 1999

## \*Résumé

Nous présentons un support méthodologique pour la spécification de systèmes “mixtes”, c’est à dire comprenant une partie contrôle et une partie données. La méthode est expliquée en utilisant une description semi-formelle, celle des “agendas”, qui permet de décrire clairement et précisément la liste des tâches et des validations correspondant à chacune de ses étapes. Nous illustrons notre méthode à travers l’exemple d’un nœud de transit dans un réseau de télécommunications.

**Mots-clés :** méthode, spécification formelle, automate, LOTOS, SDL

## \*Abstract

“Mixed systems” are systems that involve both a dynamic part (control description) and a static part (data types). We present here a methodological support for the specification of such systems. This method is presented in a semi-formal way using “agendas”, a formalism that allows to describe in a clear and precise way the tasks and validations corresponding to the method steps. We illustrate our method through the example of a telecommunication network transit node.

**Key-words:** method, formal specification, automaton, LOTOS, SDL

## Table des matières

<b>1</b>	<b>Introduction</b>	<b>2</b>
<b>2</b>	<b>Le nœud de transit</b>	<b>2</b>
<b>3</b>	<b>Une nouvelle méthode</b>	<b>4</b>
<b>4</b>	<b>Conclusion</b>	<b>24</b>
<b>A</b>	<b>Etude du Data Port Out</b>	<b>28</b>
<b>B</b>	<b>Etude du Control Port In</b>	<b>30</b>
<b>C</b>	<b>Etude de la Faulty Collection</b>	<b>32</b>
<b>D</b>	<b>Etude du Control Port Out</b>	<b>33</b>

## 1 Introduction

L'utilisation des méthodes formelles peut grandement contribuer à la qualité des systèmes informatiques et en particulier des systèmes de protocoles de communication. Elles permettent en effet d'obtenir une description non ambiguë, précise, complète des systèmes, et offrent la possibilité de pratiquer des tests, vérifications et validations, ce qui motive encore plus leur emploi.

Si l'importance de l'utilisation de spécifications formelles lors du développement de systèmes logiciels est largement acceptée, elles ne sont pas faciles à utiliser par un ingénieur "standard". Un support méthodologique est donc absolument nécessaire pour permettre la mise en pratique des méthodes formelles. Nous présentons ici une méthode pour la spécification de systèmes "mixtes", c'est à dire comportant une activité concurrente et gérant des types de données impliqués dans cette activité. Cette méthode est applicable pour des langages de spécification tels que LOTOS [PCR98b, PCR98a] et SDL, qui sont des techniques de description formelle normalisées par l'ISO pour la spécification formelle des systèmes ouverts distribués et en particulier des protocoles de communication.

Une spécification LOTOS [BB88] comporte une partie données (types et opérations) basée sur ACT ONE [EM85] et une partie contrôle (des définitions de processus - éventuellement avec paramètres formels -, et une expression comportementale - éventuellement avec instanciation des paramètres formels) basée sur CCS [Mil80] et CSP [Hoa78]. Une extension de LOTOS (ELOTOS [ISO98]) gère le temps.

SDL [EHS97, Tur93] est basé sur un automate d'état finis étendu pour le contrôle et des spécifications algébriques (même modèle que pour LOTOS) pour les données. Contrairement à LOTOS, SDL utilise des formats graphiques pour le contrôle. SDL permet aussi de spécifier le comportement des canaux de communication et de gérer le temps (timers). SDL permet depuis SDL-92 de gérer le non déterminisme. En LOTOS comme en SDL, les processus ne communiquent que par envoi de message (signaux en SDL), et il n'y a pas de variables partagées.

Nous utilisons pour la présentation de notre méthode le concept d'agendas, introduits dans [Hei98, HS97, GHD98]. Un agenda est une liste d'activités à effectuer pour réaliser une tâche de génie logiciel. Alors que de nombreuses méthodes sont expliquées de façon purement informelle, les agendas sont un moyen de décrire une méthode de façon semi-formelle, point par point et à différents niveaux de détail. Appliqués dans le cadre des spécifications formelles, les agendas guident le spécifieur dans le développement de la spécification, et facilitent ainsi l'emploi des méthodes formelles. Les agendas prévoient également les critères de validation qui permettent de vérifier la cohérence et la complétude de la spécification [Hei98].

Dans ce qui suit, nous utiliserons donc le formalisme des agendas pour expliquer notre méthode que nous appliquerons à la spécification d'un nœud de transit dans un réseau de télécommunications.

L'exemple qui servira d'étude de cas est présenté dans la section 2, il est donné sous la forme d'une suite de clauses. L'aperçu général de la méthode est illustré dans la section 3, puis chacune des différentes étapes et sous-étapes sont détaillées. L'accent est mis dans cette section sur l'un des processus de la spécification. Les annexes donnent le résultat de l'application de la méthode aux autres processus. Les spécifications algébriques des types de données peuvent aussi être trouvées en annexe.

## 2 Le nœud de transit

L'étude de cas concerne un nœud de transit servant au routage dans un réseau de télécommunications. Sa description informelle a été donnée dans le projet SPECS. Elle a ensuite été modifiée pendant le projet VTT [BCV94, Mou94]. Elle est donnée en anglais sous forme de clauses.

Il s'agit d'un simple nœud de transit où les messages arrivent, sont routés et émis.

**clause 1** The system to be specified consists of a transit node with: one *Control Port-In*, one *Control Port-Out*,  $N$  *Data Ports-In*,  $N$  *Data Ports-Out*,  $M$  *Routes Through*. (The limits of  $N$  and  $M$  are not specified.)

**clause 2** (a) Each port is serialized.

(b) All ports are concurrent to all others. The ports should be specified as separate, concurrent entities.

(c) Messages arrive from the environment only when a *Port-In* is able to treat them.

**clause 3** The node is “fair”. All messages are equally likely to be treated, when a selection must be made,

**clause 4** and all messages will eventually transit the node, or be placed in the collection of faulty messages.

\*\*\* Modification \*\*\* and all data messages will eventually transit the node, or becomes faulty.  
\*\*\*

**clause 5** *Initial State*: one *Control Port-In*, one *Control Port-Out*.

**clause 6** The *Control Port-In* accepts and treats the following three messages:

(a) *Add-Data-Port-In-~~is~~-Out*( $n$ ) gives the node knowledge of a new *Port-In*( $n$ ) and a new *Port-Out*( $n$ ). The nodes commences to accept and treat messages sent to the *Port-In*, as indicated below on *Data Port-In*.

(b) *Add-Route*( $m$ ), ( $n(i), n(j) \dots$ ) , associates route  $m$  with *Data-Port-Out*( $n(i), n(j), \dots$ ).

(c) *Send-Faults* The order in which the faulty messages are transmitted is not specified.  
\*\*\* Modification \*\*\* routes some messages in the faulty collection, if any, ... \*\*\*

**clause 7** A *Data Port-In* accepts and treats only messages of the type: *Route*( $m$ ).*Data*.

(a) The *Port-In* routes the message, unchanged, to any one (non determinate) of the *Data Ports-Out* associated with route  $m$ .

\*\*\* Modification \*\*\* ... to any one (non determinate) of the open *Data Ports-Out* associated with route  $m$ . If no such port exists, the message is put in the faulty collection. \*\*\*

(b) (Note that a *Data Port-Out* is serialized – the message has to be buffered until the *Data Port-Out* can process it).

(c) The message becomes a faulty message if its transit time through the node (from initial receipt by a *Data Port-In* to transmission by a *Data Port-Out*) is greater than a constant time  $T$ .

**clause 8** *Data Ports-Out* and *Control Port-Out* accept messages of any type and will transmit the message out of the node. Messages may leave the node in any order.

**clause 9** All faulty messages are saved until a *Send-Faults* command message causes them to be routed to *Control Port-Out*.

\*\*\* Modification \*\*\* All faulty messages are eventually placed in the faulty collection where they stay until ... \*\*\*

**clause 10** Faulty messages are (a) messages on the *Control Port-In* that are not one of the three commands listed, (b) messages on a *Data Port-In* that indicate an unknown route, or (c) messages whose transit time through the node is greater than  $T$ .

**clause 11** (a) Messages that exceed the transit time of  $T$  become faulty as soon as the time  $T$  is exceeded.

(b) It is permissible for a faulty message not to be routed to *Control Port-Out*

\*\*\* Modification \*\*\* ... not to be routed to *Control Port-Out* by a *Send-Faults* command \*\*\*

(because, for example, it has just become faulty, but has not yet been placed in a faulty message collection),

(c) but all faulty messages must eventually be sent to *Control Port-Out* with a succession of *Send-Faults* commands.

**clause 12** It may be assumed that a source of time (time-of-day or a signal each time interval) is available in the environment and need not be modeled within the specification.

### 3 Une nouvelle méthode

#### Vision globale

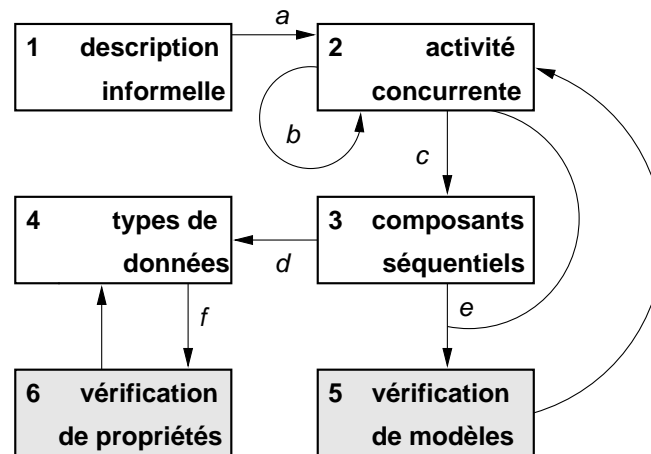


Figure 1 : Dépendances entre étapes au niveau global.

Pour l'établissement de la spécification, notre méthode se compose, à un premier niveau, de quatre étapes (cf. figure 1) auxquelles sont associées deux étapes de vérification :

1. **description informelle** du système à spécifier;
2. description de l'**activité concurrente**;
3. description des **composants séquentiels** sous forme d'automate;
4. spécification des **types de données** correspondants;
5. **vérification de modèles** / partie communication et concurrence;
6. **vérification de propriétés** sur les types de données.

Nous avons légèrement modifié le formalisme des agendas de façon à indiquer au niveau des dépendances les éléments utilisés entre étapes (cf. figure 1). Ces éléments sont :

- (a) données, fonctionnalités et contraintes décrites dans la description informelle;
- (b) types de données, contraintes et communications de processus à sous-processus;
- (c) conditions de communication et de comportement, préconditions et postconditions;
- (d) automate et typage des communications;

- (e) spécification des compositions et composants séquentiels;
- (f) spécification des types de données.

Chacune des étapes sera décrite dans un agenda différent.

## Etape 1 : Description informelle

étape	expression / schéma	conditions de validation
1.1 : fonctionnalités du système	$F_i$ : <b>texte</b>	o pas de redondance
1.2 : contraintes du système	$C_i$ : <b>texte</b>	o cohérence
1.3 : données du système	$D_i$ : <b>sorte</b>	

Figure 2 : Agenda de la description informelle.

Cette première étape permet de dégager les caractéristiques du système.

### Etape 1.1 : description des fonctionnalités externes du système

Il s'agit ici de recenser les opérations possibles. On leur donne un nom ( $F_i$  dans la figure 2) et on les décrit. Il ne doit pas y avoir de redondance : si deux opérations de nom différent font la même chose c'est que l'on a oublié quelque chose (dans la description et/ou éventuellement une contrainte associée).

Dans notre exemple, les opérations sont données par les clauses 6 et 7 :

- au niveau du Control Port In : réception d'un message de commande (**in\_cmde**)
- au niveau des Data Port In : réception d'un message de données (**in\_data**)

Les clauses 7 et 9 induisent de plus à envisager les opération suivantes :

- au niveau du Control Port Out : émission d'un message erroné (**out\_cmde**)
- au niveau des Data Port Out : émission d'un message de données (**out\_data**)

Enfin, la clause 12 précise qu'une source temporelle est disponible dans l'environnement. Nous supposons que le nœud de transit peut récupérer cette valeur du temps.

D'autres opérations apparaissent dans les clauses, mais elles sont internes au nœud de transit. Elles seront envisagées lors de la décomposition du système.

Notons que la clause 9 ne précise pas ce que le Control Port Out fait des messages d'erreurs reçus. Par analogie avec les Data Port Out, nous supposons qu'ils sont émis un par un.

### Etape 1.2 : description des contraintes du système

On exprime ici les contraintes sur le système : ordres, ordonnancement des opérations, limites de taille, ... Les contraintes doivent être cohérentes, c'est à dire ne pas être contradictoires.

Les contraintes de notre exemple seront reprises et détaillées au fur et à mesure de leur utilisation.

### Etape 1.3 : description des données du système

Les données sur lesquelles va travailler le système sont ici nommées, c'est donc un point de vue très abstrait pour l'instant (un nom et une sorte).

