

Cours ASPD
Protocoles de Communications
Telecom Nancy 2A Apprentissage

Dominique Méry
Telecom Nancy
Université de Lorraine

Année universitaire 2024-2025
2 février 2025

Summary

- ① Communications entre processus
- ② Observation et modélisation
- ③ Modélisation de protocoles de communication
- ④ Protocole de Stenning
- ⑤ Sliding Window Protocol

Section Courante

- ① Communications entre processus]
- ② Observation et modélisation
- ③ Modélisation de protocoles de communication
- ④ Protocole de Stenning
- ⑤ Sliding Window Protocol

Problèmes des communications

- La communication de données entre deux entités est en général non fiable
- Le support physique de communication peut perdre, dupliquer, réordonnancer ou détériorer les messages
- Un protocole de communication est une méthode permettant de communiquer des données, en veillant à détecter et à corriger les éventuelles erreurs de transmission
- Modèles d'architectures en couches : des couches basses du réseaux physiques aux couches les plus hautes réalisant les services.
- Modèles d'architecture : OSI et TCP/IP
- Modèles d'empilement des couches avec une relation de raffinement ou de simulation entre une couche supérieure et une couche inférieure : un service de niveau n est simulé par la couche de niveau $n-1$

Modèles de référence OSI

- OSI signifie *Open Systems Interconnection*
- Modèle de Référence proposé par l'organisation mondiale de normalisation ISO
- Ce modèle concerne la connexion entre systèmes ouverts à la communication avec d'autres systèmes ouverts
- Le modèle OSI a sept couches :
 - ▶ Une couche correspond à un nouveau d'abstraction pour les communications
 - ▶ Chaque couche possède et effectue des fonctions spécifiques
 - ▶ Les fonctions de chaque couche sont choisies en fonction de la définition de protocoles normalisés internationaux
 - ▶ Le choix des frontières entre les couches doit minimiser le flux d'informations aux interfaces
 - ▶ Le nombre de couches doit être rationnel et permettre de maîtriser l'architecture et d'éviter la cohabitation dans une même couche de fonctions très différentes.

- **Couche Physique** : elle gère la transmission des bits qui peuvent être altérés par des problèmes de transmission physique.
- **Couche Liaison de données** : elle gère les communications sous forme de trames de données et assure donc la communication entre deux entités, en visant à corriger les problèmes du niveau inférieur (gestion de trames en séquences et des trames d'acquittements) ; elle assure aussi une régulation des émetteurs.
- **Couche réseau** : elle gère le sous-réseau, en particulier elle gère les routes ; elle gère aussi les congestions de ce sous-réseau.
- **Couche transport** : elle assure le découpage des données de la couche session et ls passe à la couche réseau. Elle s'assure que les morceaux arrivent correctement au sens des couches supérieures.
- **Couche session** : elle permet d'établir des sessions par les utilisateurs et donc d'utiliser la couche transport : transfert de fichiers par exemple.
- **Couche présentation** : elle concerne la syntaxe et la sémantique de l'information transmise
- **Couche application** : elle compose les protocoles développés

Modèle TCP/IP

- Couche hôte-réseau ou liens de données : connexion de l'hôte à la couche Internet via un protocole permettant d'envoyer des paquets IP ; driver du système d'exploitation et d'une carte d'interface de l'ordinateur aux réseaux.
- Couche Réseau ou Internet : elle permet l'acheminement de paquets dans n'importe quel réseau et dans n'importe quel ordre ; les questions de réacheminement sont réglées par les couches supérieures. Le format est celui du protocole IP.
- Couche transport : elle permet à des paires ou entités connectées deux à deux, de maintenir une conversation ou une communication ; deux protocoles ont été définis :
 - ▶ le protocole TCP qui assure l'acheminement fiable d'un flux d'octets à une autre entité et
 - ▶ le protocole UDP qui est un protocole non-fiable, sans connexion, pour les applications qui ne veulent pas de séquences ou de contrôles de flux.
- Couche Application : elle contient les protocoles de haut niveau comme FTP, TELNET ; SMTP, DNS, SNMP

Section Courante

- ① Communications entre processus]
- ② Observation et modélisation
- ③ Modélisation de protocoles de communication
- ④ Protocole de Stenning
- ⑤ Sliding Window Protocol

- Modélisation par échange de messages :
 - ▶ des actions internes ou locales à un processus
 - ▶ des actions de communications : envoi ou réception
- Canaux de communication :
 - ▶ tout message envoyé est reçu fatalement
 - ▶ tout message envoyé est reçu fatalement mais pas dans l'ordre d'envoi
 - ▶ tout message peut être perdu
- Variables partagées
- communication synchrone ou asynchrone
- modélisation en TLA^+ à partir d'une description des actions en Event-B.

Choix de modélisation

- Donner le « quoi » : spécification de ce que fait le protocole

Choix de modélisation

- Donner le « quoi » : spécification de ce que fait le protocole
 - ▶ envoi d'un message m par un processus P à un processus Q

- Donner le « quoi » : spécification de ce que fait le protocole
 - ▶ envoi d'un message m par un processus P à un processus Q
 - ▶ décomposition en plusieurs phases

- Donner le « quoi » : spécification de ce que fait le protocole
 - ▶ envoi d'un message m par un processus P à un processus Q
 - ▶ décomposition en plusieurs phases
- Donner le « comment » : simulation du protocole par des événements et des phases des couches plus basses
- Modélisation par raffinement à partir du service attendu pour mettre en œuvre dans les couches plus basses ou concrètes

Observation d'un système réparti

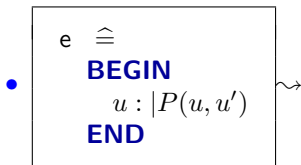
- $u_0 \xrightarrow{e_0} u_1 \xrightarrow{e_1} \dots \xrightarrow{e_{i-1}} u_i \xrightarrow{e_i} u_{i+1} \xrightarrow{e_{i+1}} \dots$
- e_0 ou e_1 ou \dots ou e_{i-1} ou e_i ou e_{i+1} ou \dots
- $e \in \{e_0, e_1, \dots, e_{i-1}, e_i, e_{i+1}, \dots\}$
- $e \in E$: E est l'ensemble fini des actions ou des événements observés sur le système modifiant l'état courant.
- $u_0 \xrightarrow{g} \dots \xrightarrow{f} u \xrightarrow{e} u' \xrightarrow{g} \dots$
- Chaque événement modélise la transformation d'une liste de variables d'états appelées *frame* et notée u :

if $cond(u)$ **then** $u := f(u)$ **fi**

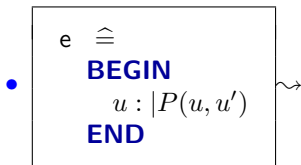
Non-déterminisme et entrelacement

Les événements de E sont observés les uns à la suite des autres en veillant à ce qu'un événement est observé quand sa *garde* est vraie. On peut ajouter une hypothèse d'équité sur la trace produite.

