Modélisation d'un système d'entrées-sorties SCSI-2

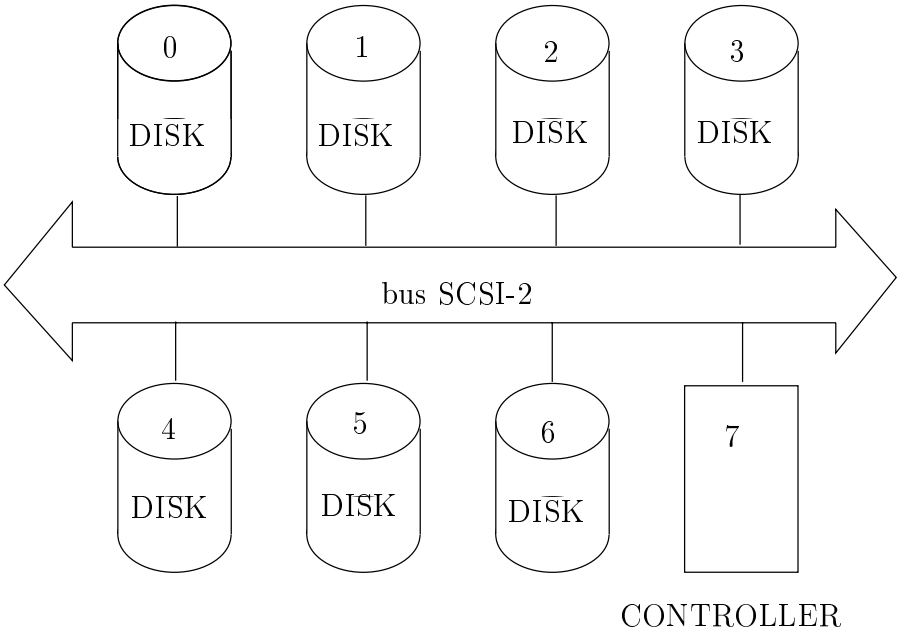
Version 2.0 - Hubert Garavel et Radu Mateescu

Action VERDON

1 Introduction

On veut modéliser en LOTOS le fonctionnement d'un système d'entrées-sorties reposant sur le bus SCSI-2 (norme ANSI X3.131-1989). Cet exemple s'inspire d'une modélisation LOTOS faite par Massimo Zendri (Bull Italie).

L'architecture du système est illustrée ci-dessous. Sur le bus SCSI, on peut connecter jusqu'à huit périphériques. Chaque périphérique est repéré par un numéro unique, compris entre 0 et 7 inclus. On considérera ici deux types de périphériques : disques et contrôleurs. On supposera ici que le système comporte sept disques (auxquels sont attribués les numéros SCSI de 0 à 6) et un contrôleur (dont le numéro SCSI est 7).



2 Ports de communication

Les communications sont modélisées au moyen de trois portes LOTOS de noms respectifs `CMD`, `REC` et `ARB` :

- L'action "`CMD !`n" (`CMD` signifiant *Commande*) indique que le contrôleur envoie au $n^{\text{ème}}$ disque une requête de transfert. En pratique, cette requête peut être de plusieurs types : demande de lecture d'un bloc de données sur le disque, demande d'écriture d'un bloc de données, etc. Ici, on ne cherchera pas à modéliser le type de la requête, ni les paramètres correspondants.
- L'action "`REC !`n" (`REC` signifiant *Reconnect* en terminologie SCSI) indique que le $n^{\text{ème}}$disque a fini de traiter une requête de transfert et renvoie au contrôleur le résultat correspondant (un bloc de données dans le cas d'une demande de lecture, une indication de terminaison dans le

cas d'une demande d'écriture, etc.) Ici encore, on ne cherchera pas à modéliser les résultats renvoyés.

- Les actions "`CMD !`n" et "`REC !`n" transitent par le bus, dont on rappelle qu'il est partagé entre les huit périphériques. Pour éviter les conflits d'accès au bus, la norme SCSI définit une politique d'arbitrage assurant qu'à un instant donné, un seul périphérique est autorisé à émettre sur le bus.

