Lot 4.2

Technologie de modélisation

Probabilités

Verification du protocole CSMA/CD sous PRISM

Description :	Le protocole CSMA/CD, plus connu sur le nom de protocole Ethernet, a déjà été vérifié plusieurs fois de manière qualitatitave avec des model-checkers temporisés. Nous avons ici, pour la première fois trouvé des propriétés quantitatives (des probabilités) grace au model checker probabiliste PRISM.
Auteur(s) :	Marie Duflot, Stephane Messika, Claudine Picaronny
Référence :	Averroes / Lot 4.2 / Fourniture 1 / V1.0
Date :	14 janvier 2004
Statut :	Version préliminaire
Version :	1.0

Réseau National des Technologies Logicielles
Projet subventionné par le Ministère de la Recherche et des Nouvelles Technologies

CRIL Technology, France Télécom R&D, INRIA-Futurs, LaBRI (Univ. de Bordeaux – CNRS), LIX (École Polytechnique, CNRS) LORIA, LRI (Univ. de Paris Sud – CNRS), LSV (ENS de Cachan – CNRS)

Historique

20 décembre 2003	V 0.1	version préliminaire
14 Janvier 2004	V 1.0	mise au format averroes

Table des matières

1 Introduction

Les protocoles de communication entre ordinateurs (ou autres composants) sont de plus en plus nombreux. Du fait que les canaux de communication sont en nombre limités, les protocoles sus-cités doivent autant que possible limiter les collisions entre messages, ou "conflits", voire les éviter. Pour ce faire, et comme on va le voir dans le processus étudié ici, on peut avoir recours aux probabilités.

Un des protocoles de communication les plus importants est connu sous le nom de protocole Ethernet. Nous allons ici vérifier, pour la première fois les propriétés quantitatives du protocole Ethernet. En effet, il nous semble plus intéressant de savoir avec quelle probabilité des événements vont se produire (envoi correct d'un message, collisions éventuelles), plutôt que d'uniquement savoir si ces événements peuvent se produire ou non. Les études précédentes concernant ce protocole ayant été réalisées à l'aide de model-checkers non probabilistes, aucune donnée quantitative n'était connue. C'est pourquoi nous avons fait le choix d'utiliser le model-checker probabiliste PRISM pour les obtenir.

2 Description basique du protocole

Le protocole CSMA/CD : Carrier Sense Multiple Access with Collision Detection est un protocole qui régit la communication entre plusieurs entités appelées émetteurs, qui communiquent au moyen d'un canal. Le standard IEEE 802.3 [IEE02] décrit avec précision les différents aspects du protocole ainsi que les différentes entités nécessaires pour le réaliser. Dans ce rapport, nous allons nous intéresser plus particulièrement à une version du protocole appelée *half duplex*.

Dans la version *half duplex*, CSMA/CD décrit la façon dont plusieurs ordinateurs partagent un canal de communication, au travers duquel un seul message peut passer à la fois (par opposition au *full duplex* pour lequel on peut faire circuler dans le canal simultanément un message dans chaque sens).

Nous allons à présent décrire le protocole CSMA/CD (en mode *half duplex*), ou plus précisément les aspects qui vont servir à notre modélisation. Ainsi, de nombreux détails, comme la façon dont un message envoyé est structuré, seront omis lors de la description. Toutes précisions supplémentaires peuvent être trouvées dans le standard [IEE02].

Émission

Pour transmettre un message, un émetteur surveille le canal. Lorsqu'il le voit libre, il commence à émettre son message bit à bit. Comme la transmission des messages se fait à la vitesse de la propagation électrique, elle n'est pas considérée comme instantanée, et cela peut justement engendrer des collisions, comme va le voir dans le paragraphe suivant.