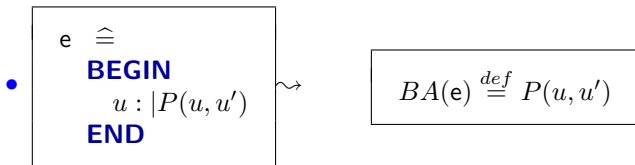
Événements



Événements



Événements



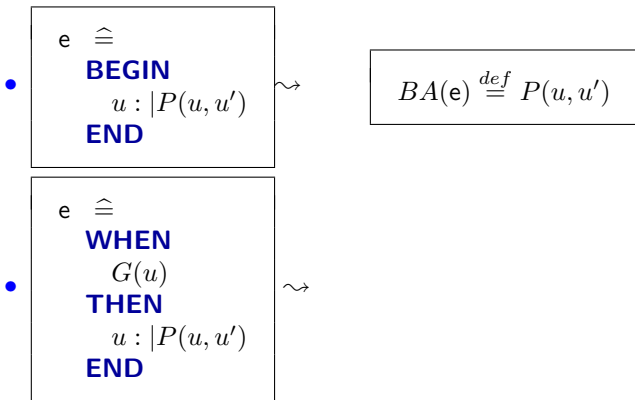
Événements

- $$\begin{array}{l} e \hat{=} \\ \text{BEGIN} \\ u : |P(u, u') \\ \text{END} \end{array} \rightsquigarrow$$

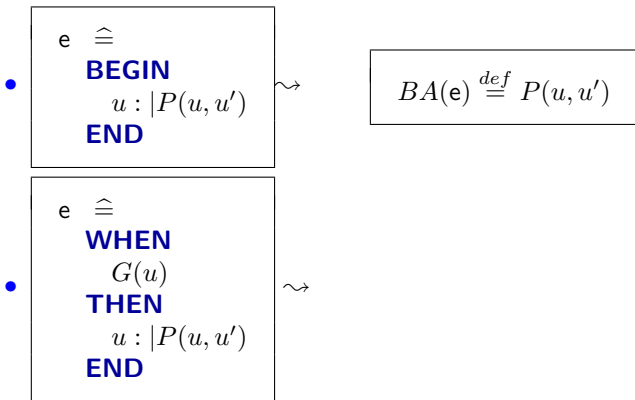
$$BA(e) \stackrel{def}{=} P(u, u')$$

- $$\begin{array}{l} e \hat{=} \\ \text{WHEN} \\ G(u) \\ \text{THEN} \\ u : |P(u, u') \\ \text{END} \end{array}$$

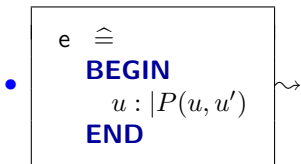
Événements



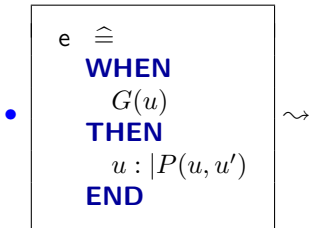
Événements



Événements



$$BA(e) \stackrel{def}{=} P(u, u')$$



$$BA(e) \stackrel{def}{=} \left(\begin{array}{l} \wedge G(u) \\ \wedge u' = f(u) \end{array} \right)$$

Événements

- $$e \hat{=} \begin{array}{l} \text{BEGIN} \\ u : |P(u, u') \\ \text{END} \end{array} \rightsquigarrow$$

$$BA(e) \stackrel{def}{=} P(u, u')$$

- $$e \hat{=} \begin{array}{l} \text{WHEN} \\ G(u) \\ \text{THEN} \\ u : |P(u, u') \\ \text{END} \end{array} \rightsquigarrow$$

$$BA(e) \stackrel{def}{=} \left(\begin{array}{l} \wedge G(u) \\ \wedge u' = f(u) \end{array} \right)$$

- $$e \hat{=} \begin{array}{l} \text{ANY } x \text{ WHERE} \\ G(u, x) \\ \text{THEN} \\ u : |P(x, u, u') \\ \text{END} \end{array}$$

Événements

- $$\begin{array}{l} e \hat{=} \\ \text{BEGIN} \\ u : |P(u, u') \\ \text{END} \end{array}$$

\rightsquigarrow

$$BA(e) \stackrel{def}{=} P(u, u')$$

- $$\begin{array}{l} e \hat{=} \\ \text{WHEN} \\ G(u) \\ \text{THEN} \\ u : |P(u, u') \\ \text{END} \end{array}$$

\rightsquigarrow

$$BA(e) \stackrel{def}{=} \left(\begin{array}{l} \wedge G(u) \\ \wedge u' = f(u) \end{array} \right)$$

- $$\begin{array}{l} e \hat{=} \\ \text{ANY } x \text{ WHERE} \\ G(u, x) \\ \text{THEN} \\ u : |P(x, u, u') \\ \text{END} \end{array}$$

\rightsquigarrow

Événements

- $$\begin{array}{l} e \hat{=} \\ \text{BEGIN} \\ u : |P(u, u') \\ \text{END} \end{array}$$

\rightsquigarrow

$$BA(e) \stackrel{def}{=} P(u, u')$$

- $$\begin{array}{l} e \hat{=} \\ \text{WHEN} \\ G(u) \\ \text{THEN} \\ u : |P(u, u') \\ \text{END} \end{array}$$

\rightsquigarrow

$$BA(e) \stackrel{def}{=} \left(\begin{array}{l} \wedge G(u) \\ \wedge u' = f(u) \end{array} \right)$$

- $$\begin{array}{l} e \hat{=} \\ \text{ANY } x \text{ WHERE} \\ G(u, x) \\ \text{THEN} \\ u : |P(x, u, u') \\ \text{END} \end{array}$$

\rightsquigarrow

Événements

- $$\begin{array}{l} e \hat{=} \\ \text{BEGIN} \\ u : |P(u, u') \\ \text{END} \end{array} \rightsquigarrow$$

$$BA(e) \stackrel{def}{=} P(u, u')$$

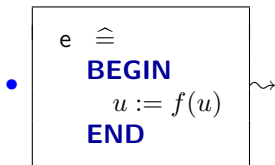
- $$\begin{array}{l} e \hat{=} \\ \text{WHEN} \\ G(u) \\ \text{THEN} \\ u : |P(u, u') \\ \text{END} \end{array} \rightsquigarrow$$

$$BA(e) \stackrel{def}{=} \left(\begin{array}{l} \wedge G(u) \\ \wedge u' = f(u) \end{array} \right)$$

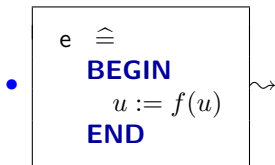
- $$\begin{array}{l} e \hat{=} \\ \text{ANY } x \text{ WHERE} \\ G(u, x) \\ \text{THEN} \\ u : |P(x, u, u') \\ \text{END} \end{array} \rightsquigarrow$$

$$BA(e) \stackrel{def}{=} \exists x. \left(\begin{array}{l} \wedge G(x, u) \\ \wedge u' = f(x, u) \end{array} \right)$$