  Cette politique d'arbitrage est basée sur les priorités : si plusieurs périphériques veulent simultanément accéder au bus, c'est le périphérique dont le numéro SCSI est le plus grand qui l'emporte.

  Cette politique d'arbitrage est aussi décentralisée : contrairement à d'autres types de bus, il n'y a pas d'arbitre centralisé pour distribuer les autorisations d'accès au bus. Chaque périphérique doit surveiller les demandes émanant des autres périphériques et n'accéder au bus que si aucun autre périphérique plus prioritaire n'a effectué de demande en même temps.

  Physiquement, l'arbitrage est implémenté par huit fils dont chaque périphérique peut consulter la valeur (niveau électrique haut ou bas). Tout périphérique de numéro n désirant accéder au bus doit porter le $n^{\text{ème}}$ fil au niveau haut. Des fonctions de logique combinatoire implémentées sur chaque périphérique décident, après examen des huit fils pendant un certain temps (appelé *période d'arbitrage*), quel périphérique peut accéder au bus.

  On modélisera en LOTOS ces huit fils par une action "`ARB !WIRE` $(r_0, r_1, r_2, \ldots, r_7)$" qui signifie que, pendant la période d'arbitrage considérée (que l'on supposera être suffisamment brève pour être représentée par une action atomique), les fils de numéros 0 à 7 ont les valeurs booléennes respectives r_0 à r_7 (la valeur *true* correspondant au niveau électrique haut), et où `WIRE` est un constructeur de type tuple (nommé aussi `WIRE`) permettant d'obtenir un vecteur de huit booléens.

3 Architecture du système

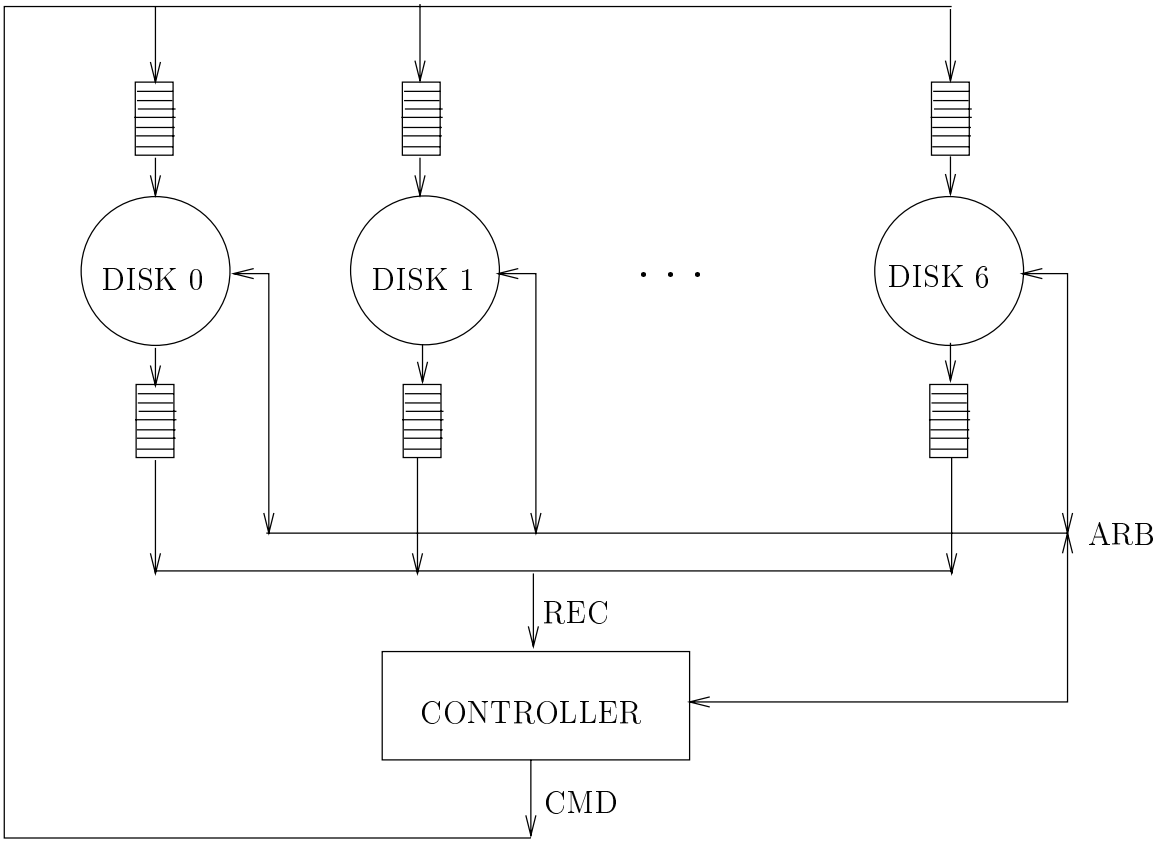
Pour spécifier le comportement du système, on adopte une approche descendante, par raffinements successifs. L'architecture du système est représentée sur la figure ci-dessous qui met en évidence les synchronisations et les communications entre les éléments du système.

