Chapitre XII- Un système d'arrosage automatique.

Objectifs du chapitre :

- Illustrer sur un exemple les possibilités de modélisation d'architectures hétérogènes (Esterel, Java).
- Donner un exemple d'intégration d'un composant Esterel complexes (tâches, signaux mixtes, multi-modules) et de son environnement asynchrone (Robinet, pompes, récipients).

Le système proposé en exemple pour illustrer l'utilisation d'un composant synchrone dans un environnement asynchrone a été présenté dans **[Gaf91]**. Il s'agit d'un système de contrôle d'un arroseur automatique. Ce contrôleur doit effectuer deux actions concurrentes :

1. Il contrôle la préparation du mélange à disperser, c'est-à-dire mélanger l'engrais avec de l'eau pour obtenir un mélange avec la concentration d'engrais souhaitée pour le type de plante à arroser. Pour cela, il dispose d'un capteur de concentration, d'une commande sur le robinet d'eau et d'une commande sur le robinet d'engrais. De plus, le contrôleur doit s'assurer qu'une quantité minimum d'engrais dilué est toujours disponible. Pour cela il dispose de deux capteurs de volume, un pour s'assurer qu'il y a suffisamment de mélange et l'autre pour s'assurer qu'il n'y en a pas trop.
2. Il doit aussi disperser le mélange sur les plantes. Pour cela, il dispose d'un actionneur pour commander l'utilisation d'une pompe et d'un capteur d'humidité qui indique le taux d'humidité des plantes arrosées afin d'assurer un arrosage suffisant.

Les caractéristiques des plantes à arroser sont lues dans une base de données à chaque ré-amorçage du système. La concentration idéale d'engrais est appelée concentration de référence et l'humidité minimum requise pour les plantes est appelée humidité de référence. Cet arroseur doit conserver une trace de son activité, c'est-à-dire mémoriser pour une configuration donnée (humidité de référence et concentration de référence) la quantité de mélange (eau et engrais) consommée.

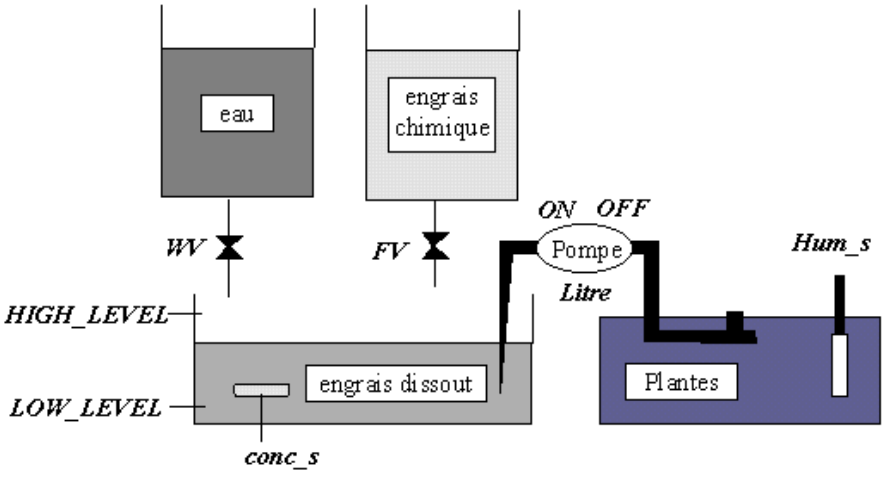

Figure 1 – Arroseur automatique.

La Figure 1 présente l'environnement que nous souhaitions modéliser. Les noms qui y apparaissent en gras sont les signaux de commande et les capteurs nécessaires pour le fonctionnement du contrôleur.

Cet exemple introduit de nouvelles difficultés. En effet, il fait intervenir des actions concurrentes, le langage Esterel est donc approprié pour sa description. De plus, il faut gérer une base de données et un environnement extérieur où les langages impératifs classiques sont mieux adaptés. En particulier, Java offre une bibliothèque de gestion des bases de données avec le protocole JDBC[1] qui est intéressante. Il est alors possible d'envisager une modélisation mixte Java/Esterel pour ce problème. Ce chapitre illustre cette modélisation dans notre environnement SEP.

Le contrôleur est modélisé dans le langage Esterel[2]. Il est ainsi possible de valider des propriétés sur ce code. En particulier, on peut s'assurer que le contrôleur ne demande jamais l'ouverture et la fermeture simultanée d'un robinet ou de la pompe. Si c'était le cas, l'environnement SEP ne saurait pas quel choix adopter.