Le nœud de transit dispose d'une liste de numéros de ports actifs (clause 6a), d'une liste de routes (clauses 6b et 7a) et d'une collection de messages erronés (clauses 4, 6c, 7a et 9).

Le figure 3 représente la vue externe que l'on a du nœud de transit.

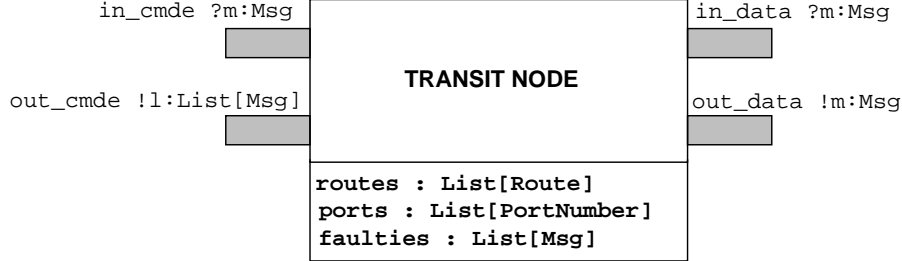


Figure 3 : Vue externe du nœud de transit.

### Etape 2 : Activité concurrente

Il s'agit de trouver les composants exécutés en parallèle. Chacun est modélisé par un processus. Les processus peuvent terminer ou non et cette information (appelée en LOTOS fonctionnalité du processus) est précisée.

Le squelette d'un processus est en LOTOS :

```
process nomProcessus [liste fonctionnalités externes] : fonctionnalité :=
--- comportement ---
endproc
```

La spécification peut être vue comme le processus de plus haut niveau. Dans le cas du nœud de transit, il est possible d'obtenir le squelette de spécification LOTOS suivant :

```
specification TransitNode [in_cmde, in_data, out_cmde, out_data] : noexit

library TRANSIT_NODE endlib

behavior

--- comportement ---

where

--- définition des processus --

endspec
```

La décomposition de processus en sous-processus concurrents s'inspire des styles de spécification orienté contraintes et orienté ressources pour LOTOS [BB88, Tur93, VSVSB91]. Chaque processus dispose d'une partie contrôle (partie dynamique) et d'éventuelles données (partie statique). En outre, un type de données  $tP$  est associé à chaque processus  $P$  et un objet de ce type constitue le paramètre formel du processus. Ce type devra disposer d'une opération d'initialisation utilisée à l'instanciation du processus que nous appellerons *init*. Le profil de cette opération dépend des données dont dispose le processus, données  $D_i$  qui sont éventuellement utilisées et/ou modifiées

par le processus. Ainsi, pour un processus  $P$ , disposant de données  $D_i$  de type  $S_i$ , un type  $tP$  lui est associé qui devra disposer d'une opération de profil  $\text{init} : S_1 \times \dots \times S_n \rightarrow tP$ .

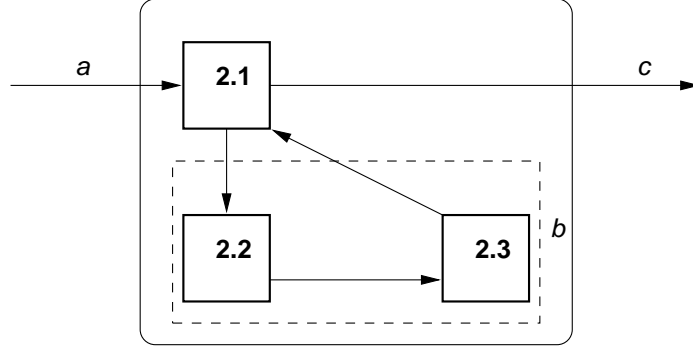


Figure 4 : Dépendances entre étapes : activité concurrente.

Nous décomposons cette étape de la manière suivante :

- 2.1 communications,
- 2.2 décomposition et répartition, et
- 2.3 composition parallèle.

Comme le montre la figure 4, les sous-étapes 2.2 et 2.3 sont itérées.

## Étape 2.1 : communications

étape	expression / schéma	conditions de validation
2.1.1 : ports de communication et données		<ul style="list-style-type: none"> <li>o pas d'oubli : prise en compte des <math>F_i</math> et <math>D_i</math> de 1.1 et 1.3</li> </ul>
2.1.2 : typage des communications	$F_i : ?x_j : s_j !x_k : s_k$	<ul style="list-style-type: none"> <li>o pas d'oubli : prise en compte des <math>F_i</math> de 1.1</li> <li>o sortes émises “disponibles”</li> </ul>

Figure 5 : Agenda des communications.

### étape 2.1.1 : ports de communication et données

Les interactions entre composants sont modélisées par des ports de communication formels qui représentent les services (interface) du composant. Les ports de communication externes sont fournis par la description informelle (fonctionnalités, sous-étape 1.1). Il en va de même pour les données ( $D_i$ , sous-étape 1.3).

### étape 2.1.2 : typage des communications

On détermine les communications entre les composants. Les données émises ou reçues sur les ports sont typées. L'émission d'une donnée de type  $T$  est notée  $!x:T$ . La réception d'une donnée de type  $T$  est notée  $?x:T$ . Ce typage des communications sert non seulement pour définir l'interface du processus mais aussi ultérieurement pour travailler sur la partie statique de la spécification<sup>1</sup>.

1. C'est pourquoi nous typons explicitement les émissions, ce qui n'est pas le cas de LOTOS dont la notation s'inspire.

Tous les types de données définis dans cette phase devront avoir une spécification algébrique associée. Il est possible d'avoir des types génériques mais ils devront alors être instanciés (étape 4).

Un critère de validation consiste à s'assurer que les sortes émises par un processus sont "disponibles". Une sorte *s* est *disponible* pour un processus si :

- soit elle est *directement disponible*, c'est-à-dire prédéfinie et importée, ou bien définie par le processus,
- soit il existe une opération importée  $f : d^* \rightarrow s$  telle que toutes les sortes de  $d^*$  sont disponibles.

La spécification des données ne se faisant qu'à l'étape 4, cette validation est incomplète à ce niveau.

Nous utilisons un type **Msg** pour représenter les différents messages du nœud de transit.

- réception d'un message de commande : `in_cmde : ?m:Msg`
- réception d'un message de données : `in_data : ?m:Msg`
- sortie d'un message erroné : `out_cmde : !m:Msg`
- sortie d'un message de données : `out_data : !m:Msg`

## Etape 2.2 : décomposition et répartition

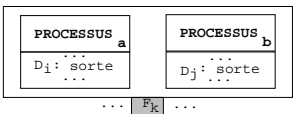
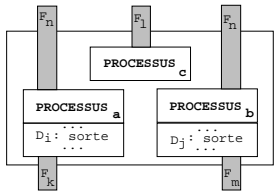
étape	expression / schéma	conditions de validation
2.2.1 : répartition des données		<ul style="list-style-type: none"> <li>o toutes les données réparties au niveau des sous-processus (cf. 2.1.1)</li> </ul>
2.2.2 : répartition des fonctionnalités		<ul style="list-style-type: none"> <li>o fonctionnalités et données associées</li> <li>o toutes les fonctionnalités réparties au niveau des sous-processus (cf. 2.1.1)</li> </ul>

Figure 6 : Agenda de la décomposition et de la répartition.

La décomposition d'un processus en sous-processus se fait en général à partir des informations (contraintes) de la description informelle. Lorsque cette dernière ne fournit pas d'information, il est possible d'effectuer le découpage à partir des données et/ou des fonctionnalités. Enfin, la décomposition peut être faite en cherchant à réutiliser des composants déjà spécifiés. Dans tous les cas, après la décomposition il est nécessaire de répartir les données et les fonctionnalités entre les sous-processus.

Décomposer un processus à partir de ses données permet de suivre un style de spécification orienté ressources. Les fonctionnalités sont ensuite regroupées avec les données qui leurs sont liées.

Décomposer un processus à partir de ses fonctionnalités se rapproche plus d'un style de spécification orienté contraintes. Dans ce cas, on considère des familles de fonctionnalités liées. Les données sont ensuite associées aux fonctionnalités qui les utilisent.



La clause 1 nous conduit naturellement à voir que 4 catégories de composants se trouvent dans le nœud de transit : le Control Port In (CPI), les Data Port In (DPI), le Control Port Out (CPO) et les Data Port Out (DPO). Un composant supplémentaire, Faulty Collection (FC), gèrera la collection des messages d'erreurs (clause 9). Plusieurs étapes de décomposition conduisent à l'obtention de ces composants.

Le système peut dans un premier temps être décomposé (figure 7) en partie contrôle (Control Ports) et partie données (Data Ports). Les routes, ainsi que les numéros de ports et les messages d'erreurs sont associés aux ports de contrôle.

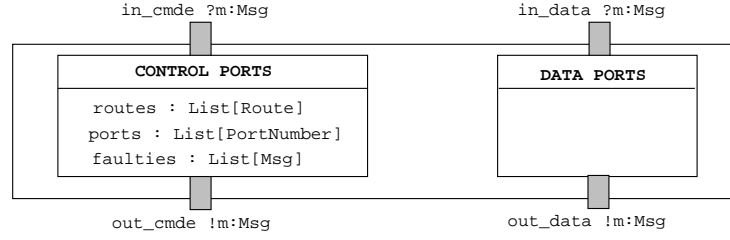


Figure 7 : Première décomposition du nœud de transit.

La partie contrôle peut ensuite être décomposée en un CPI et un composant de gestion des erreurs. Les routes et les numéros de ports seront placés dans le CPI puisqu'il modifie ces listes (clause 6). La liste des messages d'erreurs est associée au second composant. De son côté, la partie données est décomposée en ports d'entrée et ports de sortie (figure 8). Les ports de données doivent disposer d'un identifiant de façon à pouvoir les activer (voir la remarque concernant la création de nouveaux ports par l'emploi d'une communication interne de nom `enable` page 12). Le DPO dispose aussi d'un buffer pour les sorties de messages.

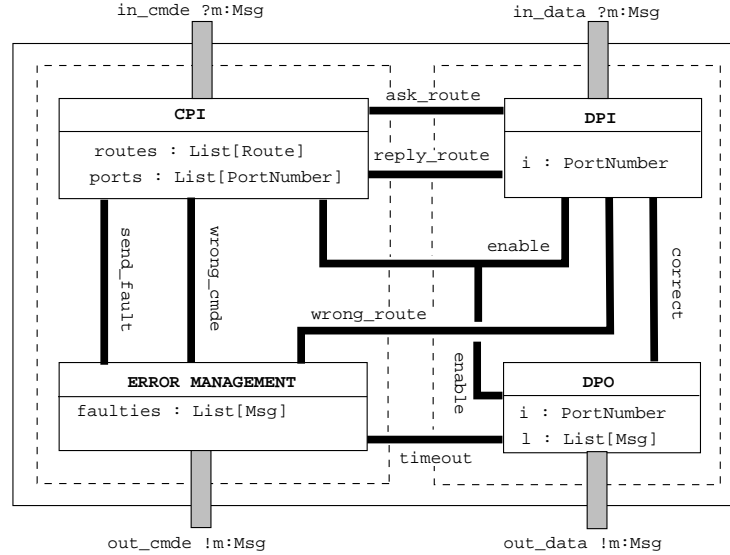


Figure 8 : Seconde décomposition du nœud de transit.

Dans un dernier temps, le composant de gestion des messages d'erreurs est décomposé en une FC qui contient les messages d'erreurs jusqu'à réception d'un message `sendfault` et en un CPO qui émet les messages d'erreurs vers l'extérieur du nœud de transit. Il dispose pour cela d'un buffer de sortie. De leur côté, les deux ensembles de ports de données sont chacun décomposés en N composants indépendants (DPI ou DPO). Les DPI devront pouvoir accéder aux informations de routage afin de connaître les ports associés à une route donnée (clause 2a).

Au regard des informations recueillies lors des étapes précédentes, nous pouvons représenter le système à l'aide de la figure 9. Les commentaires concernant les communications internes au nœud de transit et le nouvelles données induites par la décomposition ont été regroupées en fin de section concernant la composition parallèle.

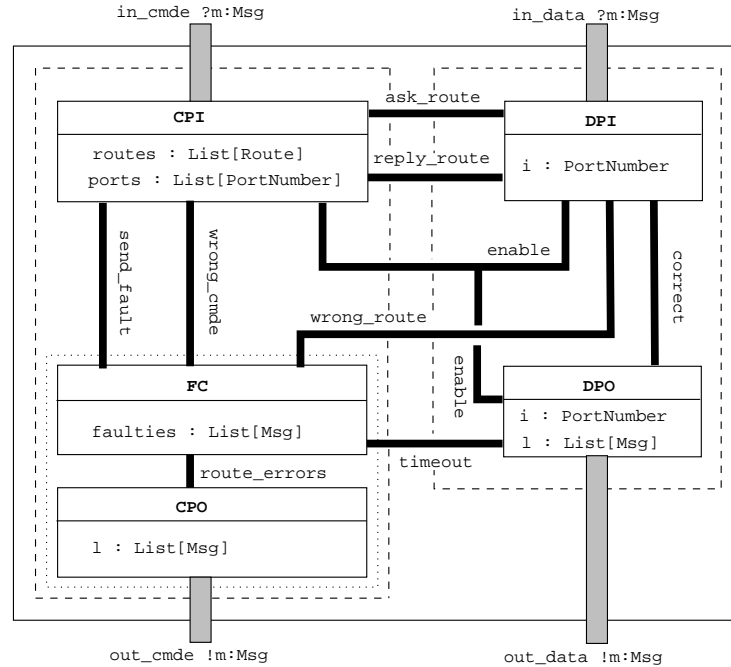


Figure 9 : Vue interne du nœud de transit.