Conflit

Supposons que, dans le réseau, deux émetteurs, `sender1` et `sender2` souhaitent chacun envoyer un message à l'autre. Appelons σ le temps de propagation du signal entre ces deux émetteurs. Initialement, le canal est libre. Ainsi, `sender1` voit le canal libre et commence à envoyer son message. Si le `sender2` regarde le canal moins de σ unités de temps plus tard, il voit toujours le canal libre et peut commencer lui aussi à envoyer son message. Ainsi les deux messages vont entrer en collision et être perdus.

Le protocole vise à éviter que ce genre de conflit ne se reproduise trop souvent. Ainsi, lorsque les deux messages entrent en collision, les deux signaux se superposent et ce signal brouillé se propage à son tour dans le réseau. Quand les émetteurs reçoivent ce signal brouillé, ils continuent à émettre un message d'erreur (jusqu'à ce que celui-ci se soit propagé sur tout le réseau). Ensuite les émetteurs attendent que le canal se vide puis tirent au sort leur temps d'attente, au bout duquel ils vont réessayer d'envoyer leur message.

Afin d'éviter que les collisions ne se répètent trop souvent, le temps d'attente est choisi de manière uniforme entre 0 et 2^k où $k = \min(n, 10)$, et où n compte le nombre de collisions depuis la dernière transmission de message réussie. Ainsi, au fur et à mesure que le nombre de collisions successives augmente, la borne supérieure du temps d'attente va elle aussi augmenter, ce qui va diminuer les chances de nouvelles collisions.

Conséquences sur l'émission

Quand l'émetteur commence à envoyer son message, il ne peut être sûr que ce message va atteindre son destinataire sans être altéré. Ainsi, durant le début de l'émission, l'émetteur est dans un état d'incertitude. Par contre, si après un certain temps (le temps nécessaire pour qu'un message parcoure le réseau et revienne jusqu'à l'émetteur considéré) l'émetteur n'a pas reçu de signal d'erreur ou de signal brouillé, il peut être sûr que son message va passer correctement. Il sait en effet que les autres émetteurs n'ont pas tenté d'envoyer de message avant de voir le canal occupé.

3 Modélisation sous forme d'automates temporisés probabilistes

Définition

Un **automate temporisé probabiliste** est un n-uplet $G = (\mathcal{S}, \mathcal{L}, s_i, \mathcal{X}, inv, prob, (\tau_s)_{s\in\mathcal{S}})$ où :

- $\mathcal{S}$ est un ensemble fini de nœuds.
- $\mathcal{L} : \mathcal{S} \to 2^{AP}$ est une fonction assignant à chaque nœud de l'automate un ensemble de propositions atomiques qui sont vraies dans ce nœud.
- s_i est le nœud initial.
- $\mathcal{X}$ est un ensemble fini d'horloges. On notera $Z_\mathcal{X}$ l'ensemble des "zones" d'horloges définies par des inéquations du genre $x \sim c$ et $x - y \sim c$, avec $x, y \in \mathcal{X}$, c une constante et $\sim \in \{<, >, \leq, \geq, =\}$.
- $inv : \mathcal{S} \to Z_\mathcal{X}$ est une fonction assignant à chaque nœud une condition invariante.
- $prob : \mathcal{S} \to \mathcal{P}(\mu(\mathcal{S} \times 2^\mathcal{X}))$ est une fonction assignant à chaque nœud un ensemble fini de distributions de probabilités sur $\mathcal{S} \times 2^\mathcal{X}$.
- $(\tau_s)_{s\in\mathcal{S}}$ est une famille de fonctions telle que, pour tout $s \in \mathcal{S}$, $\tau_s : prob(s) \to Z_\mathcal{X}$ assigne à chaque p de $prob(s)$ une condition d'activation.

Ceci est la définition formelle. Le comportement de l'automate se traduit en des transitions entre états, qui peuvent être soit le résultat du passage du temps, soit dûes à des transitions discrètes. Le rôle de la condition invariante est d'établir l'ensemble des états admissibles, et donc d'interdire les transitions vers les états inadmissibles.