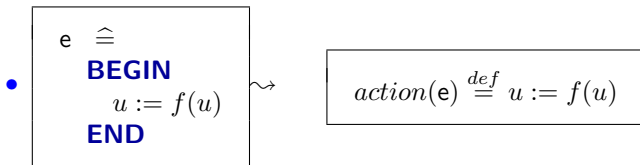
Systèmes d'actions



Systèmes d'actions



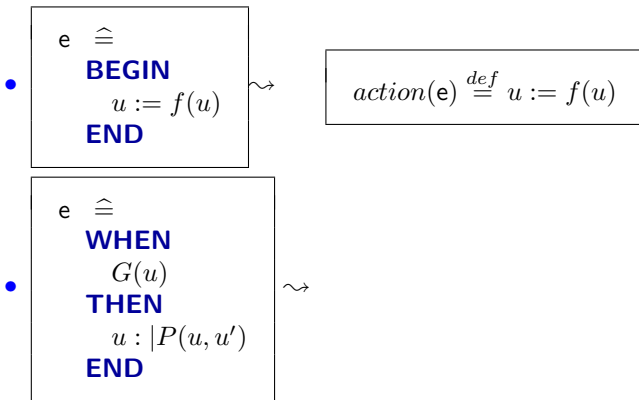
Systèmes d'actions



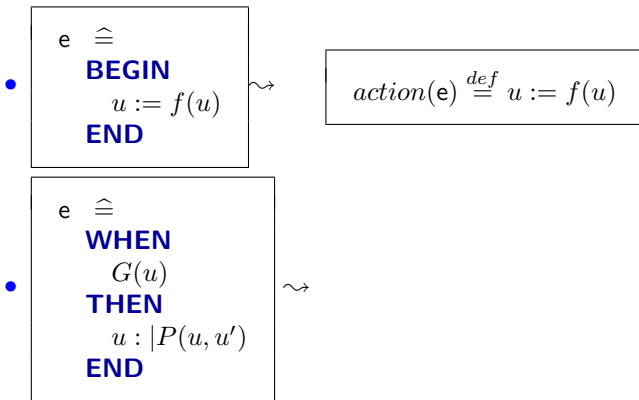
Systèmes d'actions

- $$e \hat{=} \begin{array}{l} \text{BEGIN} \\ \quad u := f(u) \\ \text{END} \end{array} \rightsquigarrow \boxed{action(e) \stackrel{def}{=} u := f(u)}$$
- $$e \hat{=} \begin{array}{l} \text{WHEN} \\ \quad G(u) \\ \text{THEN} \\ \quad u : | P(u, u') \\ \text{END} \end{array}$$

Systèmes d'actions



Systèmes d'actions



Systèmes d'actions

- $$\begin{array}{l} e \hat{=} \\ \text{BEGIN} \\ \quad u := f(u) \\ \text{END} \end{array} \rightsquigarrow$$

$$action(e) \stackrel{def}{=} u := f(u)$$

- $$\begin{array}{l} e \hat{=} \\ \text{WHEN} \\ \quad G(u) \\ \text{THEN} \\ \quad \quad u : |P(u, u') \\ \text{END} \end{array} \rightsquigarrow$$

$$action(e) \stackrel{def}{=} \text{IF } G(u) \text{ THEN } u := f(u)$$

Systèmes d'actions

- $$e \hat{=} \begin{array}{l} \text{BEGIN} \\ u := f(u) \\ \text{END} \end{array} \rightsquigarrow$$

$$action(e) \stackrel{def}{=} u := f(u)$$

- $$e \hat{=} \begin{array}{l} \text{WHEN} \\ G(u) \\ \text{THEN} \\ u : |P(u, u') \\ \text{END} \end{array} \rightsquigarrow$$

$$action(e) \stackrel{def}{=} \text{IF } G(u) \text{ THEN } u := f(u)$$

- $$e \hat{=} \begin{array}{l} \text{ANY } x \text{ WHERE} \\ G(u, x) \\ \text{THEN} \\ u : |P(x, u, u') \\ \text{END} \end{array}$$

Systèmes d'actions

- $$e \hat{=} \begin{array}{l} \text{BEGIN} \\ u := f(u) \\ \text{END} \end{array} \rightsquigarrow$$

$$action(e) \stackrel{def}{=} u := f(u)$$

- $$e \hat{=} \begin{array}{l} \text{WHEN} \\ G(u) \\ \text{THEN} \\ u : |P(u, u') \\ \text{END} \end{array} \rightsquigarrow$$

$$action(e) \stackrel{def}{=} \text{IF } G(u) \text{ THEN } u := f(u)$$

- $$e \hat{=} \begin{array}{l} \text{ANY } x \text{ WHERE} \\ G(u, x) \\ \text{THEN} \\ u : |P(x, u, u') \\ \text{END} \end{array} \rightsquigarrow$$

Systèmes d'actions

- $$e \hat{=} \begin{array}{l} \text{BEGIN} \\ u := f(u) \\ \text{END} \end{array} \rightsquigarrow$$

$$action(e) \stackrel{def}{=} u := f(u)$$

- $$e \hat{=} \begin{array}{l} \text{WHEN} \\ G(u) \\ \text{THEN} \\ u : |P(u, u') \\ \text{END} \end{array} \rightsquigarrow$$

$$action(e) \stackrel{def}{=} \text{IF } G(u) \text{ THEN } u := f(u)$$

- $$e \hat{=} \begin{array}{l} \text{ANY } x \text{ WHERE} \\ G(u, x) \\ \text{THEN} \\ u : |P(x, u, u') \\ \text{END} \end{array} \rightsquigarrow$$

Systèmes d'actions

- $$\begin{array}{l} e \hat{=} \\ \text{BEGIN} \\ \quad u := f(u) \\ \text{END} \end{array} \rightsquigarrow$$

$$action(e) \stackrel{def}{=} u := f(u)$$

- $$\begin{array}{l} e \hat{=} \\ \text{WHEN} \\ \quad G(u) \\ \text{THEN} \\ \quad u : |P(u, u') \\ \text{END} \end{array} \rightsquigarrow$$

$$action(e) \stackrel{def}{=} \text{IF } G(u) \text{ THEN } u := f(u)$$

- $$\begin{array}{l} e \hat{=} \\ \text{ANY } x \text{ WHERE} \\ \quad G(u, x) \\ \text{THEN} \\ \quad u : |P(x, u, u') \\ \text{END} \end{array} \rightsquigarrow$$

$$action(e) \stackrel{def}{=} \text{IF } G(x, u) \text{ THEN } u$$

Appel et rappel

Appel et rappel

- Traduction d'un événement en une relation TLA^+ :

Appel et rappel

- Traduction d'un événement en une relation TLA^+ :

EVENT e

ANY

x

WHERE

$G(x, u)$

THEN

$u := f(x, u)$

END

Appel et rappel

- Traduction d'un événement en une relation TLA^+ :

EVENT e

ANY

x

WHERE

$G(x, u)$

THEN

$u := f(x, u)$

END

\leadsto

Appel et rappel

- Traduction d'un événement en une relation TLA^+ :

EVENT e

ANY

x

WHERE

$G(x, u)$

THEN

$u := f(x, u)$

END

\rightsquigarrow

Appel et rappel

- Traduction d'un événement en une relation TLA^+ :

EVENT e

ANY

x

WHERE

$G(x, u)$

THEN

$u := f(x, u)$

END

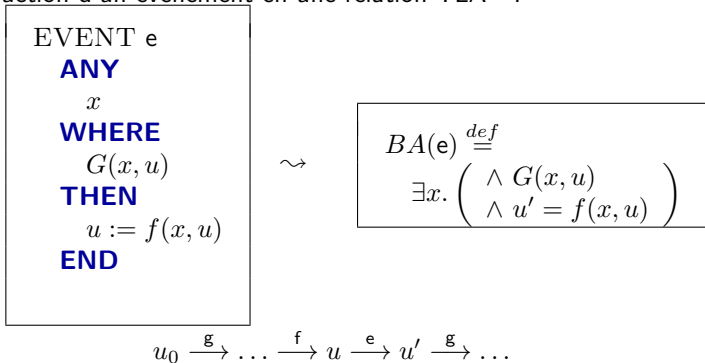
\rightsquigarrow

$BA(e) \stackrel{def}{=}$

$\exists x. \left(\begin{array}{l} \wedge G(x, u) \\ \wedge u' = f(x, u) \end{array} \right)$