### Etape 2.3 : composition parallèle (Figure 10)

étape	expression / schéma	conditions de validation
2.3.1 : choix d'un schéma de composition		
2.3.2 : application du schéma (cf. étapes 2.2 et 2.3.1)		o correspondance entre les contraintes (souhaitées) du processus et celles obtenues par la composition des sous-processus

Figure 10 : Agenda de la composition parallèle.

Les compositions de processus entrent souvent dans des cadres déjà connus. Nous proposons d'utiliser une bibliothèque de "schémas de composition". Si la composition ne se trouve pas dans cette bibliothèque, il est bien sûr possible de l'étendre. Quelques schémas (appelés schémas de

synchronisation ou styles d'architectures) peuvent être trouvés dans [Led87, Gar90, LLS97, HL97]. L'instanciation de schémas doit être simulée (voir [Tur97] par exemple) lorsque le langage de spécification ne permet pas de la faire directement (ce qui est le cas de LOTOS).

Si nécessaire, certaines particularités de la composition parallèle en LOTOS ou en SDL peuvent ici être mises en œuvre. Ainsi, il est possible de masquer les communications non pertinentes à un certain niveau d'abstraction grâce à l'opérateur *hide* de LOTOS.

Il faut noter que la composition parallèle de processus est un moyen d'exprimer certaines contraintes entre processus. Ainsi, la clause 2b crée une contrainte sur la composition parallèle des différents ports. Pour la respecter, les différents DPI (resp. DPO) seront plutôt composés par entrelacement que par synchronisation<sup>2</sup>.

Il est possible de vérifier si une composition de sous-processus respecte les contraintes désirées pour le processus englobant. Pour cela, il suffit de faire l'hypothèse sur les contraintes respectées par chaque sous-processus ou bien de réutiliser un composant pour lequel certaines propriétés ont déjà été prouvées. Il suffira alors que les contraintes du processus soient incluses dans la composition de celles des sous-processus.

La spécification de la composition parallèle des sous-processus peut être dérivée de l'ensemble des schémas de composition. Les différents composants sont reliés entre eux par les opérateurs de composition parallèle de LOTOS ou encore la structure de blocs et les canaux de SDL. Ceci se fait en respectant les contraintes de synchronisation (clause 2b par exemple).

Ainsi, en LOTOS, le système du nœud de transit s'écrit :

```
specification TransitNode [in_cmde, in_data, out_cmde, out_data] : noexit

library TRANSIT_NODE endlib

behavior

hide enable, ask_route, reply_route, wrong_route, timeout in
(
  hide send_fault, wrong_cmde in
  (
    CPI[in_cmde, enable, send_fault, wrong_cmde, ask_route, reply_route](init(vide of ListeRoute, vide of
|[send_fault, wrong_cmde]|
    (
      hide route_errors in
      (
        FC[route_errors, timeout, send_fault, wrong_cmde, wrong_route](init(vide of ListeMsg))
        |[route_errors]|
        CPO[route_errors, out_cmde](init(vide of ListeMsg))
      )
    )
  )
)
|[enable, ask_route, reply_route, wrong_route, timeout]|
(
  hide correct in
  (
    (
      DPI[in_data, ask_route, reply_route, enable, correct, wrong_route](init(1 of PortNumber))
    |||
    ...
    |||
      DPI[in_data, ask_route, reply_route, enable, correct, wrong_route](init(N of PortNumber))
    )
  )
)
```

---

2. La clause 2b précise que ces processus sont "concurrents" ce qui signifie ici qu'il ne communiquent pas entre eux et ne se synchronisent pas non plus.

```

| [correct] |
(
    DPO[out_data, enable, correct, timeout] (init(1 of PortNumber, vide of ListeMsg))
|||
    ...
|||
    DPO[out_data, enable, correct, timeout] (init(N of PortNumber, vide of ListeMsg))
)
)
)

where
    library CPI, CPO, DPI, DPO endlib

endspec

```

### Etude des contraintes induites par la décomposition du nœud de transit (communications internes)

**Messages erronés.** Ils sont sauvés dans la collection de la FC (clause 9). Il s'agit (clause 10) des messages de commande incorrects (`wrong_cmde`, origine CPI), des messages de données avec une route incorrecte (`wrong_route`, origine DPI) et des messages obsolètes (`timeout`, origine DPO).

**Emission des messages d'erreur.** La commande `send_fault`, reçue par le CPI déclenche le routage des messages erronés (en proportion non spécifiée) de la FC vers le CPO (clause 9). Cette phase utilise une communication de nom `send_fault` entre CPI et FC et une communication de nom `route_errors` entre FC et CPO. Nous supposons que les messages reçus par le CPO sont stockés et émis un par un (la spécification ne parle pas de ce point, mis à part la clause 8).

**Informations sur les routes.** Le DPI a besoin d'informations sur les routes (existence ou non, liste des ports de sortie associés). Ces informations sont dans le CPI. L'échange d'information se fera par communication de type question/réponse entre ces deux composants (`ask_route` et `reply_route`).

**Routage d'un message.** Lorsque le message indique une route correcte, le message est routé (communication `correct`) par le DPI sur un des DPO correspondant à la route (clause 7a).

**Nouveaux ports.** Lorsque le CPI reçoit le message `add_data_port_in_&_out` il crée les ports correspondants (clause 6). Ce que le CPI fait lorsque le port existe déjà n'est pas indiqué. Deux possibilités s'offraient ici. Tout d'abord, celle de créer dynamiquement des processus ports. Mais alors que se passerait-il si un port était créé deux fois? Pour ne pas sur-spécifier, nous avons opté pour la seconde solution, qui est de créer au départ tous les processus des DPI et DPO (N de chaque, clause 1); pour respecter la clause 5, ces processus ne sont activés que lorsque le CPI reçoit le message demandant de les créer (communication `enable`). Ce qu'il faut faire lorsqu'un Data Port est activé plusieurs fois n'est pas indiqué et donc pas spécifié.

### Etude des contraintes induites par la décomposition du nœud de transit (nouvelles données)

De nouvelles données apparaissent suite à la décomposition. Le DPO est sérialisé (clause 7b) et dispose donc d'un buffer de messages. Il en va de même pour le CPO. Les Data Ports dans leur ensemble ont besoin d'un identifiant de façon à les reconnaître lors des phases d'activation (par le CPI) ou de routage (DPI vers DPO).

L'étape 2 a été répétée plusieurs fois jusqu'à obtention du schéma de la figure 9.

En ce qui concerne le nœud de transit, nous présentons l'application de la méthode au composant DPI. Les automates concernant les autres composants, ainsi que la façon de les obtenir pourront être trouvés en annexe. De plus, le typage des communications du DPI a été étudié et est le suivant :

```

- in_data : ?m:Msg
- ask_route : !r:RouteNumber
- reply_route : !r:RouteNumber?l:List[PortNumber]
- enable : !ident:PortNumber
- wrong_route : !m:Msg
- correct : !ident:PortNumber !m:Msg

```

### Etape 3 : Composants séquentiels (Figure 11)

Nous avons ici légèrement modifié la notation des agendas : là où seul  $\vdash$  était utilisé pour indiquer une validation automatique, nous différencierons  $\vdash_\mu$  (qui indiquera une vérification de modèle) de  $\vdash_\lambda$  (qui indiquera une preuve algébrique).

étape	expression / schéma	conditions de validation																																								
3.1 : détermination des ports de O, OC, C, CC	0, OC, C, CC	o ensembles disjoints																																								
3.2 : conditions associées aux communications sur les ports de OC ou CC catégorie: précondition ou comportement	$F_i : C_j$ (catégorie)	o 1.2 (voir aussi 2.3)																																								
3.3 : relations entre conditions	$\vdash \phi_i(C_j)$	$\vdash_{\lambda}$ cohérence : $\vdash \wedge_i \phi_i(C_j)$																																								
3.4 : simplification des conditions		$\vdash_{\lambda}$ simplifications																																								
3.5 : création de la table des conditions	<table border="1"> <tr> <td>...</td> <td><math>C_i</math></td> <td>...</td> <td>interprétation</td> <td>référence</td> </tr> <tr><td></td><td></td><td></td><td></td><td></td></tr> <tr><td></td><td></td><td></td><td></td><td></td></tr> <tr><td></td><td></td><td></td><td></td><td></td></tr> <tr><td></td><td></td><td></td><td></td><td></td></tr> <tr><td></td><td></td><td></td><td></td><td></td></tr> <tr><td></td><td></td><td></td><td></td><td></td></tr> </table>	...	$C_i$	...	interprétation	référence																																				
...	$C_i$	...	interprétation	référence																																						
3.6 : élimination des cas impossibles	<table border="1"> <tr> <td>...</td> <td><math>C_i</math></td> <td>...</td> <td>interprétation</td> <td>référence</td> </tr> <tr><td></td><td></td><td></td><td></td><td></td></tr> <tr><td></td><td></td><td></td><td></td><td></td></tr> <tr><td></td><td></td><td></td><td></td><td></td></tr> <tr><td></td><td></td><td></td><td></td><td></td></tr> <tr><td></td><td></td><td></td><td></td><td></td></tr> <tr><td></td><td></td><td></td><td></td><td></td></tr> <tr><td></td><td></td><td></td><td></td><td></td></tr> </table>	...	$C_i$	...	interprétation	référence																																				$\vdash_{\lambda} \phi_i(C_j)$ (3.3)
...	$C_i$	...	interprétation	référence																																						
3.7 : états	$\mathcal{E}_i = \langle \dots, v(C_j), \dots \rangle$ $v(C_j) \in \{V, F\}$																																									
3.8 : préconditions des opérations	$\mathcal{P}_k = \langle \dots, v(C_j), \dots \rangle$ $v(C_j) \in \{V, F, \forall\}$	$\vdash_{\lambda}$ cohérence des préconditions par rapport à $\phi_i(C_j)$ $\vdash$ correction par rap. à 3.2																																								
3.9 : postcondition des opérations	$\mathcal{Q}_k = \langle \dots, \Phi_i(C'_j), \dots \rangle$	$C' : C +$ nouvelles conditions																																								
3.10 : relations entre conditions	$\vdash \phi_i(C'_j)$	$\vdash_{\lambda}$ cohérence : $\vdash \wedge_i \phi_i(C'_j)$ $\vdash_{\lambda}$ cohérence des postconditions par rapport à $\phi_i(C'_j)$ $\vdash$ correction par rap. à 3.2																																								
3.11 : calcul des transitions	$\mathcal{T} = f(\mathcal{E}, \mathcal{P}, \mathcal{Q})$	tous états atteignables (si non retour possible en 3.3)																																								
3.12 : choix d'un état initial en donnant la liste des opérations possibles ( $O_i$ ) et celles impossibles ( $\overline{O_j}$ )	$\mathcal{E}_{init}$	$\vdash_{\lambda}$ cohérence de $\wedge_i \mathcal{P}_{O_i} \wedge_j \neg \mathcal{P}_{\overline{O_j}}$ $\vdash_{\lambda}$ un seul état initial																																								
3.13 : simplifications de l'automate																																										
3.14 : traduction de l'automate dans le langage cible		$\vdash$ automate / spécification																																								
3.15 : simplifications de la spécification		$\vdash$ simplifications correctes																																								

Figure 11 : Agenda des composants séquentiels.

Chaque composant séquentiel est décrit par un automate d'états finis. L'utilisation d'un automate se justifie pleinement en ce que, comme le fait remarquer [Tur93], "la représentation sous forme d'arbre (comme souvent en LOTOS par exemple) du comportement entier d'un système n'est pratique que lorsque le nombre d'états est limité".

### Etape 3.1 : détermination des ports de O, OC, C, CC

Nous cherchons à représenter les composants séquentiels par des automates dont les états correspondent à différents états abstraits dans lesquels peut se trouver le composant. Dans le domaine des spécifications algébriques, le terme de constructeur s'applique à une opération dont le codomaine correspond au type de données courant (ce qui est généralement assimilé à une modification du type de donnée). De même, la notion d'observateur correspond aux autres opérations. Nous cherchons à caractériser les opérations faisant passer le composant d'un état abstrait à un autre. Par analogie avec les spécifications algébriques, nous nommerons *constructeurs* toute opération faisant changer d'état abstrait et *observateurs* les autres opérations.