Le protocole CSMA/CD peut être représenté par un automate temporisé probabiliste, mais nous n'allons pas le décrire ici pour deux raisons :

- La première est que la taille de l'automate est trop importante (les distributions probabilistes changent avec le nombre de collisions).
- La seconde est que les techniques de Model checking sur les automates temporisés probabilistes ne sont pas encore assez développées. Des algorithmes rigoureux existent mais leur complexité est trop importante et ils ne sont implémentés dans aucun model-checker. On utilise en général d'abord un model-checker temporisé (type UPPAAL ou KRONOS) pour construire le graphe des zones, puis un model checker probabiliste (type PRISM) pour vérifier les propriétés probabilistes (voir par exemple [KNS02]).

Nous avons donc fait un choix de simplification du protocole en discrétisant le temps. Ce choix est celui qui nous permet directement de passer d'un modèle compliqué (Automate Temporisé Probabiliste) en un modèle plus simple (processus de décision markovien) pris entièrement en

compte par l'outil PRISM. Ce choix a déjà été utilisé dans les exemples de protocoles probabilistes déjà vérifiés (CSMA/CA [KNS02], FireWire [DKN02, KNS03],...).

4 Le protocole CSMA/CD vu comme un processus de décision markovien

Définition

Un **processus de décision markovien** est un quadruplet $G = (\mathcal{S}, s_i, \mathcal{L}, \tau)$ où :
- $\mathcal{S}$ est un ensemble fini ou dénombrable d'états.
- s_i est l'état initial.
- $\mathcal{L} : \mathcal{S} \to 2^{AP}$ est une fonction associant à chaque état l'ensemble des propositions atomiques vraies dans cet état.
- τ est une fonction qui associe à chaque état un ensemble de distribution de probabilités sur $\mathcal{S}$.

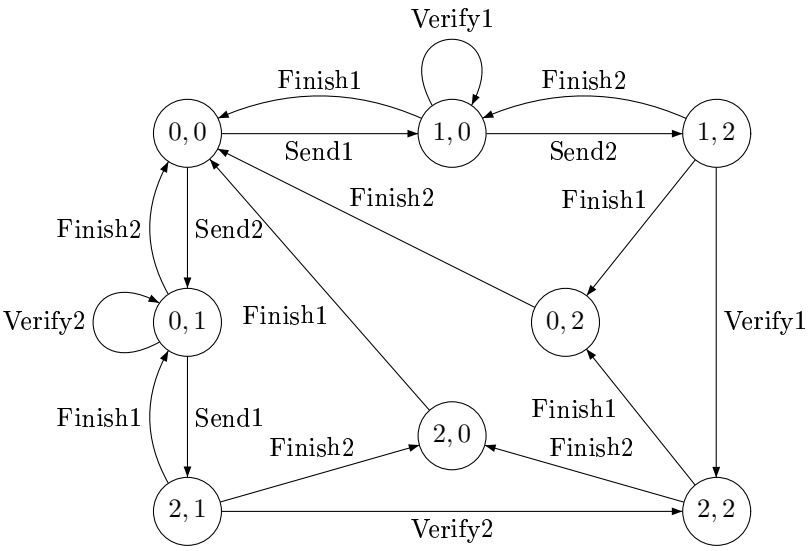
Ceci est aussi une définition formelle, mais il s'agit uniquement d'un processus alliant à la fois un choix non déterministe (entre les distributions disponibles) et un choix probabiliste.
Dans notre exemple, le non déterministe se trouve en deux endroits ; dans le choix ou non d'avoir envie d'envoyer un message et dans le choix entre les deux émetteurs.

Voici ici une description des différentes parties du protocole, sous forme d'automates en vue de les implémenter en PRISM.