Dans cette modélisation, le bus lui-même n'apparaît pas : on considère que les sept disques communiquent directement avec le contrôleur au moyen de rendez-vous binaires sur les portes `CMD` et `REC`. Comme indiqué plus haut, le serveur émet des requêtes sur la porte `CMD` et attend des réponses sur la porte `REQ`. Les disques étant en concurrence pour dialoguer avec le serveur sur ces portes, c'est le numéro SCSI transmis sur les portes `CMD` et `REC` qui permettra d'identifier le disque concerné.

Mais, pour respecter la discipline d'arbitrage centralisé, tout périphérique désirant envoyer un message `CMD` ou `REC` doit auparavant obtenir l'accès au bus. Cet accès est modélisé par un rendez-vous à huit sur la porte `ARB`. En revanche, le périphérique qui reçoit un tel message n'a pas à obtenir le bus pour ce faire, puisque c'est l'émetteur qui s'en charge.

Pour accélerer le débit, les messages `CMD` émis par le contrôleur et les messages `REC` émis par les disques sont stockés dans des files d'attente (*SCSI buffers*) qui comportent chacune huit places (voir figure ci-dessus).

Compte-tenu du fait que l'on ne modélise pas les paramètres attachés aux messages `CMD` et `REC`, tous les messages contenus dans une file d'attente donnée sont identiques. Il n'est donc pas nécessaire de représenter avec exactitude le contenu des différentes files : il suffit de compter le nombre de messages qu'elles contiennent. C'est pourquoi, les files d'attente n'apparaîtront pas explicitement dans la modélisation.



Le comportement des disques est décrit par un processus générique `DISK`. Ce processus est paramétré par le numéro SCSI du disque et par le nombre de messages `CMD` en attente de traitement (initialement nul).

Le comportement du contrôleur est décrit par un processus `CONTROLLER` qui est paramétré par une variable mémorisant le nombre de messages `CMD` en attente de traitement par les disques ; la valeur initiale de cette variable est une constante notée `ZERO` correspondant à la situation dans laquelle aucun message n'a encore été envoyé aux disques.

Le corps de la spécification LOTOS décrivant l'architecture du système est donc :

```
specification SCSI_2 [ARB, CMD, REC] : noexit :=
    (* definitions de types *)
 behaviour
   (
   DISK [ARB,CMD,REC] (0 of NUM, 0 of NAT)
   |[ARB]|
   DISK [ARB,CMD,REC] (1 of NUM, 0 of NAT)
   |[ARB]|
   DISK [ARB,CMD,REC] (2 of NUM, 0 of NAT)
   |[ARB]|
   DISK [ARB,CMD,REC] (3 of NUM, 0 of NAT)
   |[ARB]|
   DISK [ARB,CMD,REC] (4 of NUM, 0 of NAT)
   |[ARB]|
   DISK [ARB,CMD,REC] (5 of NUM, 0 of NAT)
   |[ARB]|
   DISK [ARB,CMD,REC] (6 of NUM, 0 of NAT)
```

```
  )
  |[ARB, CMD, REC]|
  CONTROLLER [ARB,CMD, REC] (7 of NUM, ZERO)
where
   (* definitions des processus CONTROLLER et DISK *)
endspec
```