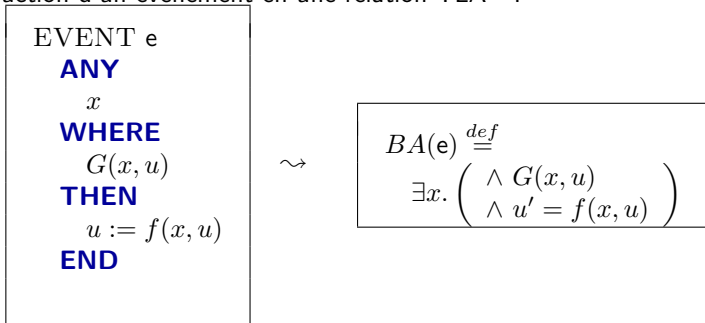
Appel et rappel

- Traduction d'un événement en une relation TLA^+ :



Appel et rappel

- Traduction d'un événement en une relation TLA^+ :

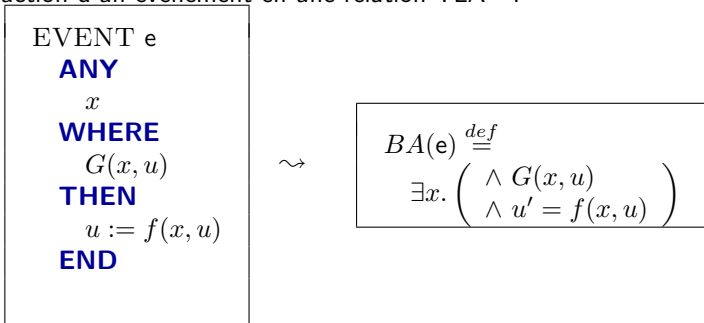


$$u_0 \xrightarrow{g} \dots \xrightarrow{f} u \xrightarrow{e} u' \xrightarrow{g} \dots$$

- Quand la condition $G(x, u)$ est vraie pour une valeur x , alors la transformation $u := f(x, u)$ met à jour la variable u .

Appel et rappel

- Traduction d'un événement en une relation TLA^+ :

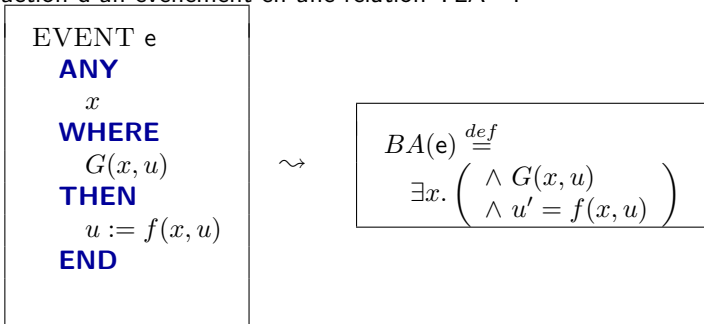


$$u_0 \xrightarrow{g} \dots \xrightarrow{f} u \xrightarrow{e} u' \xrightarrow{g} \dots$$

- Quand la condition $G(x, u)$ est vraie pour une valeur x , alors la transformation $u := f(x, u)$ met à jour la variable u .
- On dit aussi que l'événement e est observé.

Appel et rappel

- Traduction d'un événement en une relation TLA^+ :

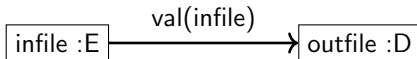


$$u_0 \xrightarrow{g} \dots \xrightarrow{f} u \xrightarrow{e} u' \xrightarrow{g} \dots$$

- Quand la condition $G(x, u)$ est vraie pour une valeur x , alors la transformation $u := f(x, u)$ met à jour la variable u .
- On dit aussi que l'événement e est observé.
- Modélisation par entrelacement non-déterministe des événements possibles : un seul événement possible est observable.

Modélisation du protocole de communication

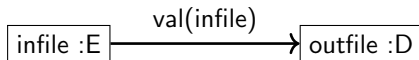
- La valeur d'un fichier *infile* est transmis d'un émetteur E à un destinataire D



- Le nombre de données du fichier est n et ces données sont des éléments de $DATA$
 - ▶ $DATA$ est un ensemble d données
 - ▶ $infile$ est un fichier de longueur n :
 - $n \in \mathbb{N}_1$
 - $infile \in 1 .. n \rightarrow DATA$

Modélisation du protocole

- Spécification du service attendu par les deux partenaires E et D : transmission du fichier *infile* de E via un canal de communication et réception dans un fichier *outfile* de D .



- *outfile* est une variable localisée en D
- $inv1 : outfile \in 1 .. n \leftrightarrow DATA$: pendant la transmission, *outfile* contient une partie des données et n'a pas tout reçu.

Modélisation du protocole

EVENT INITIALISATION

BEGIN

$act1 : outfile : \in 1 .. n \mapsto DATA$

END

EVENT communication

BEGIN

$act1 : outfile := infile$

END

EVENT protocoling

BEGIN

$act1 : outfile : \in 1 .. n \mapsto DATA$

END

- Initialement, le fichier *outfile* contient n'importe quoi.
- L'événement communication réalise le service attendu en un coup !
- L'événement protocoling modélise que tout peut arriver et en particulier, le fichier transmis peut être corrompu et la communication peut être mauvaise.

Observation des communications

Le protocole de communication est construit selon des hypothèses de l'environnement :

Modélisation du protocole

EVENT INITIALISATION

BEGIN

$act1 : outfile : \in 1 .. n \mapsto DATA$

END

EVENT communication

BEGIN

$act1 : outfile := infile$

END

EVENT protocoling

BEGIN

$act1 : outfile : \in 1 .. n \mapsto DATA$

END

- Initialement, le fichier *outfile* contient n'importe quoi.
- L'événement communication réalise le service attendu en un coup !
- L'événement protocoling modélise que tout peut arriver et en particulier, le fichier transmis peut être corrompu et la communication peut être mauvaise.

Observation des communications

Le protocole de communication est construit selon des hypothèses de l'environnement : hostilité,

Modélisation du protocole

EVENT INITIALISATION

BEGIN

$act1 : outfile : \in 1 .. n \mapsto DATA$

END

EVENT communication

BEGIN

$act1 : outfile := infile$

END

EVENT protocoling

BEGIN

$act1 : outfile : \in 1 .. n \mapsto DATA$

END

- Initialement, le fichier *outfile* contient n'importe quoi.
- L'événement communication réalise le service attendu en un coup !
- L'événement protocoling modélise que tout peut arriver et en particulier, le fichier transmis peut être corrompu et la communication peut être mauvaise.

Observation des communications

Le protocole de communication est construit selon des hypothèses de l'environnement : hostilité, fautes,

Modélisation du protocole

EVENT INITIALISATION

BEGIN

$act1 : outfile : \in 1 .. n \mapsto DATA$

END

EVENT communication

BEGIN

$act1 : outfile := infile$

END

EVENT protocoling

BEGIN

$act1 : outfile : \in 1 .. n \mapsto DATA$

END

- Initialement, le fichier *outfile* contient n'importe quoi.
- L'événement communication réalise le service attendu en un coup !
- L'événement protocoling modélise que tout peut arriver et en particulier, le fichier transmis peut être corrompu et la communication peut être mauvaise.