Description rapide des Automates

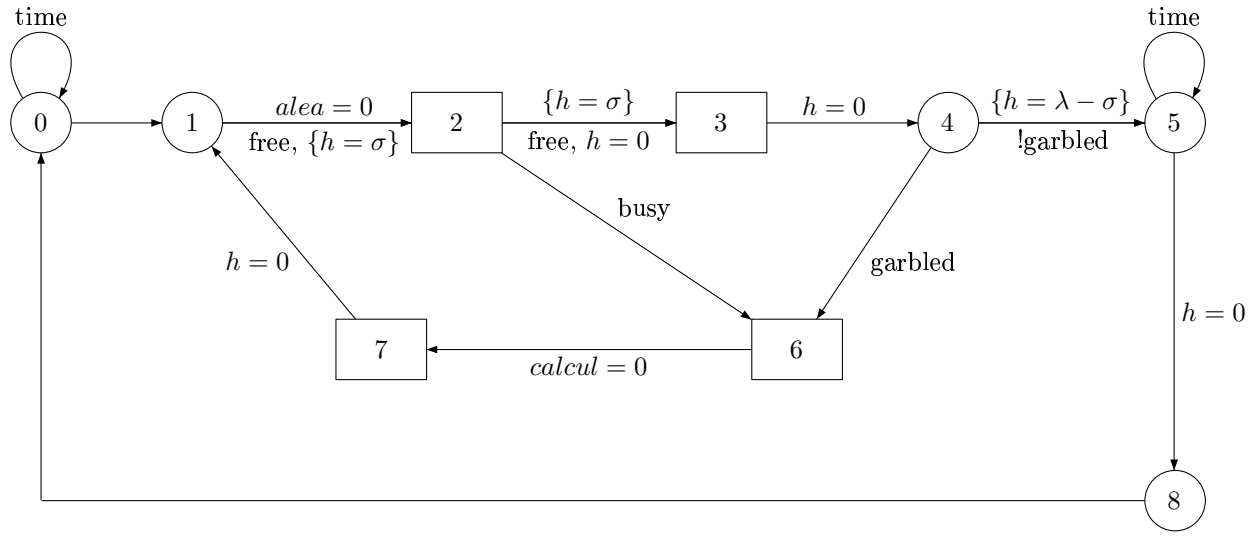
Automate du canal

Ce module représente le canal, il y a deux variables (`c1 et c2`) qui représentent l'état du canal vu par l'un ou l'autre des émetteurs. De ces variables on déduit si le canal est libre (`free`) occupé (`busy`) ou brouillé (`garbled`).



Automate du temps
Un module qui sert à comptabiliser le temps écoulé depuis le début.

Automates de `sender1` et `sender2`
Ce module décrit le comportement de chacun des émetteurs. Pour l'emetteur `sende1`, il contient une horloge (`h1`), une variable stockant le temps aléatoire `alea1`), la variable des états de l'émetteur (`s1`) et plusieurs variables tampons.
Voici l'automate correspondant à l'implémentation PRISM de `sender1`. Le module `sender2` est identique, modulo le renommage des variables.

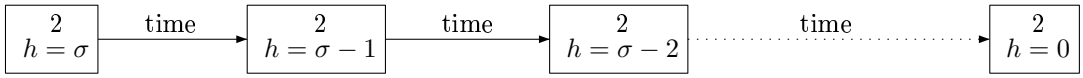


0 : Initialisation
1 : Envie d'envoyer un message
2 : Envoi du message période d'attente 1
3 : Envoi du message période d'attente 2
4 : Test au bout de 2σ
5 : Envoi du message, sur qu'il passera
6 : Collision détectée
7 : Calcul du temps aléatoire
8 : Message envoyé correctement

Maintenant nous pouvons détailler les sous-automates correspondants aux états $2, 3, 6$ et 7 :

