



Parallélisme

TP

Compte Rendu

Taavi Pehme
Laurent Berthet

Mai 2007

Sommaire

Sommaire.....	2
Introduction.....	3
1. La spécification.....	4
2. La démarche Erlang.....	11
2.1 La traduction de la spécification vers Erlang	11
2.2 L'architecture générale	13
Conclusion.....	15

Introduction

L'objectif de ce TP est créer une application parallèle qui devra simuler le fonctionnement d'un système composée des composants suivants :

1. Un ventilateur
2. Une pompe d'huile du moteur du ventilateur
3. Un voyant alarme
4. Un voyant indiquant que le ventilateur ne marche pas
5. Un bouton pour lancer le système.

On ne va pas ici présenter la tâche donnée par le professeur, on estime que celui qui va consulter ce rapport connaît déjà le sujet.

Ce TP est divisé dans deux parties. La première concerne la spécification du système écrit en CCS et la deuxième va présenter l'application correspondante programmée en Erlang.

Dans la deuxième partie on va expliquer, comment on a traduit la spécification dans le langage réel de programmation, et combien la spécification et l'application se différent.

1. La spécification

Dans ce paragraphe on va présenter la spécification du système utilisant CCS.

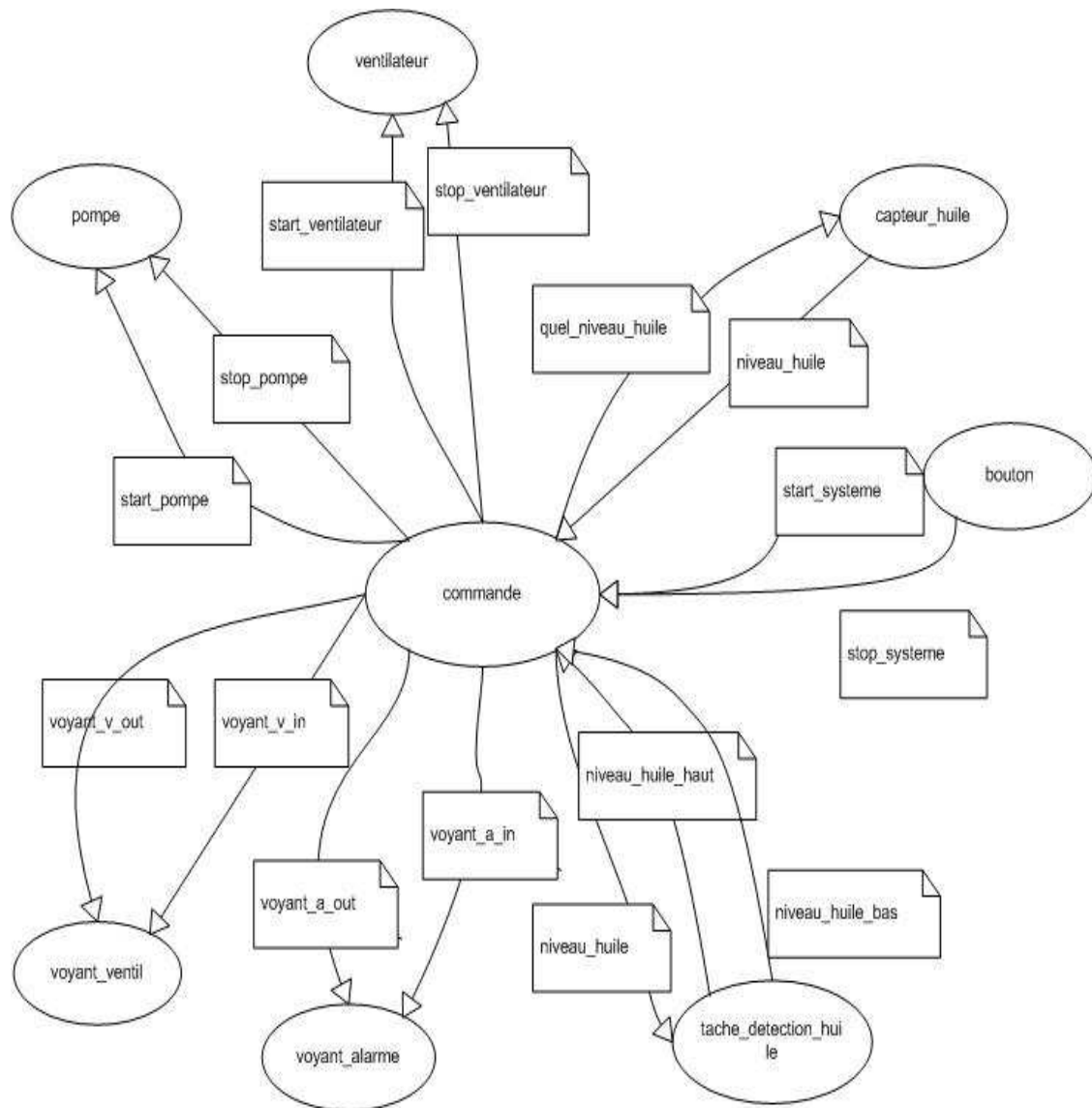


Figure 1 - schéma du système

- **Ventilateur** reçoit des messages pour se mettre en route ou s'arrêter :

```
ventilateur(start_ventilateur, stop_ventilateur) =
start_ventilateur.ventilateur( start_ventilateur,
sop_ventilateur)
+
stop_ventilateur.ventilateur( start_ventilateur,
stop_ventilateur).
```

- **Pompe** (ressemble au ventilateur). Elle aussi reçoit des messages pour se mettre en route ou pour s'arrêter.

```
pompe(start_pompe, stop_pompe) =
start_pompe.pompe( start_pompe, stop_pompe)
+
stop_pompe.pompe( start_pompe, stop_pompe).
```

- **Capteur d'huile** vérifie la pression d'huile dans le réservoir de la pompe et s'il reçoit un message de demande de la pression d'huile, il répond en renvoyant la pression courante.

```
capteur_huile( quel_niveau_huile, niveau_huile ) =
quel_niveau_huile.niveau_huile(Niv).capteur_huile(
quel_niveau_huile, niveau_huile )
```

- **Voyant alarme** reçoit des messages pour s'allumer et s'éteindre.

```
voyant_alarme(voyant_a_in, voyant_a_out) =
```

```

voyant_a_in.voyant_alarme( voyant_a_in,
voyant_a_out)
+
voyant_a_out.voyant_alarme( voyant_a_in,
voyant_a_out)

```

- **Voyant ventilateur** ressemble à voyant alarme, il reçoit des messages pour s'allumer et s'éteindre.

```

voyant_ventilateur(voyant_v_in,voyant_a_out) =
voyant_v_in.voyant_ventilateur(
voyant_v_in,voyant_v_out)
+
voyant_v_out.voyant_ventilateur(