Observation des communications

Le protocole de communication est construit selon des hypothèses de l'environnement : hostilité, fautes, pertes,

Modélisation du protocole

EVENT INITIALISATION

BEGIN

$act1 : outfile : \in 1 .. n \mapsto DATA$

END

EVENT communication

BEGIN

$act1 : outfile := infile$

END

EVENT protocoling

BEGIN

$act1 : outfile : \in 1 .. n \mapsto DATA$

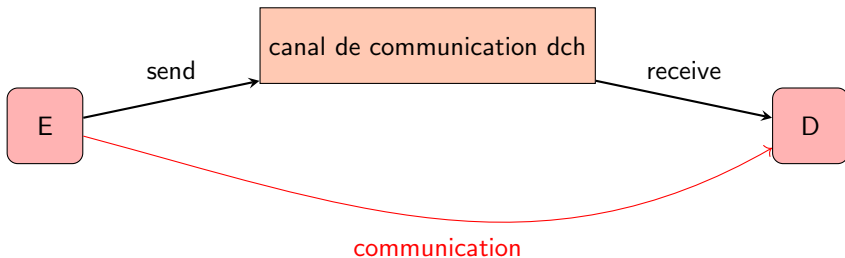
END

- Initialement, le fichier *outfile* contient n'importe quoi.
- L'événement communication réalise le service attendu en un coup !
- L'événement protocoling modélise que tout peut arriver et en particulier, le fichier transmis peut être corrompu et la communication peut être mauvaise.

Observation des communications

Le protocole de communication est construit selon des hypothèses de l'environnement : hostilité, fautes, pertes, mensonges ...

- Un émetteur envoie des trames à un récepteur
- La communication a lieu dans un seul sens
- Le canal de communication est parfait
- Le récepteur reçoit toutes les données transmises
- L'émetteur envoie les données le plus rapidement possible
- Une trame contient les informations suivantes : type (indicateur de données ou non), séquence (numéro de trame), ack (ack éventuel), info (information transportée ou octet)
- Événements :
 - ▶ `SENDING(trame,dest)` via la couche physique
 - ▶ `RECEIVING(trame)` via la couche physique



- E envoie à D par le canal de communication modélisé par la variable dch
- `sendingdata` dépose la valeur $infile(s)$ dans le canal de communication
- `receivingdata` récupère la valeur suivante se trouvant dans le canal dch .
- On conserve l'ordre d'envoi de type fifo

- *outfile* est localisée sur D et reçoit les valeurs de E via le canal.
- r et s sont deux indicis de contrôle pour gérer en type fifo
- *dch* modélise le canal de communication.

INVARIANTS

$inv1 : outfile \in 1 .. n \rightarrow DATA$

$inv2 : r \in 0 .. n$

$inv3 : s \in 1 .. n+1$

$inv4 : r \leq s$

$inv5 : dch \in 1 .. n \rightarrow DATA$

$inv6 : outfile = 1 .. r \triangleleft infile$

$inv7 : dch \subseteq 1 .. s-1 \triangleleft infile$

$inv8 : outfile \subseteq dch$

- (inv6) : *outfile* contient la copie du fichier *infile* entre 1 et r .
- (inv4) : le curseur de réception r est plus petit que le curseur d'envoi s .
- (inv8) : les données reçues sont des copies de données transmises dans *dch*
- (inv7) : le canal *dch* ne transmet que des valeurs du fichier *infile* dans l'ordre fifo avec comme borne courante s

EVENT INITIALISATION

BEGIN

$act1 : outfile := \emptyset$

$act2 : r := 0$

$act3 : s := 1$

$act4 : dch := \emptyset$

END

EVENT transmissionover

REFINES communication

WHEN

$grd1 : r = n$

THEN

$skip$

END

EVENT sendingdata

WHEN

$grd1 : s \leq n$

THEN

$act1 : dch(s) := infile(s)$

$act2 : s := s+1$

END

EVENT receivingdata

REFINES protocoling

WHEN

$grd1 : r+1 \in dom(dch)$

THEN

$act1 : outfile(r+1) := dch(r+1)$

$act2 : r := r+1$

END

- Les trames sont modélisées par l'ensemble des données *DATA*

EVENT sendingdata

WHEN

$grd1 : s \leq n$

THEN

$act1 : dch(s) := infile(s)$

$act2 : s := s+1$

END

EVENT receivingdata

REFINES protocoling

WHEN

$grd1 : r+1 \in dom(dch)$

THEN

$act1 : outfile(r+1) := dch(r+1)$

$act2 : r := r+1$

END

- Les trames sont modélisées par l'ensemble des données *DATA* : une étape ultérieure est nécessaire pour coder des structures plus complexes.

EVENT sendingdata

WHEN

$grd1 : s \leq n$

THEN

$act1 : dch(s) := infile(s)$

$act2 : s := s+1$

END

EVENT receivingdata

REFINES protocoling

WHEN

$grd1 : r+1 \in dom(dch)$

THEN

$act1 : outfile(r+1) := dch(r+1)$

$act2 : r := r+1$

END

- Les trames sont modélisées par l'ensemble des données *DATA* : une étape ultérieure est nécessaire pour coder des structures plus complexes.
- Le réseau physique est implicitement modélisé par *dch* qui modélise l'attente ou le passage dans un autre circuit de l'information.

EVENT sendingdata

WHEN

$grd1 : s \leq n$

THEN

$act1 : dch(s) := infile(s)$

$act2 : s := s+1$

END

EVENT receivingdata

REFINES protocoling

WHEN

$grd1 : r+1 \in dom(dch)$

THEN

$act1 : outfile(r+1) := dch(r+1)$

$act2 : r := r+1$

END

- Les trames sont modélisées par l'ensemble des données *DATA* : une étape ultérieure est nécessaire pour coder des structures plus complexes.
- Le réseau physique est implicitement modélisé par *dch* qui modélise l'attente ou le passage dans un autre circuit de l'information.

Protocole FIFO-FIABLE

EVENT sendingdata

WHEN

$grd1 : s \leq n$

THEN

$act1 : dch(s) := infile(s)$

$act2 : s := s+1$

END

EVENT receivingdata

REFINES protocoling

WHEN

$grd1 : r+1 \in dom(dch)$

THEN

$act1 : outfile(r+1) := dch(r+1)$

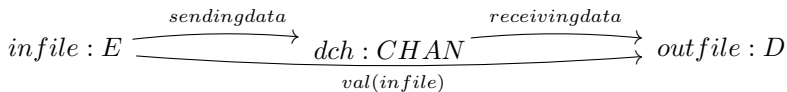
$act2 : r := r+1$

END

- $E :: \text{IF } s \leq n \text{ THEN } dch(s), s := infile(s), s+1 \text{ FI}$
- $D :: \text{IF } r+1 \in dom(dch) \text{ THEN } outfile(r+1), r := dch(r+1), r+1 \text{ FI}$
- $E :: \text{IF } s \leq n \text{ THEN } \left\{ \begin{array}{l} \text{send } infile(s) \text{ to } D \text{ in } dch \\ s := s+1 \end{array} \right. \text{ FI}$
- $D :: \text{IF } r+1 \in dom(dch) \text{ THEN } \left\{ \begin{array}{l} \text{receive } dch(r+1) \text{ in } outfile(r+1) \\ r := r+1 \end{array} \right. \text{ FI}$