États 2 et 3

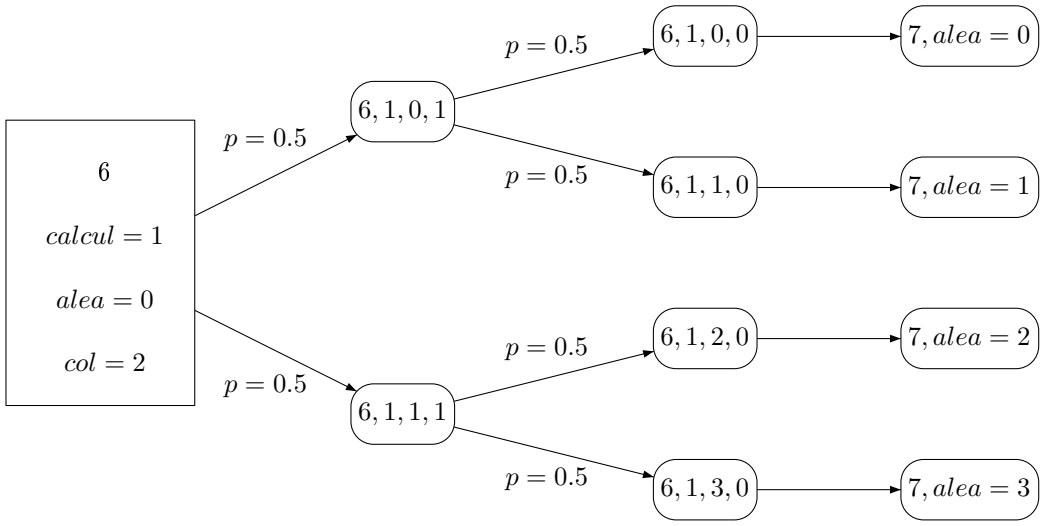
Les états 2 et 3 sont très similaires et correspondent à une attente de σ unités de temps, afin de détecter si le message va passer ou non.Après σ unités de temps sans voir le canal occupé, un émetteur sait qu'aucun autre émetteur n'a envoyé de message juste avant lui (qui risque donc d'entrer en collision avec le sien). Après 2σ unités de temps, l'émetteur sait également si soit son message est perdu, soit aucun autre émetteur ne va brouiller le message.



Etat 6 avec 2 collisions

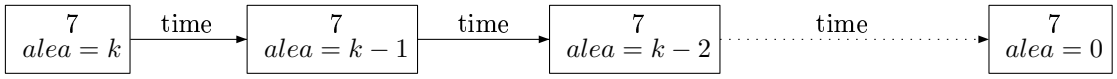
L'état 6 est ateint ors qu'un envoi de message a échoué. Il correspond au tirage aléatoire du temps d'attente. Ici il y a eu deux collisions, et donc le délai d'attente peut varier entre 0 et

3 unités de temps. Le temps d'attente tiré aléatoirement est calculé en binaire (donc ici pour 2 collisions on tire 2 valeurs booléennes aléatoirement).



État 7

L'état 7 est atteint lorsqu'un envoi de message a échoué. Dans l'état 7, on attend pendant un certain délai appelé `alea`, tiré aléatoirement dans l'état 6, avant de retourner dans l'état 1 et de réessayer d'envoyer un message.



5 Modélisation avec PRISM

Description de protocoles sous PRISM

Pour décrire un protocole en PRISM, il faut décrire les différentes entités sous forme de modules. Chacun des automates présentés dans le paragraphe précédent deviendra un module. Chaque module se compose d'une déclaration des variables, de la forme

```
c : [0..2] init 0 ;
```

qui signifie que `c` est une variable dont les valeurs possibles sont 0, 1 et 2, et que sa valeur initiale est 0. Un module se compose ensuite d'une liste de transitions de la forme

```
[etiquette] condition -> action ;
```

L'`étiquette` n'est pas obligatoire mais, quand elle existe, elle va permettre de synchroniser des transitions de différents modules. La `condition` est en général une formule portant sur les variables du module. Lorsque cette formule est évaluée à vrai, la transition peut être effectuée (modulo le fait que les éventuelles synchronisations sont possibles) et, en appliquant la transition, on met à jour certaines variables suivant la formule `action`, qui peut être probabiliste. Par exemple, la transition

```
[] (c=0) & (d<4) -> 0.5 : (c'=1) & (d'=2*d) + 0.5 : (c'=2);
```

n'a pas d'étiquette de synchronisation. Elle peut être effectuée lorsque la variable `c` vaut zéro et la variable `d` a une valeur strictement inférieure à 4. On peut remarquer que dans la partie

`action`, il y a un choix probabiliste (équiprobable) entre deux ensembles de mises à jour : `(c'=1) & (d'=2*d)` d'une part, et `(c'=2)` d'autre part. Les nouvelles valeurs des variables sont désignées par un nom primé. Par exemple dans `d'=2*d`, la lettre `d` désigne l'ancienne valeur de la variable et `d'` la nouvelle valeur.