voyant_v_in,voyant_v_out)

```

- **Bouton** est un interrupteur pour lancer ou arrêter la séquence

```

bouton(start_system,stop_system) =
start_system.bouton(start_system, stop_system)
+
stop_system.bouton(start_system,stop_system)

```

- **Tache détection huile** : va boucler et dans chaque pas de la boucle il va demander au capteur le niveau courant et puis va l'envoyer à commande.

```

tache_detection_huile(quel_niveau_huile,
niveau_huile,niveau_huile_haut,
niveau_huile_bas) =
quel_niveau_huile.niveau_huile(Niv).
( SI

```

```

Niv >= niveau_suffisant
    ALORS
Niveau_huile_haut.tache_detection_huile(quel_niveau_huile
,niveau_huile,
niveau_huile_haut,niveau_huile_bas)
    SINON
Niveau_huile_bas.tache_detection_huile(quel_niveau_huile,
niveau_huile,
niveau_huile_haut, niveau_huile_bas)
)

```

- **Pompe timer** : reçoit un message de lancer le timer avec une certaine période et après le fin de cette période va notifier celui qui à posé la demande.

```

pompe_timer(wait_t,limite_de_temps) =
wait_t(t).Δt.limite_de_temps.
pompe_timer(wait_t,limite_de_temps)

```

- **Commande** : est le processus central qui va diriger le fonctionnement du système. Il reçoit les informations sur la pression d'huile est va prendre les décisions sur le déclanchement et l'arrêt de la pompe et du ventilateur, l'allumage et l'extinction des voyants. Pour faire la différence entre les étapes dans le travaille du système, la commande utilise la variable *Status* qui correspond au l'état courant du système. Les états sont les suivants :
 1. *stat_stop* – le fonctionnement du système est arrêté.
 2. *stat_pompe* – seulement la pompe est mise en route pour 30 secondes.
 3. *stat_verif* – les 30 secondes sont passées et on vérifie la pression d'huile.
 4. *stat_ventilo* – la pression d'huile était suffisante et le ventilateur est mis en route. La pompe continue à tourner.

```

        commande(Status, start_system, stop_system, limite_de
            temps,
niveau_huile_haut,
niveau_huile_bas,
voyant_v_in,
    voyant_v_out,
voyant_a_in, voyant_a_out,
start_pompe,
stop_pompe,
start_ventilateur,
stop_ventilateur,
wait_t) =
start_system.voyant_v_in.voyant_a_out.start_pompe
    .wait_t(30).commande(stat_pompe, _____)
    +
stop_system.voyant_v_out.voyant_a_out.stop_pompe
    .stop_ventilateur.commande(stat_stop, _____)
    +
limite_de_temps.commande(stat_verif, _____)
    +
niveau_huile_haut.( SI
Status = stat_verif
                ALORS
voyant_v_out.start_ventilateur.commande(stat_ventilo,
    _____)
                SINON
nil
)
    +
niveau_huile_bas.( SI
Status = stat_verif ou Status = stat_ventilo