Sommaire du protocole

- $E :: \text{IF } s \leq n \text{ THEN } \left\{ \begin{array}{l} \text{send } infile(s) \text{ to } D \text{ in } dch \\ s := s+1 \end{array} \right. \text{FI}$
- $D :: \text{IF } r+1 \in \text{dom}(dch) \text{ THEN } \left\{ \begin{array}{l} \text{receive } dch(r+1) \text{ in } outfile(r+1) \\ r := r+1 \end{array} \right. \text{FI}$



fiable avec réordonnement possible

VARIABLES

outfile, s, dch

INVARIANTS

$inv1 : s \in 1 .. n+1$

$inv2 : dch \in 1 .. n \rightarrow DATA$

$inv4 : dom(dch) \cap dom(outfile) = \emptyset$

$inv3 : dch \cup outfile = 1 .. (s-1) \triangleleft infile$

réordonnancement possible

EVENT INITIALISATION

BEGIN

$act1 : outfile := \emptyset$

$act2 : s := 1$

$act3 : dch := \emptyset$

END

EVENT communication2

REFINES communication

WHEN

$grd1 : s = n+1$

$grd2 : dch = \emptyset$

THEN

$skip$

END

Modélisation du protocole fiable avec attente

EVENT sendingdata

WHEN

$grd1 : s \leq n$

THEN

$act1 : dch(s) := infile(s)$

$act2 : s := s+1$

END

EVENT receivingdata

REFINES protocoling

ANY

r, m

WHERE

$grd1 : r \in dom(dch)$

$grd2 : m \in DATA$

$grd3 : dch(r) = m$

$grd4 : r \notin dom(outfile)$

THEN

$act1 : outfile := outfile \cup \{r \mapsto m\}$

$act2 : dch := dch \setminus \{r \mapsto m\}$

END

- Hypothèse de fiabilité excessive
- Le canal de communication peut ne pas protéger les données transmises
- Exercice : traduire en TLA^+ ce modèle.



Prise en compte des pertes de messages.

Protocole fiable avec attente

- Le protocole envoie des trames mais attend que le récepteur lui signale la réception
- Le canal de communication est toujours supposé fiable mais le flux est contrôlé
- Événements :
 - ▶ `SENDING(trame,dest)` via la couche physique
 - ▶ `WAITING(trameack)` via la couche physique
 - ▶ `RECEIVING(trame)` via la couche physique
 - ▶ `SENDIONG(trameack,eme)` via la couche physique

Protocole fiable avec attente

- Le protocole envoie des trames mais attend que le récepteur lui signale la réception
- Le canal de communication est toujours supposé fiable mais le flux est contrôlé
- Événements :
 - ▶ `SENDING(trame,dest)` via la couche physique
 - ▶ `WAITING(trameack)` via la couche physique
 - ▶ `RECEIVING(trame)` via la couche physique
 - ▶ `SENDIONG(trameack,eme)` via la couche physique
- Hypothèse de fiabilité irréaliste

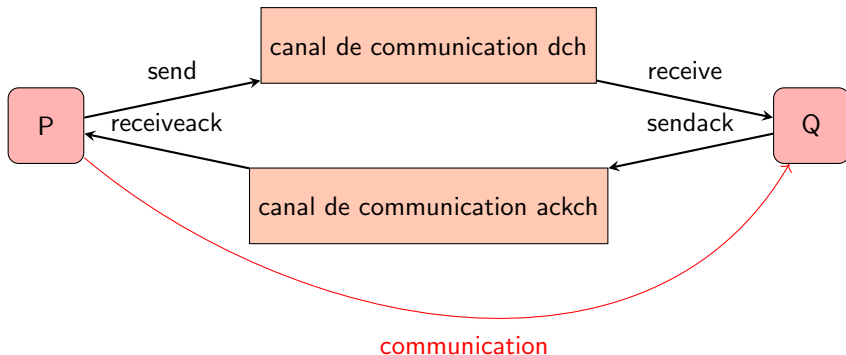
Protocole avec acquittement et retransmission

- Le protocole précédent est non fiable dans la mesure où la couche physique est peut-être non fiable ou bruitée
- L'idée est de contrôler si la trame envoyée est bien reçue sinon on renvoie mais le récepteur doit être capable de savoir qu'il s'agit d'une trame dupliquée.
- Événements :
 - ▶ `SENDING(trame,dest)` via la couche physique avec un numéro de trame
 - ▶ `WAITING(trameack)` via la couche physique du numéro de trame et ré-émission de la trame si nécessaire
 - ▶ `RECEIVING_ACK` via le canal physique de la trame ack avec le numéro
 - ▶ `RECEIVING(trame)` via la couche physique
 - ▶ `SENDING(trameack,eme)` via la couche physique avec le numéro de trame reçue

Section Courante

- ① Communications entre processus]
- ② Observation et modélisation
- ③ Modélisation de protocoles de communication
- ④ Protocole de Stenning**
- ⑤ Sliding Window Protocol

Protocole de Stenning



- Soient deux entités P et Q : P veut envoyer des messages à Q
- P envoie une suite de données d_1, \dots, d_n à Q
- P répète les deux opérations :
 - ▶ phase d'envoi : P envoie la donnée (d_i, i) à Q
 - ▶ phase d'attente : P attend de recevoir la valeur i
 - ▶ phase de confirmation : si i est reçue, alors le protocole reprend à la phase d'envoi

Protocole de Stenning

VARIABLES : $dch, s, r, ackch, outfile$.

$inv1 : r \in 0 .. n$
 $inv2 : s \in 1 .. n+1$
 $inv3 : dch \in 1 .. n \rightarrow DATA$
 $inv4 : ackch \subseteq 1 .. n$
 $inv5 : r \leq s$
 $inv6 : s \leq r+1$
 $inv7 : outfile = 1 .. r \triangleleft infile$
 $inv8 : dch \subseteq 1 .. s \triangleleft infile$
 $inv9 : s \leq n+1$
 $inv10 : r \leq n$
 $inv11 : outfile \in 1 .. n \rightarrow DATA$
 $inv12 : s \notin dom(outfile) \Rightarrow s = r+1$
 $inv13 : s \in dom(outfile) \Rightarrow s = r$
 $inv14 : s \in ackch \Rightarrow s \in dom(outfile)$
 $inv15 : ackch \subseteq 1 .. r$

THEOREMS

$th : r = n \Rightarrow outfile = infile$ *safety property*

Protocole de Stenning

EVENT INITIALISATION

BEGIN

$act1 : dch := \emptyset$

$act2 : ackch := \emptyset$

$act3 : s := 1$

$act4 : r := 0$

$act5 : outfile := \emptyset$

END

EVENT servicedone

REFINES communication

WHEN

$grd1 : r = n$

THEN

$skip$

END

EVENT sendingdata

WHEN

$grd1 : s \leq n$

THEN

$act1 : dch(s) := infile(s)$

END

Protocole de Stenning

```
EVENT receivingadata  
  REFINES protocoling  
  WHEN  
     $grd1 : r+1 \in dom(dch)$   
  THEN  
     $act1 : outfile(r+1) := dch(r+1)$   
  
     $act2 : r := r+1$   
  
  END
```

```
EVENT sendingack  
  WHEN  
     $grd1 : r \neq 0$   
  THEN  
     $act1 : ackch := ackch \cup \{r\}$   
  END  
EVENT receivingack  
  WHEN  
     $grd1 : s \in ackch$   
  THEN  
     $act1 : s := s+1$   
  END
```

Protocole de Stenning

EVENT daemondch

ANY

i, m

WHERE

$grd1 : i \in 1 .. n$

$grd2 : m \in DATA$

$grd3 : i \mapsto m \in dch$

THEN

$act1 : dch := dch \setminus \{i \mapsto m\}$

END

EVENT daemonack

ANY

i

WHERE

$grd1 : i \in ackch$

THEN

$act1 : ackch := ackch \setminus \{i\}$

END

- Les deux événements introduisent des erreurs du type perte de messages
- Le modèle permet donc de décrire à la fois le protocole et l'environnement.