Un protocole en PRISM peut également contenir des déclarations de constantes et des formules, comme nous le verrons dans le paragraphe suivant.

Modélisation de CSMA/CD

Comme on a pu le voir dans la section 2, ce protocole met en œuvre à la fois un aspect probabiliste et un aspect temporisé. Pour traiter directement le protocole avec PRISM, nous avons choisi de discrétiser le temps. Ainsi, le temps va s'écouler par petits "pas" de temps.
Malheureusement, cette méthode fait trop rapidement grandir le nombre d'états du système, en effet, dans le module du temps, un état est créé par valeur d'horloge. Nous avons donc du faire les bons choix d'implémentation pour réduire les temps de calcul. Ainsi, la variable qui sert à calculer le temps aléatoire d'attente fera par la même occasion office d'horloge (nous l'avons notée `alea`).
De plus, il a fallu faire des choix aussi pour les constantes de propagation et de longueur des messages (24 et 780 microsecondes respectivement), mais nous avons respecté l'orde de grandeur de leur rapport. Ces valeurs n'étant pas fixée à la base (le temps de propagation par exemple varie en fonction de l'éloignement entre deux émetteurs), le fait de ne pas préserver exactement leur rapport n'es pas un problème.
Enfin, nous avons calculé de manière astucieuse le temps aléatoire d'attente en base 2 (ce qui réduit le temps de calcul).

En annexe, on peut voir le code commenté implémenté sous PRISM version 1.3.

Description et choix des variables

module channel

- `c1` représente l'état du message de `sender1` (0 : rien est envoyé, 1 : le message est envoyé,2 : il est perdu)
- `c2` identique pour `sender2`.

module temps

Une seule variable t, variant de 0 à une valeur limite `tempsmax`, et représentant le temps écoulé.

module Sender

- h horloge permettant de faire attendre le sender
- *alea* variable stockant le résultat du tirage du temps aléatoire et servant aussi d'horloge.
- *col* stocke le nombre de collisions
- s stocke l'état courant du sender
- *nbc* et *calcul* utilisées pour faire le calcul du temps aléatoire.

6 Vérification de propriétés du protocole - Résultats

Nous avons deux séries de résultats pour des valeurs différentes de σ et λ qui sont $(1, 32)$ et $(2, 64)$, les temps de calcul pour les valeur plus grandes sont trop grands. Nous espérons trouver des résultats approximer pour des valeurs plus réalistes avec APMC.

La propriété que nous avons vérifiée s'écrit en PRISM sous la forme :

$P_{>=p}[trueU(((s1 = 8)|(s2 = 8)))]$

Elle exprime le fait qu'en moins de *tempsmax*, la deadline unités de temps un des sender a envoyé son message avec la probabilité p.

Cas $\lambda = 32$ et $\sigma = 1$

Cas $tempsmax = 1000$

On obtient une fois la compilation du modèle effectuée (10 minutes) plus de 26000000 d'états. Puis en 160 secondes on obtient que la propriété est vraie pour $p = 0.3$ et fausse pour $p = 0.4$.

Cas $tempsmax = 2000$

Ici, la plus grande valeur de p qui rend la formule valide est :

Cas $\lambda = 64$ et $\sigma = 2$

Cas $tempsmax = 1000$

On obtient une fois la compilation du modèle effectuée (20 minutes) plus de 27000000 d'états. Puis en 160 secondes on obtient que la propriété est vraie pour $p = 0.2$ et fausse pour $p = 0.3$.