Dans nos automates les états sont caractérisés par des conditions (étapes 3.5 à 3.7). Les constructeurs sont donc les opérations modifiant les conditions et les observateurs celles qui ne modifient pas les conditions.

Pour chaque processus, nous déterminons les quatre ensembles suivants :

- *C*, *constructeurs non conditionnés*: la communication sur ces ports modifie la valeur d'au moins une condition et n'est pas soumise à condition ;
- *CC*, *constructeurs conditionnés*: la communication sur ces ports modifie la valeur d'au moins une condition et est soumise à certaines conditions ;
- *O*, *observateurs non conditionnés*: la communication sur ces ports ne modifie la valeur d'aucune condition et n'est pas soumise à condition ;
- *OC*, *observateurs conditionnés*: la communication sur ces ports ne modifie la valeur d'aucune condition et est soumise à certaines conditions.

Ces ensembles sont récapitulés dans le tableau 1.

Le but de cette distinction est de faciliter la création de l'automate du processus. De plus, ces ensembles correspondent aussi à des ensembles d'opérations du type sous-jacent au processus.

$\sigma = C + CC + O + OC$  dénote l'ensemble des ports du processus. Il faut noter qu'un port reliant un processus *P* à un processus *Q* peut appartenir à deux ensembles différents selon que l'on s'occupe de *P* ou de *Q*.

	non conditionnés	conditionnés
modification état interne	<i>C</i>	<i>CC</i>
conservation état interne	<i>O</i>	<i>OC</i>

TAB. 1 – Ensembles *C*, *CC*, *O* et *OC*.

### Etape 3.2 : conditions associées aux communications

Par "condition" nous entendons aussi bien ce qui autorise une communication (précondition) que ce qui peut modifier l'effet d'une communication (condition de comportement). Dans le cas de *OC* et de *CC* nous donnons et nommons les conditions.

Il faut ici s'assurer que l'on a bien pris en compte toutes les conditions énumérées lors de l'étape 1.2. Toutefois, certaines d'entre elles sont prises en compte lors de la composition parallèle des sous-processus (étape 2.3).

L'application des étapes 3.1 et 3.2 au DPI à permis de recenser les conditions suivantes : autorisé (le port est actif), reçu (un message reçu et non traité), demandé (demande d'information de routage envoyée), répondu (réponse à la demande de routage reçue) et routeErr (erreur de routage). La condition routeErr porte sur la valeur de la variable `l` de `reply_route` après communication (voir le typage de la communication à la fin de l'étape 2).

Les opérations ont été rangées dans les différentes catégories et sont conditionnées comme indiqué ci-dessous :

- $O : \emptyset$ ,
- $C : \{\text{enable}\}$ ,
- $OC : \emptyset$ ,
- $CC : \{\text{in\_data} : \text{autorisé} \wedge \neg \text{reçu}, \text{ask\_route} : \text{autorisé} \wedge \text{reçu} \wedge \neg \text{répondu}, \text{reply\_route} : \text{autorisé} \wedge \text{reçu} \wedge \text{demandé} \wedge \neg \text{répondu}, \text{correct} : \text{autorisé} \wedge \text{reçu} \wedge \text{demandé} \wedge \text{répondu} \wedge \text{routeErr}, \text{wrong\_route} : \text{autorisé} \wedge \text{reçu} \wedge \text{demandé} \wedge \text{répondu} \wedge \neg \text{routeErr}\}$

### Etape 3.3 : relations entre conditions

Des formules logiques ( $\vdash \phi_i(C_j)$ ) expriment des propriétés reliant les conditions. Dans cette étape, un prouveur de théorèmes peut être employé afin de vérifier l'absence de contradictions entre ces formules.

Pour le DPI :  $\text{reçu} \Rightarrow \text{autorisé}$ ,  $\text{demandé} \Rightarrow \text{reçu}$ ,  $\text{répondu} \Rightarrow \text{demandé}$ ,  $\text{routeErr} \Rightarrow \text{répondu}$ . Il n'y a aucune contradiction.

### Etape 3.4 : simplification des conditions

Les formules logiques données précédemment par le spécifieur vont éventuellement permettre de simplifier certaines conditions, voire d'en restreindre le nombre. Cette étape peut être automatisée par l'utilisation d'un prouveur de théorèmes basé sur la réécriture (par exemple LP [GG89]).

L'application au DPI donne les résultats suivants (modifications uniquement) :

- $CC : \{\text{in\_data} : \text{autorisé} \wedge \neg \text{reçu}, \text{ask\_route} : \text{reçu} \wedge \neg \text{répondu}, \text{reply\_route} : \text{demandé} \wedge \neg \text{répondu}, \text{correct} : \text{répondu} \wedge \text{routeErr}, \text{wrong\_route} : \text{répondu} \wedge \neg \text{routeErr}\}$

### Etapes 3.5 à 3.7 : obtention des états

L'idée est qu'un processus peut se trouver dans différents états abstraits dans lesquels il est en mesure ou non de rendre un service (c'est à dire d'autoriser une communication sur un port). Les états sont donc obtenus par composition des conditions de communication (un recensement des conditions a été fait lors du travail sur les ports - étapes 3.2 à 3.4 - dans une table de vérité - 3.5).

Des formules logiques ( $\vdash_{\lambda} \phi_i(C_j)$ ) expriment des propriétés reliant ces conditions et permettent de supprimer les états incohérents (3.6).

Un automate est construit avec un état (3.7) pour chaque cas différent dans la table de composition des états (3.5). Les n-uplets correspondant à la table de vérité y sont notés.

La table du DPI (après suppression des états incohérents) est donnée en table 2.



autorisé	reçu	demandé	répondu	routeErr	état
V	V	V	V	V	RI (Route Incorrecte)
V	V	V	V	F	RC (Route Correcte)
V	V	V	F	F	AR (Attente Réponse)
V	V	F	F	F	PAD (Pret A Demander)
V	F	F	F	F	PAR (Pret A Recevoir)
F	F	F	F	F	NA (Non Autorisé)

TAB. 2 – Table des conditions du Data Port In.

### Etapas 3.8 à 3.11 : obtention des transitions

Pour chacun des ports, on donne les préconditions (3.8) et les postconditions (3.9) en fonction des conditions trouvées auparavant.  $\mathcal{P}$  désigne les valeur des conditions avant communication et  $\mathcal{Q}$  leur valeur après communication.  $\forall$  signifie que la valeur de la condition n'importe pas et  $=$  qu'elle n'est pas modifiée par l'opération.

Plusieurs vérifications sont possibles. Tout d'abord, les préconditions doivent être cohérentes par rapport aux relations existant entre les conditions. Ainsi, une précondition ne peut réclamer que deux conditions  $c_1$  et  $c_2$  soient toutes les deux vraies si d'un autre côté, on sait (étape 3.3) que  $\vdash c_1 \Rightarrow \neg c_2$ . Il est d'autre part possible de restreindre certaines préconditions automatiquement. En conservant la relation ci-dessus, si la précondition est que  $c_1$  soit vraie et  $\forall$  pour  $c_2$ , alors on la restreint en signifiant que  $c_2$  doit être fausse. Une seconde vérification peut être faite en ce qui concerne le classement des ports dans les différents ensembles (3.1). Les opérations dont la précondition est  $\forall$  pour toutes les conditions doivent être dans O ou C, les autres dans OC ou CC. A ce niveau, il est donc possible de devoir revenir à l'étape 3.1 et de devoir redérouler l'agenda à partir de ce point. Un exemple de cela est proposé dans l'étude du DPO, annexe A.

En ce qui concerne les postconditions, les vérifications sont de même ordre. Après avoir donné d'éventuelles relations faisant intervenir les nouvelles conditions apparues dans les postconditions (c'est l'étape 3.10, identique à l'étape 3.3), il faut vérifier la cohérence des postconditions. Au niveau du classement dans les ensembles, les opérations ne modifiant la valeur d'aucune condition ( $\forall j \bullet \mathcal{Q}_k = \langle \dots, \Phi(C'_j), \dots \rangle, \Phi(C') = C'$ ) doivent être dans O ou OC, et les autres dans C ou CC. Comme pour les préconditions, il est donc possible de devoir revenir à l'étape 3.1.

Les préconditions et postconditions des opérations du DPI se trouvent ci-dessous avec les notations suivantes : a pour autorisé, r pour reçu, d pour demandé, rep pour répondu et rerr pour routeErr.

enable	a	r	d	rep	rerr
$\mathcal{P}$	$\forall$	$\forall$	$\forall$	$\forall$	$\forall$
$\mathcal{Q}$	V	=	=	=	=

in_data	a	r	d	rep	rerr
$\mathcal{P}$	V	F	$\forall$	$\forall$	$\forall$
$\mathcal{Q}$	=	V	=	=	=

ask_route	a	r	d	rep	rerr
$\mathcal{P}$	V	V	$\forall$	F	$\forall$
$\mathcal{Q}$	=	=	V	=	=

reply_route	a	r	d	rep	rerr
$\mathcal{P}$	V	V	V	F	$\forall$
$\mathcal{Q}$	=	=	=	V	$l=[]$

correct	a	r	d	rep	rerr
$\mathcal{P}$	V	V	V	V	F
$\mathcal{Q}$	=	F	F	F	=

wrong_route	a	r	d	rep	rerr
$\mathcal{P}$	V	V	V	V	V
$\mathcal{Q}$	=	F	F	F	=

L'idée générale pour l'obtention de l'automate est qu'un état abstrait est identifié par un ensemble de services que peut rendre le processus. Les transitions correspondent à des opérations qui sont possibles sous certaines conditions (et donc à partir de certains états ; nous avons ici les points de départ des transitions) et conduisent ou non à modifier l'état abstrait (c'est le cas des constructeurs qui font changer d'état abstrait ; nous avons ici les points d'arrivée des transitions).

Les préconditions décrivent l'ensemble des états dans lesquels une communication sur un port est possible ou non. Les éléments de  $C$  ou  $O$  ne sont pas préconditionnés et ceux de  $CC$  ou  $OC$  ne sont préconditionnés que par leur condition d'application définie plus tôt par le spécifieur.

Les postconditions décrivent l'effet des opérations sur la valeur des conditions et donc le changement d'état abstrait.

Il existe en général des *cas critiques* [PCR98b] et cet effet ne peut donc pas être exprimé uniquement en fonction des conditions d'état. Il est nécessaire d'utiliser de nouvelles conditions pour tester si le type de données correspondant au processus est ou non dans un état critique. Ces nouvelles conditions vont apparaître lorsque l'on cherche à exprimer les postconditions (3.9).

En ce qui concerne les transitions (3.11), les automates peuvent être construits séparément (un par opération), puis intégrés, en prenant dans l'ordre :

- état initiaux possibles du processus ainsi que l'ensemble des observateurs sans conditions
- chacun des observateurs avec conditions
- chacun des constructeurs sans conditions
- chacun des constructeurs avec conditions.

Les état initiaux indiquent dans quel(s) état(s) sont instanciés les processus.

Les observateurs sans conditions correspondent à des transitions d'un état vers lui-même (un observateur ne modifiant pas l'état abstrait) et sont possibles dans tous les états.

Les observateurs conditionnels ne sont autorisés qu'à partir des états ayant la bonne valeur de vérité pour la condition correspondante. Ils correspondent à des transitions de chacun de ces états vers eux mêmes.

L'idée générale pour obtenir les transitions correspondant aux constructeurs est la suivante : à l'aide d'un n-uplet représentant les valeurs des conditions avant une opération (répondant donc à sa précondition) et des postconditions, il est possible d'obtenir le n-uplet correspondant à la valeur des conditions après cette opération (postcondition).

Les transitions correspondant aux constructeurs non conditionnés permettent de passer d'un état à un autre mais n'ont pas de précondition associée.

Pour les trouver on procède comme suit :

- prendre un état  $e$  (n-uplet de booléens)
- en supposant que ce n-uplet corresponde aux valeurs de précondition de l'opération, trouver le n-uplet correspondant aux valeurs de postcondition et l'état  $f$  correspondant
- on a une transition entre  $e$  et  $f$
- recommencer avec un état non encore choisi

Chaque n-uplet correspondant à un et un seul état, on a donc une relation entre états représentant les transitions de l'automate.

Les constructeurs conditionnés permettent de passer d'un état à un autre lorsque certaines préconditions sont vérifiées. Pour les trouver on procède comme pour les constructeurs non conditionnés mais les nouvelles conditions apparaissant au niveau des postconditions sont reportées en tant que préconditions sur les transitions. Une notation particulière est utilisée au niveau des transitions. Une transition d'étiquette  $[c] \ e \ [c']$  représente une communication de nom  $e$  conditionnée par deux préconditions :  $c$  concernant l'état du processus (données internes) avant la communication, et  $c'$  concernant des contraintes sur les données reçues lors de la communication. C'est à rapprocher de la différence entre garde et précondition en LOTOS.

Dans certains cas, les conditions inhérentes aux états impliquent celles des préconditions et postconditions (voir [PCR98b] pour plus de détails) et simplifient ainsi les tests à faire.