Protocole bidirectionnel à fenêtre de taille 1

- Ce protocole est aussi appelé le protocole du *bit alterné*
- Pour régler les problèmes de codage des numéros de trame, on observe que l'on a besoin uniquement d'un bit pour contrôler si la trame est effectivement répétée et reçue in fine.

Protocole bidirectionnel à fenêtre de taille n

- Ce protocole est aussi appelé le protocole du *sliding window protocol*
- Le protocole précédent peut être amélioré par une fenêtre de longueur $n \geq 1$ et cette fenêtre glisse en suivant les trames reçues.

- 1

SERVICE Sliding-Window Protocol

INVARIANTS

$inv2 : OUT \in \mathbb{N} \leftrightarrow D$

EVENTS

EVENT INITIALISATION

BEGIN

$act2 : OUT := \emptyset$

END

EVENT communication

BEGIN

$act1 : OUT := IN$

END

EVENT protocoling

BEGIN

$act1 : OUT : |(OUT' \subseteq IN)$

END

END

CONTEXT *DATA*
SETS

D

CONSTANTS

n, IN, l

AXIOMS

axm1 : $n \in \mathbb{N}_1$

axm2 : $IN \in \mathbb{N} \rightarrow D$

axm3 : $\text{dom}(IN) = 0 \dots n$

axm4 : $l \in \mathbb{N}_1$

axm5 : $l \leq n$

END

PROCESSUS Sliding-Window Protocol

VARIABLES : OUT , i , $chan$, ack , got

$inv1 : OUT \in \mathbb{N} \leftrightarrow D$

$inv2 : i \in 0 .. n+1$

$inv3 : 0 .. i-1 \subseteq dom(OUT)$

$inv3bis : dom(OUT) \subseteq 0 .. n$

$inv7 : chan \in \mathbb{N} \leftrightarrow D$

$inv8 : ack \subseteq \mathbb{N}$

$inv9 : ack \cup got \subseteq i .. i+l \cap 0 .. n$

$inv10 : got \subseteq \mathbb{N}$

$inv12 : dom(chan) \subseteq 0 .. i+l \cap 0 .. n$

$inv13 : got \subseteq dom(OUT)$

$inv14 : ack \subseteq dom(OUT)$

$inv16 : 0 .. i-1 \triangleleft OUT = 0 .. i-1 \triangleleft IN$

$inv17 : chan \subseteq IN$

$inv18 : OUT \subseteq IN \wedge got \subseteq dom(chan)$

PROCESSUS Sliding-Window Protocol

VARIABLES : OUT , i , $chan$, ack , got

$inv1 : OUT \in \mathbb{N} \leftrightarrow D$

$inv2 : i \in 0 .. n+1$

$inv3 : 0 .. i-1 \subseteq dom(OUT)$

$inv3bis : dom(OUT) \subseteq 0 .. n$

$inv7 : chan \in \mathbb{N} \leftrightarrow D$

$inv8 : ack \subseteq \mathbb{N}$

$inv9 : ack \cup got \subseteq i .. i+l \cap 0 .. n$

$inv10 : got \subseteq \mathbb{N}$

$inv12 : dom(chan) \subseteq 0 .. i+l \cap 0 .. n$

$inv13 : got \subseteq dom(OUT)$

$inv14 : ack \subseteq dom(OUT)$

$inv16 : 0 .. i-1 \triangleleft OUT = 0 .. i-1 \triangleleft IN$

$inv17 : chan \subseteq IN$

$inv18 : OUT \subseteq IN \wedge got \subseteq dom(chan)$

- Idée du protocole : faire glisser une fenêtre

PROCESSUS Sliding-Window Protocol

VARIABLES : OUT , i , $chan$, ack , got

$inv1 : OUT \in \mathbb{N} \rightarrow D$

$inv2 : i \in 0 .. n+1$

$inv3 : 0 .. i-1 \subseteq dom(OUT)$

$inv3bis : dom(OUT) \subseteq 0 .. n$

$inv7 : chan \in \mathbb{N} \rightarrow D$

$inv8 : ack \subseteq \mathbb{N}$

$inv9 : ack \cup got \subseteq i .. i+l \cap 0 .. n$

$inv10 : got \subseteq \mathbb{N}$

$inv12 : dom(chan) \subseteq 0 .. i+l \cap 0 .. n$

$inv13 : got \subseteq dom(OUT)$

$inv14 : ack \subseteq dom(OUT)$

$inv16 : 0 .. i-1 \triangleleft OUT = 0 .. i-1 \triangleleft IN$

$inv17 : chan \subseteq IN$

$inv18 : OUT \subseteq IN \wedge got \subseteq dom(chan)$

- Idée du protocole : faire glisser une fenêtre
- Réception progressive des données au cours du glissement

PROCESSUS Sliding-Window Protocol

VARIABLES : OUT , i , $chan$, ack , got

$inv1 : OUT \in \mathbb{N} \rightarrow D$

$inv2 : i \in 0 .. n+1$

$inv3 : 0 .. i-1 \subseteq dom(OUT)$

$inv3bis : dom(OUT) \subseteq 0 .. n$

$inv7 : chan \in \mathbb{N} \rightarrow D$

$inv8 : ack \subseteq \mathbb{N}$

$inv9 : ack \cup got \subseteq i .. i+l \cap 0 .. n$

$inv10 : got \subseteq \mathbb{N}$

$inv12 : dom(chan) \subseteq 0 .. i+l \cap 0 .. n$

$inv13 : got \subseteq dom(OUT)$

$inv14 : ack \subseteq dom(OUT)$

$inv16 : 0 .. i-1 \triangleleft OUT = 0 .. i-1 \triangleleft IN$

$inv17 : chan \subseteq IN$

$inv18 : OUT \subseteq IN \wedge got \subseteq dom(chan)$

- Idée du protocole : faire glisser une fenêtre
- Réception progressive des données au cours du glissement
- Deux canaux d'échanges :

PROCESSUS Sliding-Window Protocol

VARIABLES : OUT , i , $chan$, ack , got

$inv1 : OUT \in \mathbb{N} \rightarrow D$
 $inv2 : i \in 0 .. n+1$
 $inv3 : 0 .. i-1 \subseteq dom(OUT)$
 $inv3bis : dom(OUT) \subseteq 0 .. n$
 $inv7 : chan \in \mathbb{N} \rightarrow D$
 $inv8 : ack \subseteq \mathbb{N}$
 $inv9 : ack \cup got \subseteq i .. i+l \cap 0 .. n$
 $inv10 : got \subseteq \mathbb{N}$
 $inv12 : dom(chan) \subseteq 0 .. i+l \cap 0 .. n$
 $inv13 : got \subseteq dom(OUT)$
 $inv14 : ack \subseteq dom(OUT)$
 $inv16 : 0 .. i-1 \triangleleft OUT = 0 .. i-1 \triangleleft IN$
 $inv17 : chan \subseteq IN$
 $inv18 : OUT \subseteq IN \wedge got \subseteq dom(chan)$

- Idée du protocole : faire glisser une fenêtre
- Réception progressive des données au cours du glissement
- Deux canaux d'échanges :
 - ▶ $chan$ est le canal de communication des données à partir de IN vers OUT

PROCESSUS Sliding-Window Protocol

VARIABLES : OUT , i , $chan$, ack , got

$inv1 : OUT \in \mathbb{N} \rightarrow D$
 $inv2 : i \in 0 .. n+1$
 $inv3 : 0 .. i-1 \subseteq dom(OUT)$
 $inv3bis : dom(OUT) \subseteq 0 .. n$
 $inv7 : chan \in \mathbb{N} \rightarrow D$
 $inv8 : ack \subseteq \mathbb{N}$
 $inv9 : ack \cup got \subseteq i .. i+l \cap 0 .. n$
 $inv10 : got \subseteq \mathbb{N}$
 $inv12 : dom(chan) \subseteq 0 .. i+l \cap 0 .. n$
 $inv13 : got \subseteq dom(OUT)$
 $inv14 : ack \subseteq dom(OUT)$
 $inv16 : 0 .. i-1 \triangleleft OUT = 0 .. i-1 \triangleleft IN$
 $inv17 : chan \subseteq IN$
 $inv18 : OUT \subseteq IN \wedge got \subseteq dom(chan)$

- Idée du protocole : faire glisser une fenêtre
- Réception progressive des données au cours du glissement
- Deux canaux d'échanges :
 - ▶ $chan$ est le canal de communication des données à partir de IN vers OUT
 - ▶ ack est le canal de retour.