7 Interprétation des résultats - Conclusion

Partie à remplir ! !

Références

[DKN02] Conrado Daws, Marta Z. Kwiatkowska, and Gethin Norman. Automatic verification of the root contention protocol with KRONOS and PRISM. In *Proc. 7th Int. Work. on Formal Methods for Industrial Critical Systems (FMICS'02)*, volume 66 :2 of *ENTCS*, 2002.

[IEE02] IEEE Computer Society. *IEEE Standard 802.3 : Carrier Sense Multiple Access with Collision Detection*, 2002. Disponible à l'adresse `http ://standards.ieee.org/getieee802/802.3.html`.

[KNS02] Marta Z. Kwiatkowska, Gethin Norman, and Jeremy Sproston. Probabilistic model checking of the IEEE 802.11 wireless local area network protocol. In *Proc.2nd Int. Work. on Process Algebra and Probabilistic Methods, Performance Modelling and Verification (PAPM-PROBMIV'02)*, volume 2399 of *LNCS*, pages 169–187. Springer, 2002.

[KNS03] Marta Z. Kwiatkowska, Gethin Norman, and Jeremy Sproston. Probabilistic model checking of deadline properties in the IEEE 1394 FireWire root contention protocol. *Special issue of Formal Aspects of Computing*, 14 :295–318, 2003.