Cette démarche automatique permet de repérer des cas qui peuvent ne pas être détectés sans suivre la méthode. Ces cas correspondent aux cas critiques.

### Etape 3.12 : état initial

Pour déterminer l'état *initial*, on examine les services (qui sont associés à des ports) que le processus doit rendre ou non (contraintes). A partir des préconditions des ports et de la table de création des états on trouve les états susceptibles d'être initiaux. S'il n'y en a pas, c'est que le spécifieur a donné des contraintes non cohérentes pour son état initial. En ce qui concerne la spécification LOTOS, il est nécessaire de n'avoir qu'un seul état initial (état dans lequel "démarré" le processus lors de son instanciation). Dans le cas où il y a plusieurs état initiaux possibles, un choix peut être fait en ajoutant une contrainte sur les services ou en faisant un choix arbitraire.

L'automate de DPI est donné dans la figure 12. On peut remarquer que, comme signalé plus haut, les conditions de `reply_route` concernent les données reçues lors de cette communication et sont donc représentées après le nom de l'opération dans les transitions.

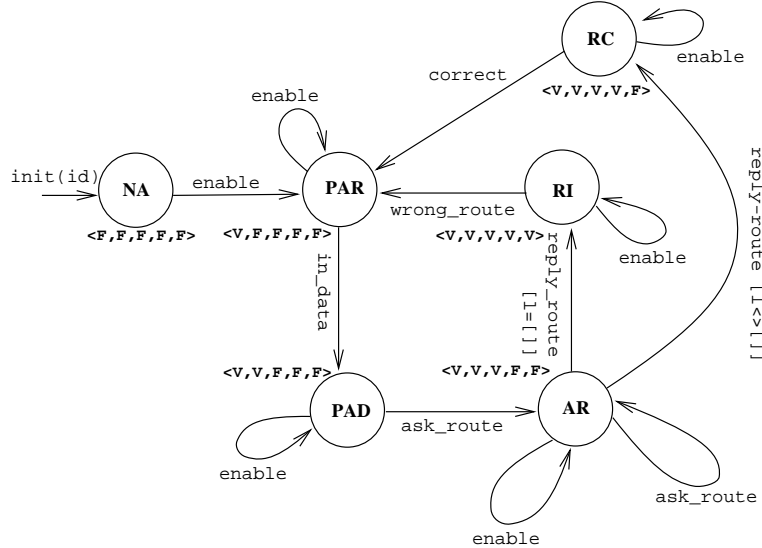


Figure 12 : Automate du Data Port In.

### Etales 3.13 à 3.15 : simplifications et traduction

Il est possible de traduire l'automate vers divers langages de spécification en lui appliquant des schémas de traduction. Cette technique a été appliquée à LOTOS [PCR98b] et est ici appliquée à SDL.

En préalable, il est nécessaire de parler des différences les plus importantes entre ces deux langages, c'est à dire celle qui concernent la communication.

En LOTOS, les communications sont en théorie proposées à tout l'environnement du processus émetteur (proche du `via all` de SDL), mais il est possible d'utiliser l'émission de valeurs particulières pour remédier à cette impossibilité (4ème ligne de la table 3). En SDL, il est possible de communiquer directement entre processus particuliers de façon naturelle. Pour cela, chaque processus dispose à sa création d'un identifiant particulier (type de données PID). Un processus peut ensuite s'adresser à un processus donné en utilisant des mots-clés comme `sender` (le dernier processus à lui avoir envoyé un signal), `parent` (le processus qui l'a créé), ...

En SDL, contrairement à LOTOS, les processus ne se synchronisent pas. L'émission est non bloquante. La réception passe par un buffer ou sont placés les messages (signaux) dans l'ordre de leur arrivée. Lorsqu'un processus est en attente, le premier signal du buffer est dépilé. S'il ne correspond à aucun des signaux qu'attendait le receveur, alors ce signal est éliminé (sauf cas particulier ou le receveur peut explicitement demander à ce qu'il soit conservé). Si aucun signal attendu n'est dans le buffer, le receveur se bloque.

Nous pensons toutefois que les communications en LOTOS sont plus générales, c'est pourquoi nous utilisons une notation qui lui est empruntée (voir la table 3). Une fois les différences prises en compte, il est possible d'appliquer une série de schémas de traduction aux automates construits à l'aide de notre méthode.

Tout d'abord, à un automate correspond un processus SDL. Le cadre du processus est construit (figure 13) et comprend : la déclaration du processus (**Process P**), la déclaration du paramètre formel correspondant à son type propre (**fpar paramP : tP**) et enfin la déclaration de son type propre (**dcl a\_tP tP**). Le type **tP** devra disposer d'une opération **init : S<sub>i</sub> -> tP**.

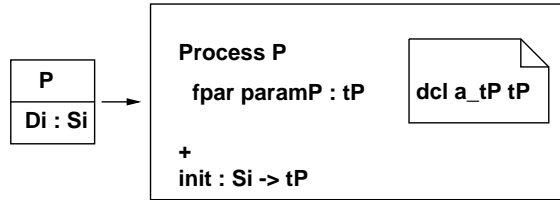


Figure 13 : Schéma du cadre des processus.

La transition initiale est construite en utilisant le schéma de la figure 14a. L'instanciation se fait comme donné dans la figure 14b, c'est à dire en instanciant le processus à l'aide de l'opération **init**.

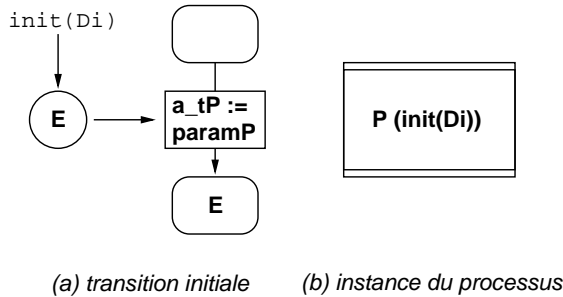


Figure 14 : Schéma de transition initiale.

La traduction des transitions non conditionnées se fait selon le schéma de la figure 15a. La prise en compte des conditions au niveau de la traduction de fait en utilisant des conditions d'acceptation et des symboles de décision (figure 15b). Les offres correspondant aux transitions sont traduites en utilisant les schémas des figures suivantes.

Quatre type d'offres existent (comme en LOTOS) et sont indiquées dans la table 3. Les offres multiples sont découpées en offres simples.

La dernière catégorie correspond à un cas particulier en LOTOS et se traduit en SDL par l'emploi du mot-clé **to**.

Le traduction de la première catégorie d'offre est donnée dans la figure 16a, et celle de la seconde catégorie (symétrique) dans la figure 16b. Il faut noter que dans le cas d'une réception il est nécessaire de déclarer une variable du bon type. Ce n'est pas le cas des émissions car les données émises sont en fait des fonctions sur le type paramètre du processus (**a\_tP**).

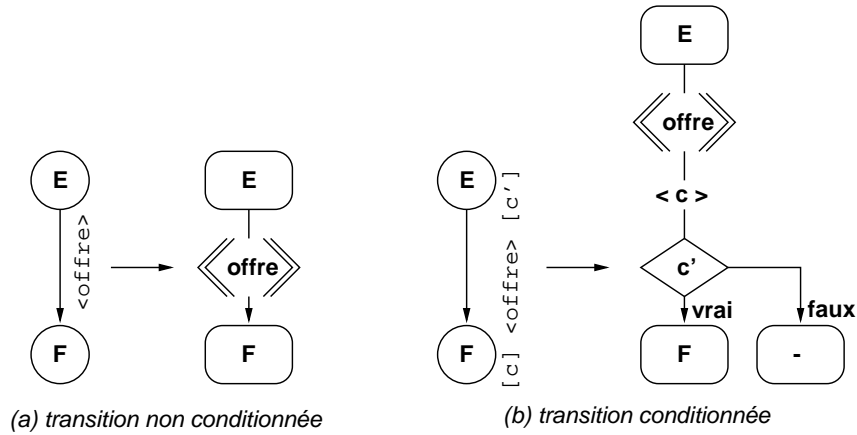


Figure 15 : Schéma de traduction des transitions.

processus 1	processus 2	interprétation par rapport au processus 1
$p?x:T$	$p!x:T$	réception de valeur
$p!x:T$	$p?x:T$	émission de valeur
$p?x:T$	$p?x:T$	partage d'une valeur prise "au hasard" dans T (indéterminisme)
$p!x:T$	$p!x:T$	synchronisation sur valeur particulière (permet en LOTOS de s'adresser à un processus particulier)

TAB. 3 – Différents types d'offres.

Pour pouvoir traduire la troisième catégorie d'offre, il est nécessaire de lui attribuer un "sens", c'est à dire de choisir un processus qui calculera la valeur  $x$  à l'aide du mot-clé `any` et qui sera chargé de l'émettre au second processus (figure 17).

La traduction peut ne pas être optimale et c'est pourquoi des simplifications doivent éventuellement lui être appliquées. Des règles concernant la simplification des spécifications engendrées en LOTOS peuvent être trouvées dans [PCR98b].

En ce qui concerne SDL, l'utilisation des listes d'états ou de l'étoile dans les états de début des transitions ainsi que du tiret dans les états de fin des transitions permet de structurer la description des processus par rapport aux transitions (signaux reçus) et non pas des états. Cette technique peut par exemple être appliquée avec succès aux membres de O (figure 18) : ces communications étant possibles depuis tous les états, le regroupement par signal permet de regrouper toutes les transitions  $\circ$  en une seule transition SDL.

Lors de la traduction des offres, nous avons découpé les offres multiples en offres simples. Du fait du typage des signaux en SDL, il faut créer autant de signaux que d'offres simples. Il est en fait possible de regrouper des séries d'émissions (resp. réceptions) les unes à la suite des autres et de ne modifier le type paramètre du processus qu'à la fin des réceptions. Cette modification peut même être omise si elle n'est pas pertinente (comme par exemple un processus qui récupérerait deux nombres et émettrait le minimum, figure 19)

L'application de ce type de simplification à l'offre  $p?x:T?y:T!z:U$  (où  $z$  dépendrait des valeurs de  $x$ ,  $y$  et du type paramètre) est donné en figure 20a. A des fins de comparaison, ce que nous aurions obtenu sans simplifier est donné en figure 20b.

En amont, l'automate lui-même peut parfois être simplifié selon des techniques identiques ou proches. [And95] propose ainsi trois techniques de simplifications :

- regroupement des transitions similaires, c'est à dire ayant même opération, même condition

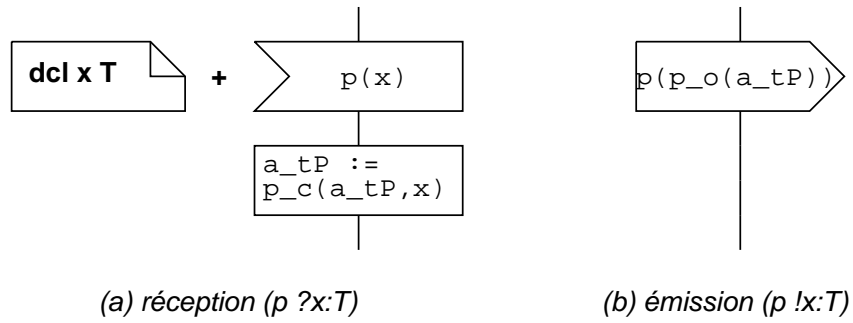


Figure 16 : Schéma de traduction des offres d’émission/réception.

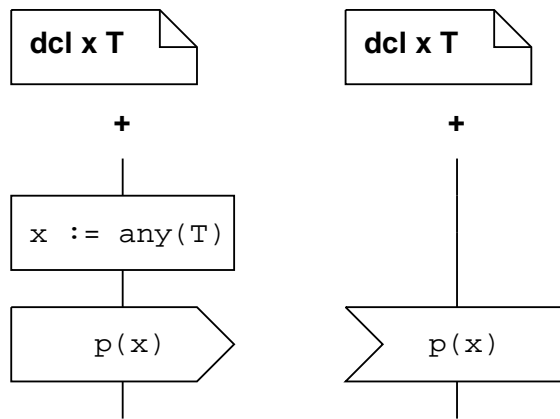


Figure 17 : Schéma de traduction des offres  $?x:T$  /  $!x:T$ .

et des destinations identiques (les transitions en boucle sont un cas particulier de transitions similaires où l’on considère que la destination identique est “-” au sens de SDL), en agrégeant les états d’origine.

- composition d’états : les états fortement connexes sont regroupés et des boucles conditionnées remplacent les transitions initialement existantes entre les états de départ. Un état composé est en fait un sous-automate.
- composition de transitions : des notations particulières pour des transitions dites “séquentielles” (deux transitions telles que l’état  $e$  de destination de l’une soit l’état d’origine de l’autre et que le degré de  $e$  soit 2) et des transitions dites “parallèles” (même origine et même destination) simplifient l’automate.

La figure 21 donne une partie du processus SDL correspondant au DPI.