4 Demandes d'arbitrage

Comme indiqué plus haut, la politique d'arbitrage distribuée est mise en œuvre par un rendez-vous sur la porte `ARB` auquel les huit périphériques participent.

On cherche à exprimer ce rendez-vous de manière similaire pour chacun des huit périphériques, ceci afin de permettre l'écriture d'un processus `DISK` générique, paramétré par le numéro SCSI du disque.

Pour cela, on notera, dans la description du périphérique de numéro SCSI n (c'est-à-dire à la fois dans les processus `CONTROLLER` et `DISK`) les rendez-vous sur la porte `ARB` de la manière suivante :

$$\texttt{ARB ?}w\texttt{:WIRE [}C(w,n)\texttt{]}$$

où w est un vecteur de la forme `WIRE` $(r_0,\ldots,r_7)$ et $r_0,\ldots,r_7$ désignent les valeurs des huit fils. $C(w,n)$ est un prédicat valant *true* ssi le $n^{\text{ème}}$périphérique autorise le rendez-vous. Il s'agit d'une programmation par *filtrage* : chaque périphérique pose ses propres contraintes (exprimées par le prédicat C) pour accepter le rendez-vous, lequel n'aura lieu que si les contraintes posées par les huit périphériques sont satisfaites.

Pour la suite, on utilisera deux prédicats $C(w,n)$ particuliers, notés respectivement $P(w,n)$ et $A(w,n)$:

- $P(w,n)$ caractérise la situation dans laquelle le $n^{\text{ème}}$ périphérique indique qu'il ne désire pas accéder au bus. Ceci s'obtient en donnant au $n^{\text{ème}}$ bit de w la valeur *false*:

$$P(\texttt{WIRE }(r_0,\ldots,r_7),n) = \neg r_n$$

- $A(w,n)$ caractérise la situation dans laquelle le $n^{\text{ème}}$ périphérique indique qu'il désire accéder au bus et constate (en examinant les demandes des autres périphériques) qu'il y est autorisé. Ceci s'obtient en donnant au $n^{\text{ème}}$ bit de w la valeur *true*, sous réserve que tous les sites de numéro strictement supérieur à n n'aient pas demandé le bus :

$$A(\texttt{WIRE }(r_0,\ldots,r_7),n) = r_n \ \wedge\ \neg(r_{n+1}\ \vee\ \ldots\ \vee\ r_7)$$

5 Comportement du contrôleur

Le processus `CONTROLLER` gère une variable interne C de type `CONTENTS`. Ce type peut être vu comme un tableau d'entiers naturels indexé par un numéro SCSI (différent du numéro du contrôleur). A un instant donné, l'élément de C d'indice n mémorise le nombre de commandes en attente de traitement par le $n^{\text{ème}}$ disque, c'est-à-dire la différence entre le nombre de messages `CMD !`n émis et le nombre de messages `REC !`n reçus.

Le contrôleur doit implémenter un *mécanisme de contrôle de flux*, de manière à garantir que les files ne débordent pas, c'est-à-dire que le nombre de requêtes en attente pour un disque donné est toujours inférieur ou égal à huit.