```

```

                                ALORS
stop_pompe.stop_ventilateur.voyant_v_in.
voyant_a_in.commande(stat_stop, _____)
                                SINON
nil
)

```

- **Le système** contient tous les processus en parallèle.

```

Systeme(<TOUS LES CANNAUX>) =
    ventilateur(start_vetilateur,
                stop_ventilateur) ||
pompe(start_pompe,
        stop_pompe) ||
voyant_ventilateur(voyant_v_in,
                    voyant_a_out) ||
voyant_alarme(voyant_a_in,
               voyant_a_out) ||
bouton(start_system,
        stop_system) ||
capteur_huile(quel_niveau_huile,
               niveau_huile) ||
tache_detection_huile(quel_niveau_huile,
                       niveau_huile,
                       niveau_huile_haut,
                       niveau_huile_bas) ||
pompe_timer(wait_t,
             limite_de_temps) ||
commande(Status,
         start_system,
         stop_system,
         limite_de_temps,

```

```
niveau_huile_haut,  
niveau_huile_bas,  
voyant_v_in,  
voyant_v_out,  
voyant_a_in,  
voyant_a_out,  
start_pompe,  
stop_pompe,  
start_ventilateur,  
stop_ventilateur,  
wait_t)  
\ {start_system, stop_system, limite_de temps,  
niveau_huile_haut,  
niveau_huile_bas,  
voyant_v_in,  
voyant_v_out,  
voyant_a_in,  
voyant_a_out,  
start_pompe,  
stop_pompe,  
start_ventilateur,  
stop_ventilateur,  
wait_t,  
quel_niveau_huile}
```

2. La démarche Erlang

Dans ce chapitre on va s'occuper de la traduction de la spécification à Erlang. Notre but était d'implémenter l'application de la façon qu'il aurait semblé le plus possible à la spécification. A la fois, s'il y avait une différence énorme d'optimisation entre la spécification et les possibilités en Erlang, on a donc changé un peu et on a appliqué les astuces d'Erlang pour optimiser notre travail. On va d'abord présenter la traduction des modules de base de CCS vers Erlang et après on va décrire l'architecture générale de l'application et le concept d'intégration d'une interface pour l'interaction avec des utilisateurs.

2.1 La traduction de la spécification vers Erlang

Les principes de traduction sont en fait assez simples. On crée un module d'Erlang pour chaque processus de CCS. Chaque module sera stocké dans son propre fichier. Comme les processus sont permanents, ca veut dire qu'en CCS ils sont décrits sous la forme d'une boucle, ils auront le même comportement en Erlang. Puis, pour simuler aussi les objets réels, comme la source d'huile, il faut qu'on rajoute quelques modules qui vont s'en occuper. A cause de cela, il faut aussi rajouter dans les modules spécifiés le code supplémentaire pour gérer la communication vers les nouveaux modules.

Les fichiers/modules de bases générées sont les suivants :

- *Ventilateur ... ventilateur.eri* – la traduction de ce module est assez précise. Pour afficher les changements de l'état à l'utilisateur, quand le ventilateur se déclenche ou s'éteint, il envoie un message correspondant au module « membrane » qui est une couche virtuelle entre le model et l'interface. Tous les changements qui doivent être affichés à l'utilisateur sont passés par « membrane ».

- *Pompe ... pompe.erl* - la traduction de la partie spécifiée est assez précise. On a rajouté deux principaux morceaux de code supplémentaires. Le premier est la réception de message «changer_pompe ». Cela est nécessaire pour que la pompe puisse tourner de façon inverse : à place d'augmenter la pression , elle va la diminuer. Cette fonctionnalité permet aux utilisateurs de tester la situation d'erreur qui peut arriver quand la pompe et le ventilateur tournent et la pression d'huile tombe. Ca va devenir plus claire quand on va décrire l'interface. Le deuxième morceau de code fait la pompe, quand elle tourne, constamment envoyer les messages à la source d'huile pour le faire diminuer. Il est tout à fait nécessaire de faire comme ca pour simmuler un réservoir réel qui se vide et se rempli à cause du fonctionnement de la pompe.
- *Capteur_huile ... capteur_huile.erl* – ici on peut dire qu'on a réussi à faire une traduction « directe ». Le capteur attend un message de demande de niveau et va lui répondre en envoyant le niveau courant.
- *Voyant_alarme ... voyant_alarme.erl* et *Voyant_ventilateur ... voyant_ventilateur.erl* - les deux modules sont pareils. Ils reçoivent des messages d'allumage et d'extinction. Pour afficher les changements à l'utilisateur, ils renvoyaient les messages à membrane, comme c'était décrit précédemment dans le ventilateur.
- *Boutton ... bouton.erl* – Dans la spécification le bouton est présenté comme un processus qui va envoyer un message soit pour démarrer, soit pour arrêter le système. Après la traduction, le bouton n'a pas changé mais on a rajouté la réception de deux messages venant de l'utilisateur qui doit apparemment donner l'ordre pour lancer la séquence.
- *Pompe_timer ... pompe_timer.erl* – La traduction d'horloge qui mesure le temps pour pomper et assez précis. L'horloge reçoit la demande, il attend la