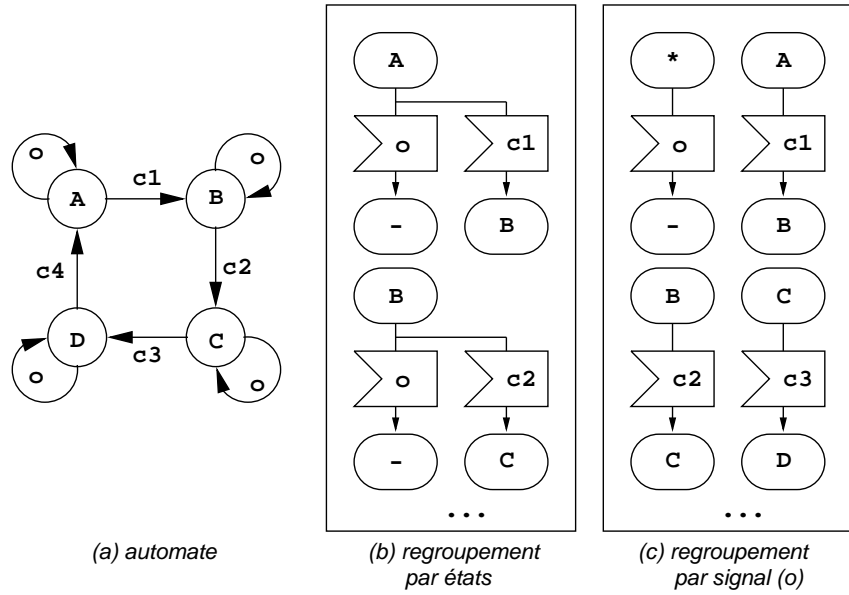


Figure 18 : Comparaison entre regroupements.

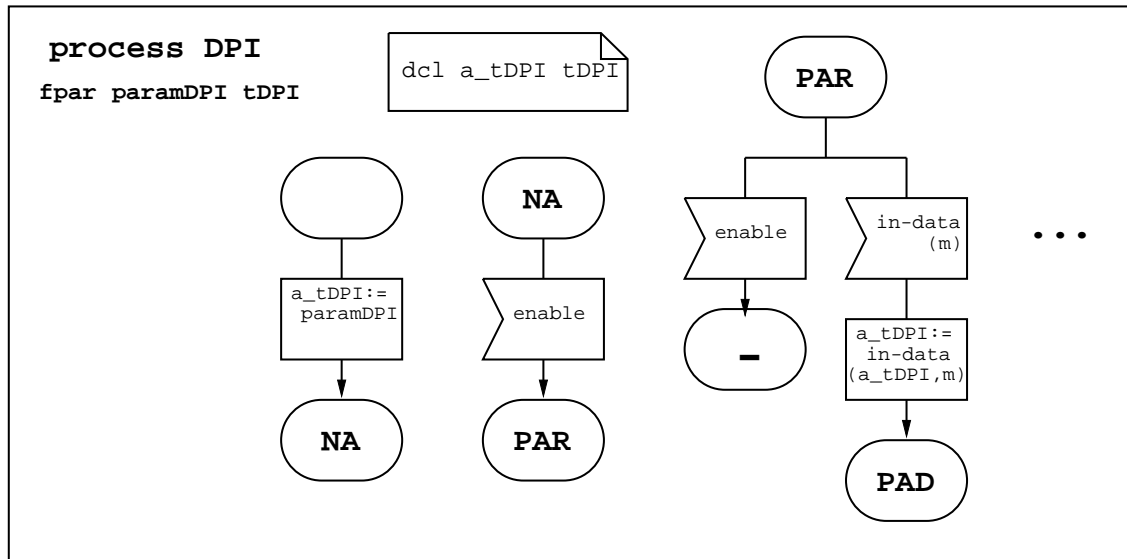


Figure 21 : Une partie du processus SDL du Data Port In.

## Etape 4 : Types de données

La dernière étape est de construire la spécification des types de données associés à chaque processus, ainsi que celle des types de données manipulées par les processus. En ce qui concerne le type de données associé à chaque processus, le travail effectué au cours des étapes précédentes fournit l'essentiel de la signature (cf. [PCR98b] pour la description du traitement automatique). En effet, les noms des opérations et leurs profils sont obtenus automatiquement à partir du typage des communications, et à partir des différentes conditions identifiées en construisant les automates. Quelques opérations supplémentaires peuvent être nécessaires pour exprimer les axiomes.

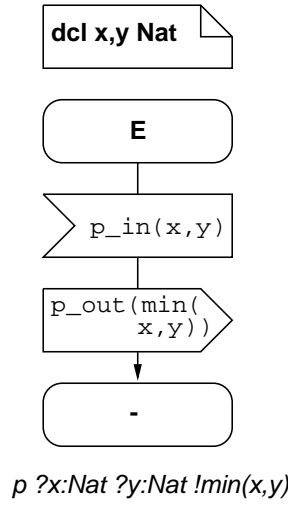


Figure 19 : Calcul de minimum.

La plupart des axiomes<sup>3</sup> sont écrits dans un style “constructif” ce qui nécessite d’avoir identifié les générateurs. [AR97] développe une méthode pour obtenir le type abstrait associé à un automate et pour calculer un ensemble minimal d’opérations nécessaire pour atteindre tous les états. Dans notre exemple, cet ensemble est `init`, `enable`, `in_data`, `ask_route`, `reply_route`. [AR97] utilise l’ $\Omega$ -dérivation [Bid82] pour écrire les axiomes (qui sont des équations conditionnelles). Pour extraire les axiomes décrivant les propriétés d’une opération `op(dpi:DPI)`, on recherche les états où cette opération peut être effectuée ainsi que les générateurs permettant d’atteindre ces états, ce qui permet d’obtenir les prémisses et les membres gauches des axiomes.

Nous montrons ici une partie de ce processus automatique pour les axiomes de l’opération `correct_c`<sup>4</sup>. Cette opération est associée à la transition `correct` qui est possible dans l’état `RC`, qui est atteignable via les compositions de générateurs qui apparaissent en partie gauche des axiomes ci-dessous. Les prémisses expriment les conditions sur les états de départ et sur la valeur de la variable `l`.

```
RC(dpi) => correct_c(enable(dpi)) = correct_c(dpi)
AR(dpi) /\ not(l=[]) => correct_c(reply_route(ask_route(dpi),l)) = correct_c(dpi)
AR(dpi) /\ not(l=[]) => correct_c(reply_route(enable(dpi),l)) = correct_c(dpi)
PAD(dpi) /\ not(l=[]) => correct_c(reply_route(ask_route(enable(dpi)),l)) = correct_c(dpi)
PAR(dpi) /\ not(l=[]) => correct_c(reply_route(ask_route(in_data(dpi,m)),l)) = dpi
```

La spécification algébrique peut ensuite être utilisée pour effectuer des preuves de propriétés à des fins de vérification et de validation de la spécification.

## 4 Conclusion

Bien qu’il soit maintenant admis que les spécifications formelles offrent des avantages dans le développement logiciel l’absence de méthodes et d’outils en freine l’utilisation courante. Un autre aspect important est de pouvoir lier les aspects dynamiques, souvent exprimés par des diagrammes d’états, et les aspects statiques, souvent exprimés par des types de données.

Nous proposons dans cet article l’agenda d’une méthode formelle permettant de spécifier les aspects dynamiques et statiques d’une façon cohérente et assistée. La méthode présentée est un cadre générale qui permet de spécifier dans des langages comme LOTOS ou SDL. La méthode

<sup>3</sup>. à l’exception des formules exprimant les relations entre conditions

<sup>4</sup>. La spécification complète est donnée en annexe.



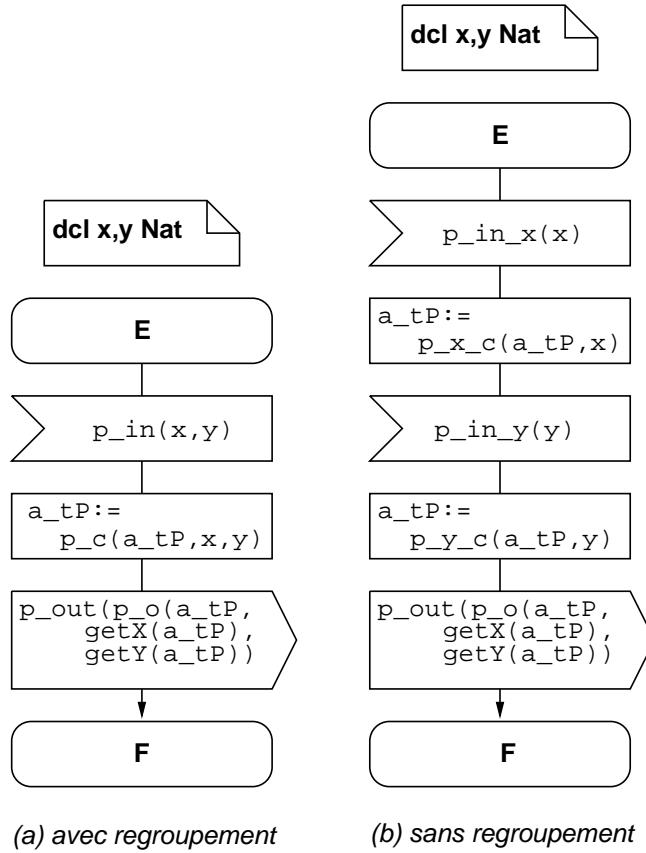


Figure 20 : Regroupement d'offres simples.

s'inspire, à la fois, des approches orientées contraintes, orientées ressources et orientées états que l'on rencontre dans les méthodes LOTOS.

Le premier pas est de définir une architecture parallèle pour le système. Ensuite le comportement de chaque composant est décrit par un processus séquentiel auquel est associé un type de données. L'étude des communications et de leurs effets sur le type de données permet de définir un automate du comportement dynamique du processus d'une manière quasi-automatique. L'automate est ensuite traduit en comportements LOTOS ou SDL à l'aide de schémas prédéfinis. La spécification algébrique des types de données est elle-aussi extraite à l'aide de l'automate de façon semi-automatique. La méthode permet de décomposer simplement les processus et d'avoir des types de données cohérents avec le comportement dynamique du processus. La méthode est utilisée avec comme langage cible LOTOS et SDL mais elle est adaptable à d'autres formalismes. Des expérimentations ont été faites avec Estelle et nous prévoyons d'étendre ces expériences à Raise.

Notre méthode a été utilisée sur des exemples de taille réduite mais elle doit être testée sur des études de cas grandeur nature.

## Références

- [And95] Pascal André. *Méthodes formelles et à objets pour le développement du logiciel : Etudes et propositions*. PhD Thesis, Université de Rennes I (Institut de Recherche en Informatique de Nantes), Juillet 1995.

- [AR97] Pascal André and Jean-Claude Royer. How To Easily Extract an Abstract Data Type From a Dynamic Description. Research Report 159, Institut de Recherche en Informatique de Nantes, September 1997. <http://www.sciences.univ-nantes.fr/info/perso/permanents/andre/PUBLI/rr159.ps.gz>.
- [BB88] Tommaso Bolognesi and Ed Brinksma. Introduction to the ISO Specification Language LOTOS. *Computer Networks and ISDN Systems*, 14(1):25–29, January 1988.
- [BCV94] M. Bidoit, C. Choppy, and F. Voisin. Validation d’une spécification algébrique du “Transit-node” par prototypage et démonstration de théorèmes. Chapitre du Rapport final de l’Opération VTT, Validation et vérification de propriétés Temporelles et de Types de données (commune aux PRC PAOIA et C3), LaBRI, Bordeaux, 1994.
- [Bid82] Michel Bidoit. Types abstraits algébriques : spécifications structurées et présentations gracieuses. In *Colloque AFCET, Les mathématiques de l’informatique*, pages 347–357, Mars 1982.
- [EHS97] Jan Ellsberger, Dieter Hogrefe, and Amardeo Sarma. *SDL : Formal Object-oriented Language for Communicating Systems*. Prentice-Hall, 1997.
- [EM85] H. Ehrig and B. Mahr. *Fundamentals of Algebraic Specification*, volume 1. Springer-Verlag, Berlin, 1985.
- [Gar90] Hubert Garavel. Introduction au langage LOTOS. Technical report, VERILOG, Centre d’Etudes Rhône-Alpes, Forum du Pré Milliet, Montbonnot, 38330 Saint-Ismier, 1990.
- [GG89] Stephan Garland and John Guttag. An overview of LP, the Larch Prover. In *Proc. of the Third International Conference on Rewriting Techniques and Applications*, volume 355 of *Lecture Notes in Computer Science*, pages 137–151. Springer-Verlag, 1989.
- [GHD98] Wolfgang Grieskamp, Maritta Heisel, and Heiko Dörr. Specifying Embedded Systems with Statecharts and Z: An Agenda for Cyclic Software Components. In Egidio Astesiano, editor, *FASE’98*, volume 1382 of *Lecture Notes in Computer Science*, pages 88–106. Springer-Verlag, 1998.
- [Hei98] Maritta Heisel. Agendas – A Concept to Guide Software Development Activities. In R. N. Horspool, editor, *Proceedings Systems Implementation 2000*, pages 19–32. Chapman & Hall, 1998.
- [HL97] Maritta Heisel and Nicole Lévy. Using LOTOS Patterns to Characterize Architectural Styles. In Michel Bidoit and Max Dauchet, editors, *TAPSOFT’97 (FASE’97)*, volume 1214 of *Lecture Notes in Computer Science*, pages 818–832, 1997.
- [Hoa78] C.A.R. Hoare. Communicating Sequential Processes. *Communications of the ACM*, 21(8):666–677, August 1978.
- [HS97] Maritta Heisel and Carsten Suhl. Methodological Support for Formally Specifying Safety-Critical Software. In P. Daniel, editor, *16th International Conference on Computer Safety, Reliability and Security (SAFECOMP)*, pages 295–308. Springer-Verlag, 1997.
- [ISO98] ISO. Final Commite Draft on Enhancements to LOTOS. <ftp://ftp.dit.upm.es/pub/lotos/elotos/Working.Docs/ELOTOSfed.ps>, April 1998.
- [Led87] G. J. Leduc. LOTOS, un outil utile ou un autre langage académique? In *Actes des Neuvièmes Journées Francophones sur l’Informatique — Les réseaux de communication — Nouveaux outils et tendances actuelles (Liège)*, Janvier 1987.