PROCESSUS Sliding-Window Protocol

VARIABLES : OUT , i , $chan$, ack , got

$inv1 : OUT \in \mathbb{N} \rightarrow D$
 $inv2 : i \in 0 .. n+1$
 $inv3 : 0 .. i-1 \subseteq dom(OUT)$
 $inv3bis : dom(OUT) \subseteq 0 .. n$
 $inv7 : chan \in \mathbb{N} \rightarrow D$
 $inv8 : ack \subseteq \mathbb{N}$
 $inv9 : ack \cup got \subseteq i .. i+l \cap 0 .. n$
 $inv10 : got \subseteq \mathbb{N}$
 $inv12 : dom(chan) \subseteq 0 .. i+l \cap 0 .. n$
 $inv13 : got \subseteq dom(OUT)$
 $inv14 : ack \subseteq dom(OUT)$
 $inv16 : 0 .. i-1 \triangleleft OUT = 0 .. i-1 \triangleleft IN$
 $inv17 : chan \subseteq IN$
 $inv18 : OUT \subseteq IN \wedge got \subseteq dom(chan)$

- Idée du protocole : faire glisser une fenêtre
- Réception progressive des données au cours du glissement
- Deux canaux d'échanges :
 - ▶ $chan$ est le canal de communication des données à partir de IN vers OUT
 - ▶ ack est le canal de retour.

Initialisation du protocole

EVENT INITIALISATION

BEGIN

act1 : *OUT* := \emptyset

act2 : *i* := 0

act4 : *chan* := \emptyset

act5 : *ack* := \emptyset

act6 : *got* := \emptyset

END

Phase d'envoi et de réception

EVENT send

ANY

j

WHERE

$grd1 : j \in i .. i+l$

$grd2 : j \leq n$

$grd3 : j \notin got$

THEN

$act1 : chan(j) := IN(j)$

END

EVENT receive

REFINES protocoling

ANY

j

WHERE

$grd1 : j \in dom(chan)$

$grd2 : j \in i .. i+l$

THEN

$act1 : OUT(j) := chan(j)$

$act2 : ack := ack \cup \{j\}$

END

Phase d'accusé de réception et de complétion

EVENT receiveack

ANY

k

WHERE

$grd1 : k \in ack$

THEN

$act1 : got := got \cup \{k\}$

$act2 : ack := ack \setminus \{k\}$

END

EVENT completion

REFINES communication

WHEN

$grd1 : i = n+1 \wedge got = \emptyset$

THEN

$skip$

END

Gestion de la fenêtre

La fenêtre glisse quand elle peut le faire ($i+l < n$).

EVENT sliding

WHEN

$grd1 : got \neq \emptyset$

$grd3 : i \in got$

$grd4 : i+l < n$

THEN

$act1 : i := i+1$

$act2 : got := got \setminus \{i\}$

$act3 : ack := ack \setminus \{i\}$

END

La fenêtre ne glisse plus quand elle ne peut plus mais elle se vide et fond en quelque sorte ($i+l \geq n$).

EVENT emptywindow

WHEN

$grd1 : got \neq \emptyset$

$grd2 : i \in got$

$grd3 : i+l \geq n$

$grd4 : i \leq n$

THEN

$act1 : i := i+1$

$act2 : got := got \setminus \{i\}$

$act3 : ack := ack \setminus \{i\}$

END

deux canaux

EVENT loosingchan

ANY

j

WHERE

$grd1 : j \in i .. i+l$

$grd2 : j \in dom(chan)$

$grd3 : j \notin got$

THEN

$act1 : chan := \{j\} \triangleleft chan$

END

EVENT loosingack

ANY

k

WHERE

$grd1 : k \in ack$

THEN

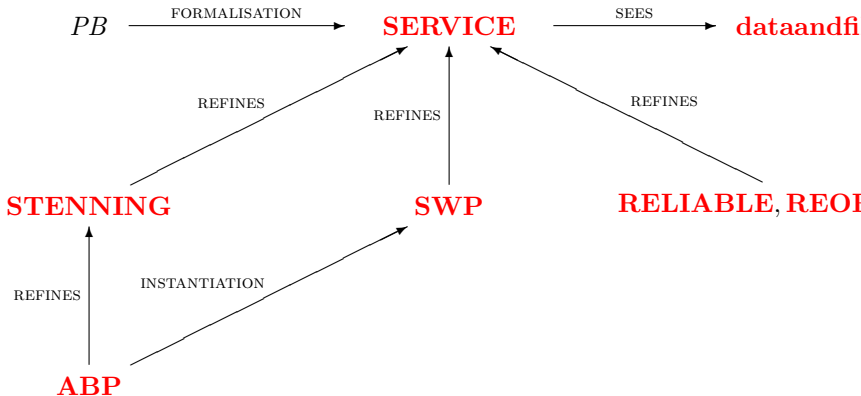
$act1 : ack := ack \setminus \{k\}$

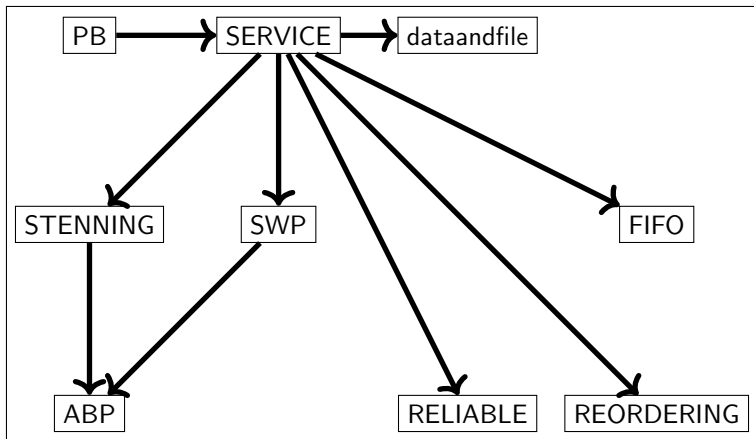
END

Le protocole du bit alterné en TLA⁺

- Le protocole du bit alterné est une instance de ce protocole pour $l = 0$.
- Le choix de l est important puisqu'il intervient dans le codage du témoin de transmission.
- On peut imaginer que les choix suivants sont pertinents :
 - ▶ $l = 0$: codage sur 0 bits
 - ▶ $l = 2$: codage sur 1 bits
 - ▶ $l = 4$: codage sur 2 bits
 - ▶ $l = 2^k$: codage sur k bits
- Dans le cas d'un codage sur k bits, on choisit ce qui reste sur la trame d'envoi comme place.

Status of development





Conclusion

- Modèles en couches
- Relation d'abstraction
- Mécanismes de répétition et de contrôle du
- Exemple de TCP/IP