8 Annexe : Code PRISM du modèle

```
//Protocole Csma-cd
//constantes de temps
const lambda=32;  //longueur d'un message
const sigma=1;  //temps de propagation d'un message
const tempsmax=2000;
const timeMax=63;
//nondeterministic

module Channel

//etats

c1 : [0..2] init 0;
c2 : [0..2] init 0;

//0 rien n'est envoyé
//1 un message est correctement envoyé
//2 le message est perdu

[send1] (c1=0) & (c2=0) ->  c1'=1 ;
[send2] (c2=0) & (c1=0) ->  c2'=1 ;

//On envoie un message mais le canal est occupé

[send1] (c1=0) & (c2>0) ->  (c1'=2) ;
[send2] (c2=0) & (c1>0) ->  (c2'=2) ;

//On vérifie que notre message est passe

[verify1] (c1=1) & (c2=2) -> (c1'=2) ;
[verify2] (c1=2) & (c2=1) -> (c2'=2) ;
[verify1] (c1=1) & (c2=0) -> (c1'=1) ;
[verify2] (c1=0) & (c2=1) -> (c2'=1) ;

//Un message se termine

[finish1] (c1>0) -> (c1'=0) ;
[finish2] (c2>0) -> (c2'=0) ;

endmodule

//Le canal est occupé

formula busy = (c1>0) | (c2>0) ;

//le canal est libre

formula free = (c1=0) & (c2=0) ;

//le canal est brouillé

formula garbled =  (c1=2) | (c2=2) ;
```

```
//Un module pour modéliser le temps discret qui se synchronisera
//avec les actions étiquetées par time

module temps

t : [0..tempsmax] init 0;

[time] t<tempsmax -> t'=t+1;

endmodule

module Sender1

//horloges du sender1
h1 : [0..lambda] init 0;

//temps aléatoire d'attente
alea1 : [0..timeMax] init 0;

//Nombres de collisions du sender1
col1 : [0..6] init 0;\\

//états du sender1
s1 : [0..8] init 0;

//0: Initialisation
//1: envie d'envoyer un message
//2: Envoi du message période d'attente 1
//3: Envoi du message période d'attente 2
//4: test au bout de 2sigma
//5: Envoi du message, sur qu'il passera
//6: Collision détectée
//7: Calcul du temps aléatoire
//8: Message envoyé correctement

//variables servant à faire le calcul du temps d'attente aléatoire
nbc1 : [0..6] init 0;
calcul1 : [0..1] init 0;
n1 : [0..1] init 0;
n2 : [0..1] init 0;
n3 : [0..1] init 0;
n4 : [0..1] init 0;
n5 : [0..1] init 0;
n6 : [0..1] init 0;

//passage à l'état critique, message à envoyer
[time] (s1=0) -> 0.5 : (s1'=1) + 0.5 : (s1'=0);
// on attend que le canal soit libre :
[time] (s1=1) & busy & (h1=0)  -> (s1'=1);
//On passe à l'envoi du message
[] (s1=1) & free & (h1=0) -> (s1'=2) & (h1'=sigma);

//le temps passe
```

```
[time] (s1=7) & (h1>0) & (alea1>0) -> (alea1'=alea1-1) ;
[time] (s1=7) & (h1>0) & (alea1=0) -> (h1'=0) ;
[time] (s1=2) & (h1>0) -> (h1'=h1-1) ;
[time] (s1=3) & (h1>0) -> (h1'=h1-1) ;
[time] (s1=5) & (h1>0) -> (h1'=h1-1) ;
[time] (s1=6) & (h1>0) -> (h1'=h1-1) ;

//Au bout de sigma unités de temps, le message est envoyé
[send1] (s1=2) & (h1=0) & free -> (s1'=3) & (h1'=sigma);
[send1] (s1=2) & (h1=0) & busy -> (s1'=6) & (h1'=sigma) & (col1'=min(6,col1+1))
& (calcul1'=1);  //Il y a eu collision
[verify1] (s1=3) & (h1=0) -> (s1'=4); // On va vérifier au bout de 2sigma
[] (s1=4) & !garbled -> (s1'=5) & (h1'=lambda-sigma); //le message passera
[finish1] (s1=5) & (h1=0) -> (s1'=8); //le temps passe le message est passé
[] (s1=8) -> (s1'=0) & (col1'=0); //retour à l'état d'origine
[] (s1=4) & garbled -> (s1'=6) & (h1'=sigma) & (col1'=min(6,col1+1)) & (calcul1'=1);
//il y a eu collision

//Calcul du temps aléatoire

[](s1=6) & (calcul1=1) & (nbc1=0) & (h1=0) -> (nbc1'=col1);

[]s1=6 & nbc1=6 -> 0.5 : (n6'=0) & (nbc1'=nbc1-1) + 0.5 : (n6'=1) & (nbc1'=nbc1-1);
[]s1=6 & nbc1=5 -> 0.5 : (n5'=0) & (nbc1'=nbc1-1) + 0.5 : (n5'=1) & (nbc1'=nbc1-1);
[]s1=6 & nbc1=4 -> 0.5 : (n4'=0) & (nbc1'=nbc1-1) + 0.5 : (n4'=1) & (nbc1'=nbc1-1);
[]s1=6 & nbc1=3 -> 0.5 : (n3'=0) & (nbc1'=nbc1-1) + 0.5 : (n3'=1) & (nbc1'=nbc1-1);
[]s1=6 & nbc1=2 -> 0.5 : (n2'=0) & (nbc1'=nbc1-1) + 0.5 : (n2'=1) & (nbc1'=nbc1-1);
[]s1=6 & nbc1=1 -> 0.5 : (n1'=1) & (nbc1'=nbc1-1) & (calcul1'=0) + 0.5 : (n1'=1)
& (nbc1'=nbc1-1) & (calcul1'=0);
[](s1=6) & (nbc1=0) & (calcul1=0) -> (alea1'=n1+2*n2+4*n3+8*n4+16*n5+32*n6)
& (n1'=0) & (n2'=0) & (n3'=0) & (n4'=0) & (n5'=0) & (n6'=0) & (s1'=7) & (h1'=1);
[finish1] (s1=7) & (h1=0) -> (s1'=1) & (h1'=0);
//le canal se vide, on retourne en attente mais apres un temps aléatoire

endmodule

module

Sender2=Sender1 [h1=h2,alea1=alea2,col1=col2,s1=s2,nbc1=nbc2,calcul1=calcul2,
n1=m1,n2=m2,n3=m3,n4=m4,n5=m5,n6=m6,finish1=finish2,send1=send2,verify1=verify2]

endmodule\\
```