- [LLS97] Thomas Lambolais, Nicole Lévy, and Jeanine Souquière. Assistance au développement de spécifications de protocoles de communication. In *AFADL'97 Approches Formelles dans l'Assistance au Développement de Logiciel*, pages 73–84, 1997.
- [Mil80] Robin Milner. *A Calculus of Communicating Systems*, volume 92 of *Lecture Notes in Computer Science*. Springer-Verlag, Berlin, 1980.
- [Mou94] Laurent Mounier. A LOTOS Specification of a “Transit-Node”. Technical Report 94-8, SPECTRE, VERIMAG, 1994.
- [PCR98a] Pascal Poizat, Christine Choppy, and Jean-Claude Royer. Concurrency and Data Types: a Specification Method. An Example with LOTOS. In J. Fiadeiro, editor, *Recent Trends in Algebraic Development Techniques, Selected Papers of the 13th International Workshop WADT98*, Lecture Notes in Computer Science. Springer-Verlag, 1998. to appear.
- [PCR98b] Pascal Poizat, Christine Choppy, and Jean-Claude Royer. Une nouvelle méthode pour la spécification en LOTOS. Rapport de Recherche 170, Institut de Recherche en Informatique de Nantes, Février 1998. <http://www.sciences.univ-nantes.fr/info/perso/permanents/poizat/papers/rr170.ps.gz>.
- [Tur93] Kenneth J. Turner, editor. *Using Formal Description Techniques, an introduction to Estelle, Lotos and SDL*. Wiley, 1993.
- [Tur97] K. Turner. Relating architecture and specification. *Computer Networks and ISDN Systems*, 29(4):437–456, 1997.
- [VSVSB91] C.A. Vissers, G. Scollo, M. Van Sinderen, and E. Brinksma. Specification styles in distributed systems design and verification. *Theoretical Computer Science*, (89):179–206, 1991.

## A Etude du Data Port Out

Le typage des communications du DPO a été étudié et est le suivant :

- `out_data` : !m:Msg
- `correct` : !ident:PortNumber?m:Msg
- `enable` : !ident:PortNumber
- `timeout` : !m:Msg

L'application des étapes 3.1 et 3.2 au DPO donne les résultats suivants :

- $O : \emptyset$ ,
- $C : \{\text{enable}\}$ ,
- $OC : \emptyset$ ,
- $CC : \{\text{out\_data} : \text{autorisé}, \text{correct} : \text{autorisé}, \text{timeout} : \text{autorisé}\}$

La table des états du DPO est donnée en table 4.

autorisé	état
V	A (autorisé)
F	$\bar{A}$ (non autorisé)

TAB. 4 – Table des conditions du Data Port Out.

Les préconditions et postconditions des opérations du DPO se trouvent ci-dessous.

enable	autorisé
$\mathcal{P}$	$\forall$
$\mathcal{Q}$	V

out_data	autorisé
$\mathcal{P}$	V
$\mathcal{Q}$	=

correct	autorisé
$\mathcal{P}$	V
$\mathcal{Q}$	=

timeout	autorisé
$\mathcal{P}$	V
$\mathcal{Q}$	=

On peut ici remarquer que les préconditions et postconditions ne correspondent pas au classement des opérations. Ainsi, au regard de ces tables, `out_data`, `correct` et `timeout` devraient se trouver dans OC et non pas dans CC. Pourtant, intuitivement nous les avons bien placées dans CC.

En fait, nous avons omis la condition concernant le fait que le buffer du DPO soit vide ou non. Cette condition influe sur la possibilité d'effectuer ou non des émissions de message (`out_cmde` et `timeout`). D'autre part, les opérations d'émission et de réception de message (`correct`) vont modifier sa valeur.

Nous recommençons donc l'application de l'agenda à partir de l'étape 3.1.

L'application des étapes 3.1 et 3.2 au DPO donne les résultats suivants :

- $O : \emptyset$ ,
- $C : \{\text{enable}\}$ ,
- $OC : \emptyset$ ,
- $CC : \{\text{out\_data} : \text{autorisé} \wedge \neg \text{vide}, \text{correct} : \text{autorisé}, \text{timeout} : \text{autorisé} \wedge \neg \text{vide}\}$

autorisé	vide	état
V	V	$AV$
V	F	$A\bar{V}$
F	V	$\bar{A}V$
F	F	$\bar{A}\bar{V}$

TAB. 5 – Table révisée des conditions du Data Port Out.

La table des états du DPO est donnée en table 5.

Les préconditions et postconditions des opérations du DPO se trouvent ci-dessous.

enable	autorisé	vide	out_data	autorisé	vide
$\mathcal{P}$	$\forall$	$\forall$	$\mathcal{P}$	V	F
$\mathcal{Q}$	V	=	$\mathcal{Q}$	=	unSeulMsg

correct	autorisé	vide	timeout	autorisé	vide
$\mathcal{P}$	V	$\forall$	$\mathcal{P}$	V	F
$\mathcal{Q}$	=	F	$\mathcal{Q}$	=	unSeulMsg

L'automate de DPO est donné dans la figure 22. L'état  $\bar{A}\bar{V}$  est supprimé car il n'est pas atteignable. A ce niveau, nous trouvons la formule caractéristique de cet état. Elle est obtenue à partir de la ligne de la table des conditions qui lui correspond :  $\neg\text{autorisé} \wedge \neg\text{vide}$ . Puisque cet état n'est pas atteignable, nous prenons la négation de cette formule, ce qui donne : autorisé  $\vee$  vide. Cette formule est ensuite proposée au spécifieur en tant qu'ajout aux formules de l'étape 3.3. Si le spécifieur valide la formule, il est possible de redérouler l'agenda à partir de cette étape (phases de validation essentiellement). Si le spécifieur ne valide pas cette formule (ou si elle est incohérente par rapport aux autres formules), c'est qu'une incohérence dans sa spécification a été détectée.

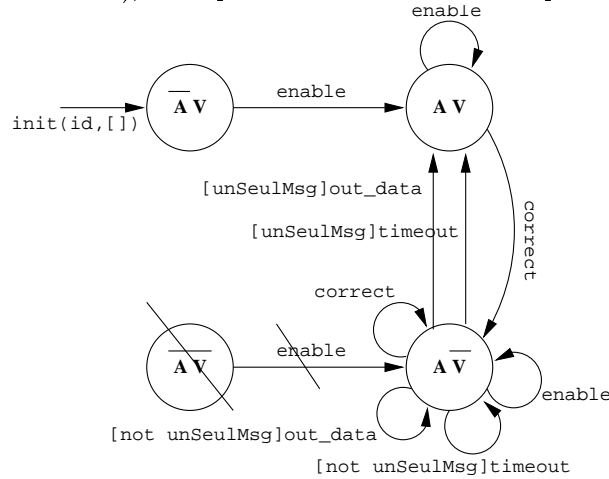


Figure 22 : Automate du Data Port Out.

## B Etude du Control Port In

Le typage des communications du CPI a été étudié et est le suivant :

- `in_cmde : ?m:Msg`
- `enable : !n:PortNumber`
- `sendfault`
- `wrong_cmde : !m:Msg`
- `ask_route : ?r:RouteNumber`
- `reply_route : !r:RouteNumber !l:list[PortNumber]`

L'application des étapes 3.1 et 3.2 au CPI donne les résultats suivants :

- `O :  $\emptyset$ ,`
- `C :  $\emptyset$ ,`
- `OC :  $\emptyset$ ,`
- `CC : {in_cmde :  $\neg$ reçu, ask_route :  $\neg$ demandé, reply_route : demandé, sendfault : reçu  $\wedge$  code=Send-Faults, enable : reçu  $\wedge$  code=Add-Data-Port-In- $\mathcal{E}$ -Out, wrong_cmde : reçu  $\wedge$  code $\notin$  {Send-Faults, Add-Data-Port-In- $\mathcal{E}$ -Out, Add-Route}}`

La condition de `wrong_cmde` fait apparaitre la nécessité d'un évènement interne correspondant à la reception d'un message/transition interne d'ajout de route. Nous ajoutons donc à CC : `i_add_route : reçu  $\wedge$  code=Add-Route.`

Nous utiliserons les notations suivantes.

- `code=Send-Faults :  $c_{sf}$`  (pour `sendfault`)
- `code=Add-Data-Port-In- $\mathcal{E}$ -Out :  $c_{en}$`  (pour `enable`)
- `code=Add-Route :  $c_i$`
- `code $\notin$  {Send-Faults, Add-Data-Port-In- $\mathcal{E}$ -Out, Add-Route} :  $c_{wc}$`  (pour `wrong_cmde`)

Nous avons la relation suivante : `reçu  $\Leftrightarrow c_{sf} \oplus c_{en} \oplus c_i \oplus c_{wc}$ .`

La table des états du CPI est donnée en table 6.

$c_{sf}$	$c_{en}$	$c_i$	$c_{wc}$	reçu	demandé	état
F	F	F	V	V	V	WC-DEM
F	F	F	V	V	F	WC-ATT
F	F	V	F	V	V	I-DEM
F	F	V	F	V	F	I-ATT
F	V	F	F	V	V	EN-DEM
F	V	F	F	V	F	EN-ATT
V	F	F	F	V	V	SF-DEM
V	F	F	F	V	F	SF-ATT
F	F	F	F	F	V	ATT-DEM
F	F	F	F	F	F	ATT-ATT

TAB. 6 – Table des conditions du Control Port In.

Ces relations permettent de simplifier les conditions des opérations de CC comme suit :

$\{in\_cmde: \neg \text{reçu}, ask\_route: \neg \text{demandé}, reply\_route: \text{demandé}, sendfault: c_{sf}, enable: c_{en}, wrong\_cmde: c_{wc}, i\_add\_route: c_i\}$

Les préconditions et postconditions des opérations du CPI se trouvent ci-dessous (en utilisant  $r$  pour reçu et  $d$  pour demandé). En ce qui concerne les postconditions de  $in\_cmde?m:Msg$ , nous utilisons des raccourcis,  $c_{sf}$  désignant par exemple en fait le fait que le message  $m$  reçu est de type *Send-Faults* soit :  $is\_sendfault(m)$ .

$in\_cmde$	$c_{sf}$	$c_{en}$	$c_i$	$c_{wc}$	$r$	$d$
$\mathcal{P}$	F	F	F	F	F	$\forall$
$\mathcal{Q}$	$c_{sf}$	$c_{en}$	$c_i$	$c_{wc}$	V	=

$ask\_route$	$c_{sf}$	$c_{en}$	$c_i$	$c_{wc}$	$r$	$d$
$\mathcal{P}$	$\forall$	$\forall$	$\forall$	$\forall$	$\forall$	F
$\mathcal{Q}$	=	=	=	=	=	V

$reply\_route$	$c_{sf}$	$c_{en}$	$c_i$	$c_{wc}$	$r$	$d$
$\mathcal{P}$	$\forall$	$\forall$	$\forall$	$\forall$	$\forall$	V
$\mathcal{Q}$	=	=	=	=	=	F

$sendfault$	$c_{sf}$	$c_{en}$	$c_i$	$c_{wc}$	$r$	$d$
$\mathcal{P}$	V	F	F	F	V	$\forall$
$\mathcal{Q}$	F	F	F	F	F	=

$enable$	$c_{sf}$	$c_{en}$	$c_i$	$c_{wc}$	$r$	$d$
$\mathcal{P}$	F	V	F	F	V	$\forall$
$\mathcal{Q}$	F	F	F	F	F	=

$i\_add\_route$	$c_{sf}$	$c_{en}$	$c_i$	$c_{wc}$	$r$	$d$
$\mathcal{P}$	F	F	V	F	V	$\forall$
$\mathcal{Q}$	F	F	F	F	F	=

$wrong\_cmde$	$c_{sf}$	$c_{en}$	$c_i$	$c_{wc}$	$r$	$d$
$\mathcal{P}$	F	F	F	V	V	$\forall$
$\mathcal{Q}$	F	F	F	F	F	=

L'automate de CPI est donné dans la figure 23.