On dispose des primitives suivantes pour manipuler les valeurs de type `CONTENTS` :

- La constante `ZERO` renvoie un tableau dont toutes les valeurs sont égales à zéro.
- La fonction `NOT_FULL` (C, n) renvoie un résultat booléen qui vaut *true* ssi l'élément d'indice n du tableau C est strictement inférieur à huit.
- La fonction `ALL_FULL` (C) renvoie un résultat booléen qui vaut *true* ssi tous les éléments du tableau C sont égaux à huit (ce qui signifie que toutes les files des disques sont pleines).
- La fonction `INCR` (C, n) renvoie le tableau C dont la valeur de l'élément d'indice n a été incrémentée.
- La fonction `DECR` (C, n) renvoie le tableau C dont la valeur de l'élément d'indice n a été décrémentée.

Informellement, le comportement du processus `CONTROLLER` peut être décrit comme une sélection non-déterministe entre différentes actions :

- Si cela est possible, le contrôleur peut spontanément décider d'envoyer des messages `CMD` aux disques.
- Le contrôleur peut accepter des messages `REC` ; on supposera que le contrôleur "fait confiance" aux disques et ne cherche pas à se prémunir contre la situation dans laquelle un disque enverrait plus de messages `REC` qu'il n'a reçu de messages `CMD`.
- Enfin, lorsque toutes les files des disques sont pleines, le contrôleur cesse d'utiliser le bus, ce qui permet aux disques ayant une priorité inférieure d'y accéder.

Le processus LOTOS ci-dessous décrit le comportement du contrôleur (la variable NC dénotant le numéro SCSI du contrôleur) :

```
process CONTROLLER [ARB,CMD,REC] (NC:NUM, C:CONTENTS) : noexit :=
   choice N:NUM []
      [N <> NC] ->
         [NOT_FULL (C, N)] ->
            (
            ARB  ?W:WIRE [A (W, NC)];
            (* il demande le bus et il l'obtient *)
               CMD !N ;
                  CONTROLLER [ARB, CMD, REC] (NC, INCR (C, N))
            []
            ARB  ?W:WIRE [not (A (W, NC)) and not (P (W, NC))];
               (* il demande le bus et il ne l'obtient pas *)
                  CONTROLLER [ARB, CMD, REC] (NC, C)
            )
   []
   REC ?N:NUM [N <> NC] ;
      CONTROLLER [ARB,CMD,REC] (NC, DECR (C,N))
   []
   [ALL_FULL (C)] ->
      ARB ?W:WIRE [P (W, NC)];
         CONTROLLER [ARB, CMD, REC] (NC, C)
endproc
```

6 Comportement du disque

Le comportement du processus `DISK` peut être décrit comme une sélection non-déterministe entre différentes actions. En particulier :

- Il peut accepter les messages `CMD` qui lui sont adressés. On suppose que le disque fait confiance au mécanisme de contrôle de flux réalisé par le contrôleur, et ne cherche donc pas à se prémunir contre les débordements de sa file d'entrée.
- S'il y a un (ou plusieurs) messages `CMD` dans la file d'entrée, le processus `DISK` doit en prélever un et le traiter. Pour cela, le disque doit réclamer constamment l'accès au bus jusqu'à ce qu'il l'obtienne. Le traitement d'un message `CMD` est simplement modélisé par l'envoi du message `REC` correspondant.
- S'il n'y a aucun message `CMD` dans la file d'entrée, le processus `DISK` ne demande pas l'accès au bus.

```
process DISK [ARB, CMD, REC](N:NUM, L:NAT) : noexit :=
   [L > 0 of NAT] ->
      (
      ARB  ?W:WIRE [A (W, N)];
         (* il demande le bus et l'obtient *)
            REC !N;
               DISK [ARB, CMD, REC] (N, L-1)
      []
      ARB  ?W:WIRE [not (P (W, N)) and not (A (W, N))];
         (* il demande le bus et ne l'obtient pas *)
            DISK [ARB, CMD, REC] (N, L)
      )
   []
   [L = 0 of NAT] ->
      ARB  ?W:WIRE [P (W, N)];
         (* il ne demande pas le bus *)
            DISK [ARB, CMD, REC] (N, L)
   []
   CMD !N;
      DISK [ARB, CMD, REC] (N, L+1)
endproc
```