période de temps demandé et il notifie à celui qui a demandé que le temps est passé.

- *Tache_detection_huile ... tache_detection_huile.erl* – le module de Erlang correspond assez précisément à la spécification.
- *Commande ... commande.erl* – L’algorithme de prise de décisions dans le module principal est transformé au code Erlang assez précisément.
- *Système ... Application Erlang* – On va décrire la façon comment Les processus sont lancés en parallèle dans le paragraphe d’architecture générale.

2.2 L’architecture générale

Pour cette application on a utilisé l’architecture générale d’Erlang. En fait, les modules d’application sont divisés en deux parties. La première partie, ce sont les processus de base, qui exécutent les actions de l’application. L’autre partie se constitue en des processus « superviseurs » qui ont la responsabilité de lancer et contrôler les processus de base.

Ici, on a le superviseur qui se place le plus haut et qui lance les autres processus « superviseur », il s’appelle « main_sup ». Après on a le superviseur pour chaque partie logique de l’application qui lance les processus de base appartenant dans cette catégorie. On va donner la liste des superviseurs avec la liste des processus de base qui sont influencés par ce superviseur, et une description, si, ce n’était déjà pas fait avant.

- **Main_sup** – le superviseur qui lance tous les autres superviseurs.
- **Capteur_sup** – ce superviseur s’occupe seulement le démarrage et le contrôle de *capteur_huile*

- **Tache_detection_sup** – il contrôle la boucle dans le module *tache_detection_huile* qui notifie la Commande du niveau de l'huile.
- **Commande_sup** – il s'occupe du module principal *commande* et les modules qui font la partie plus principal du système, comme :
pompe, ventilateur, voyat_alarme, voyant_ventilateur, bouton, pompe_timer
- **Interface_sup** – Ce superviseur s'occupe du module *interface* qui est la représentation graphique du fonctionnement du système. Le module *interface* reçoit les messages de chaque module qui est présenté sur l'interface, est s'anime en prenant en compte l'information envoyée.
- **Membrane_sup** – En fait, les messages ne sont pas directement envoyés par des modules à l'interface. Il y a le module *membrane* qui est au milieu et qui fait les messages en provenance des modules s'adapter à l'interface. C'est bien pour vraiment distinguer la modèle et l'interface. *Membrane_sup* s'occupe à processus *membrane*.
- **Parametre_sup** – ce superviseur s'occupe au module *parametre*, qui est nécessaire pour la gestion de paramètres du système. Les valeurs comme le débit de la pompe et le nombre des seconds qu'on laisse la pompe tourner, sont lu du fichier *variable.eini*. Le module *prametre* s'occupe à la tâche de lire et sauvegarder les paramètres.

Il y a encore un module qui n'a pas de superviseur, c'est « *initialisation* ». Ce module est démarré avant la première exécution d'application. Ca crée le fichier des paramètres.

Le module qui représente l'application est « *main* ». Il est situé le plus haut dans l'hierarchie et il lance le superviseur *main*. Après ca, la séquence de lancement et déjà connue.

Conclusion

Dans ce TP on a réussi à spécifier l'application en CCS d'une manière assez précise et en suite l'implémenter utilisant le langage parallèle – Erlang.

Il est apparu que si on suit cette démarche de développement, au niveau de la conception, la traduction de CCS vers Erlang est assez précise. Ca veut dire que la partie spécifiée en CCS sera toujours présente dans le code d'Erlang. Il faut seulement rajouter dans les modules d'Erlang les éléments supplémentaires, et aussi les modules supplémentaires qui sont nécessaires pour suivre les principes de développement et les patrons d'architecture d'Erlang.

Au cours de ce TP on a appliqué la technique de CCS appris en TD, on a appris le nouveau langage Erlang, et on a créé un exemple qui montre une démarche possible pour développement d'une application parallèle.

Dans le rapport on a présenté la spécification, le concept Erlang et le code de l'application.