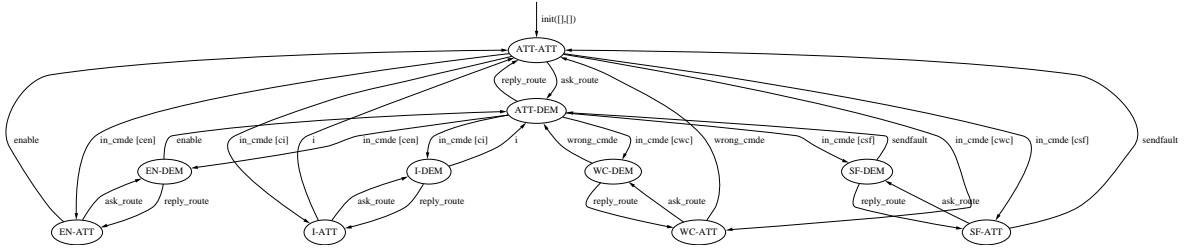


Figure 23 : Automate du Control Port In.

## C Etude de la Faulty Collection

Le typage des communications de la FC a été étudié et est le suivant :

- `timeout : ?m:Msg`
- `wrong_route : ?m:Msg`
- `wrong_cmde : ?m:Msg`
- `sendfault`
- `route_errors : !l:List[Msg]`

Nous emploirons le terme `erreurs` pour représenter `timeout`, `wrong_route` et `wrong_cmde`. L'application des étapes 3.1 et 3.2 à la FC donne les résultats suivants :

- $O : \emptyset$ ,
- $C : \emptyset$ ,
- $OC : \{\text{erreurs} : \neg \text{reçu}\}$ ,
- $CC : \{\text{sendfault} : \neg \text{reçu}, \text{route\_errors} : \text{reçu}\}$

La table des états de la FC est donnée en table 7.

reçu	état
V	SF (sendfault reçu)
F	$\overline{SF}$ (sendfault non reçu)

TAB. 7 – Table des conditions de la Faulty Collection.

Les préconditions et postconditions des opérations de la FC se trouvent ci-dessous.

sendfault	reçu
$\mathcal{P}$	F
$\mathcal{Q}$	V

erreurs	reçu
$\mathcal{P}$	F
$\mathcal{Q}$	=

route_errors	reçu
$\mathcal{P}$	V
$\mathcal{Q}$	F

L'automate de la FC est donné dans la figure 24a et sa version simplifiée dans la figure 24b. La simplification a consisté à réunir les deux transitions qui constituaient un “tunnel”, c’est à dire qui reliaient un état de degré 2.

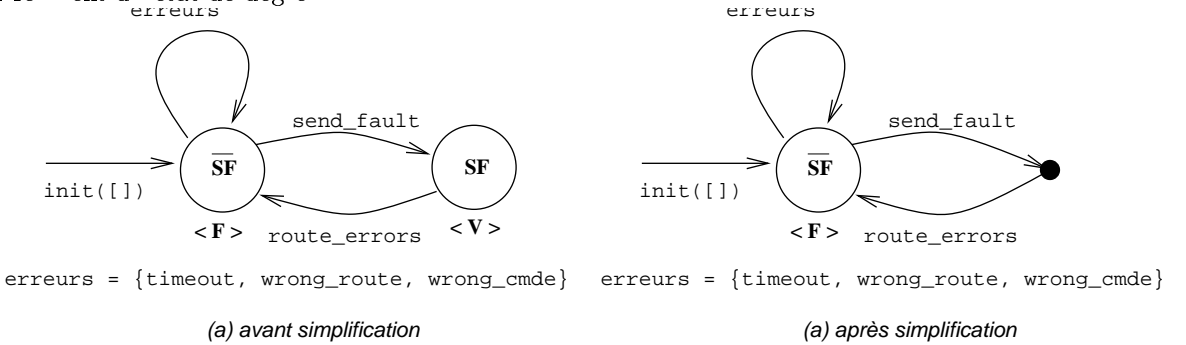


Figure 24 : Automate de la Faulty Collection.



## D Etude du Control Port Out

Le typage des communications du CPO a été étudié et est le suivant :

- `out_cmde : !m:Msg`
- `route_errors : ?l:List[Msg]`

L'application des étapes 3.1 et 3.2 au CPO donne les résultats suivants :

- $O : \emptyset$ ,
- $C : \{\text{route\_errors}\}$ ,
- $OC : \emptyset$ ,
- $CC : \{\text{out\_cmde} : \neg \text{vide}\}$

La table des états du CPO est donnée en table 8.

vide	état
V	Vide
F	$\overline{\text{Vide}}$

TAB. 8 – Table des conditions du Control Port Out.

Les préconditions et postconditions des opérations du CPO se trouvent ci-dessous.

route_errors	vide
$\mathcal{P}$	$\forall$
$\mathcal{Q}$	$l=[] \wedge \text{vide}$

out_cmde	vide
$\mathcal{P}$	F
$\mathcal{Q}$	presqueVide

L'automate du CPO est donné dans la figure 25a et sa version simplifiée dans la figure 25b. La simplification a consisté à réunir dans une seule transition les transitions de même origine, même destination et même étiquette.

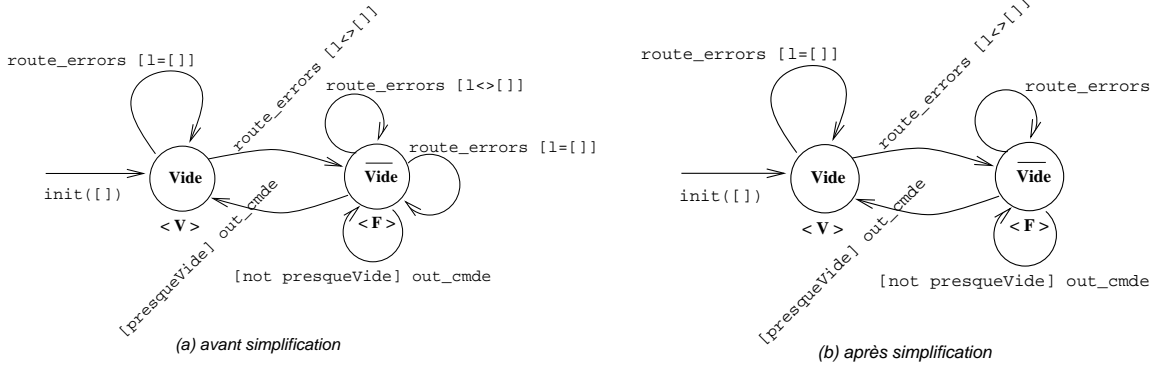


Figure 25 : Automate du Control Port Out.

## E Spécification algébrique des types de données

### Data Port In

Résultats de la dérivation:

```
type DPI

% GENERATEURS
init : Nat -> DPI
enable : DPI -> DPI
in_data : DPI x MSG -> DPI
ask_route : DPI -> DPI
reply_route : DPI x LIST[NAT] -> DPI

% CONDITIONS DE BASE
enabled : DPI -> BOOL
  recu : DPI -> BOOL
  demande : DPI -> BOOL
  repondu : DPI -> BOOL
routeerr : DPI -> BOOL

% CONDITIONS D'ETAT
DIS : DPI -> BOOL
PAR : DPI -> BOOL
PAD : DPI -> BOOL
  AR : DPI -> BOOL
  CR : DPI -> BOOL
  WR : DPI -> BOOL

% AUTRES
correct_c : DPI -> DPI
wrong_route_c : DPI -> DPI
correct_o : DPI -> MSG
wrong_route_o : DPI -> MSG

% RELATIONS ENTRE CONDITIONS
recu(dpi) => enabled(dpi)
demande(dpi) => recu(dpi)
repondu(dpi) => enabled(dpi)
routeerr(dpi) => repondu(dpi)

% CONDITIONS DE BASE
enabled(init(dpi,id)) = false
enabled(enable(dpi)) = true
enabled(in_data(dpi,m)) = enabled(dpi)
enabled(ask_route(dpi)) = enabled(dpi)
enabled(reply_route(dpi,l)) = enabled(dpi)

recu(init(dpi,id)) = false
recu(enable(dpi)) = recu(dpi)
recu(in_data(dpi,m)) = true
recu(ask_route(dpi)) = recu(dpi)
recu(reply_route(dpi,l)) = recu(dpi)
```

```

demande(init(dpi,id)) = false
demande(enable(dpi)) = demande(dpi)
demande(in_data(dpi,m)) = demande(dpi)
demande(ask_route(dpi)) = true
demande(reply_route(dpi,l)) = demande(dpi)

repondu(init(dpi,id)) = false
repondu(enable(dpi)) = repondu(dpi)
repondu(in_data(dpi,m)) = repondu(dpi)
repondu(ask_route(dpi)) = repondu(dpi)
repondu(reply_route(dpi,l)) = true

routeerr(init(dpi,id)) = false
routeerr(enable(dpi)) = routeerr(dpi)
routeerr(in_data(dpi,m)) = routeerr(dpi)
routeerr(ask_route(dpi)) = routeerr(dpi)
routeerr(reply_route(dpi,l)) = (l=[])

% CONDITIONS D'ETAT
DIS(dpi) = not(enabled(dpi)) /\ not(recu(dpi)) /\ not(demande(dpi)) /\ not(repondu(dpi)) /\ not(r
PAR(dpi) =     enabled(dpi) /\ not(recu(dpi)) /\ not(demande(dpi)) /\ not(repondu(dpi)) /\ not(r
PAD(dpi) =     enabled(dpi) /\      recu(dpi) /\ not(demande(dpi)) /\ not(repondu(dpi)) /\ not(r
  AR(dpi) =     enabled(dpi) /\      recu(dpi) /\      demande(dpi) /\ not(repondu(dpi)) /\ not(r
  CR(dpi) =     enabled(dpi) /\      recu(dpi) /\      demande(dpi) /\      repondu(dpi) /\ not(r
  WR(dpi) =     enabled(dpi) /\      recu(dpi) /\      demande(dpi) /\      repondu(dpi) /\      r

% AUTRES
CR(dpi) => correct_c(enable(dpi)) = correct_c(dpi)
AR(dpi) /\ not(l=[]) => correct_c(reply_route(ask_route(dpi),l)) = correct_c(dpi)
AR(dpi) /\ not(l=[]) => correct_c(reply_route(enable(dpi),l)) = correct_c(dpi)
PAD(dpi) /\ not(l=[]) => correct_c(reply_route(ask_route(enable(dpi)),l)) = correct_c(dpi)
PAR(dpi) /\ not(l=[]) => correct_c(reply_route(ask_route(in_data(dpi,m)),l)) = dpi

CR(dpi) => correct_o(enable(dpi)) = correct_o(dpi)
AR(dpi) /\ not(l=[]) => correct_o(reply_route(ask_route(dpi),l)) = correct_o(dpi)
AR(dpi) /\ not(l=[]) => correct_o(reply_route(enable(dpi),l)) = correct_o(dpi)
PAD(dpi) /\ not(l=[]) => correct_o(reply_route(ask_route(enable(dpi)),l)) = correct_o(dpi)
PAR(dpi) /\ not(l=[]) => correct_o(reply_route(ask_route(in_data(dpi,m)),l)) = m

CR(dpi) => wrong_route_c(enable(dpi)) = wrong_route_c(dpi)
AR(dpi) /\ not(l=[]) => wrong_route_c(reply_route(ask_route(dpi),l)) = wrong_route_c(dpi)
AR(dpi) /\ not(l=[]) => wrong_route_c(reply_route(enable(dpi),l)) = wrong_route_c(dpi)
PAD(dpi) /\ not(l=[]) => wrong_route_c(reply_route(ask_route(enable(dpi)),l)) = wrong_route_c(dpi)
PAR(dpi) /\ not(l=[]) => wrong_route_c(reply_route(ask_route(in_data(dpi,m)),l)) = dpi

CR(dpi) => wrong_route_o(enable(dpi)) = wrong_route_o(dpi)
AR(dpi) /\ not(l=[]) => wrong_route_o(reply_route(ask_route(dpi),l)) = wrong_route_o(dpi)
AR(dpi) /\ not(l=[]) => wrong_route_o(reply_route(enable(dpi),l)) = wrong_route_o(dpi)
PAD(dpi) /\ not(l=[]) => wrong_route_o(reply_route(ask_route(enable(dpi)),l)) = wrong_route_o(dpi)
PAR(dpi) /\ not(l=[]) => wrong_route_o(reply_route(ask_route(in_data(dpi,m)),l)) = m

```