Enfin le contrôleur synchrone doit prendre les informations spécifiques aux plantes qu'il arrose et constituer une base d'information sur son activité. Cet aspect est traité par des tâches asynchrones contrôlées par le module synchrone (cf. section 3-).

Pour la construction de l'environnement dans SEP, il est nécessaire d'introduire plusieurs composants supplémentaires.

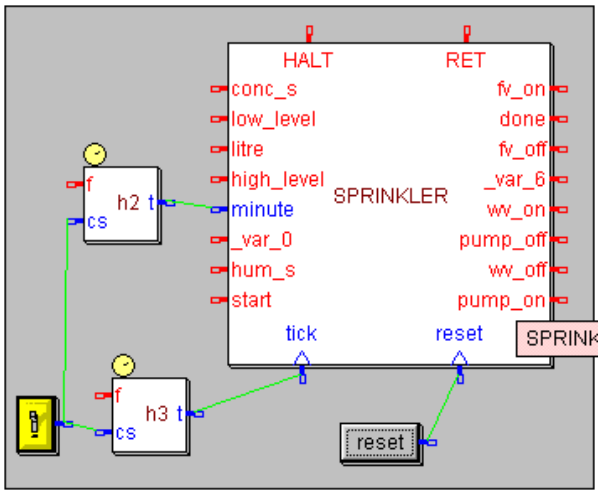

Figure 2 – Composant STRL de l'arroseur.

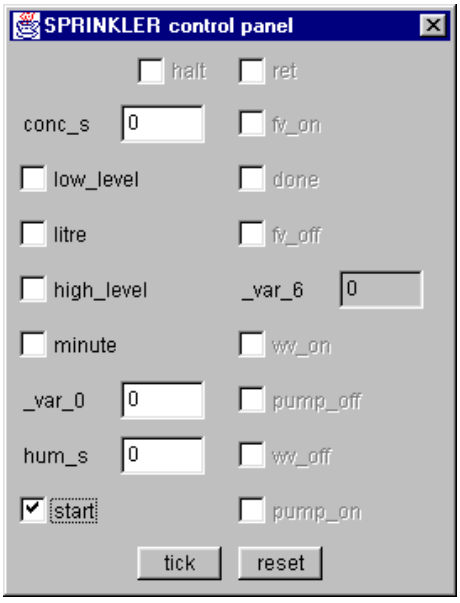

Figure 3 – Panneau de contrôle de l'arroseur.

1- Modélisation du contrôleur synchrone.

Le comportement du contrôleur est décrit en Esterel. Comme l'indique le chapitre IX, un composant générique nommé STRL permet l'intégration d'une machine d'exécution Esterel dans un composant SEP. C'est cette méthode qui est appliquée ici.

[1] Java Data Base Connectivity : Ce protocole permet une interaction via ODBC avec les principaux moteurs de base de données du commerce.

[2] Le code Esterel de ce contrôleur fourni par D. Gaffé **[Gaf91]** est disponible en annexe.

Le code Esterel du contrôleur est compilé dans le format Ssc. En effet, le format blif n'est pas suffisant puisque ce contrôleur met en œuvre des signaux mixtes, qui ne sont pas purs, ainsi que des appels à des tâches asynchrones.

Après avoir paramétré le composant STRL avec le fichier Ssc compilé, nous obtenons le composant de la Figure 2 ainsi que le panneau de contrôle de la Figure 3 qui permet une simulation pas à pas.

Remarque : Les signaux mixtes (dans l'exemple *start*) sont décomposés en un signal pur (ici *start*) et une valeur pure (ici *_var_0*).

2- Modélisation de l'environnement asynchrone du contrôleur.

2-1. Le composant récipient.

Le composant *récipient* (cf. Figure 4) propose deux services. Un service *ajouter* qui permet d'ajouter une quantité de liquide correspondant à l'entrée *in* et un service *retirer* qui permet d'enlever une quantité de liquide correspondant à l'entrée *out*. La sortie *quantité* représente la quantité de liquide contenu dans le récipient. Un paramètre *content* permet d'initialiser le contenu.

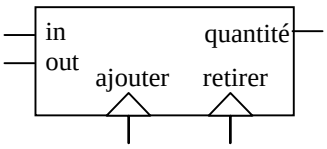

Figure 4 – Le composant récipient.

Voici le fournisseur de services *Recipient* :

```
public class Recipient implements sep.model.ServiceProvider {
        protected Value quantite = new IntValue(0);
        // ---- Services -----
        public Value ajouter(Value val) {
          return quantite = sep.type.Type.perform("add", quantity, val);
        }
        public Value retirer(Value val) {
          return quantite = sep.type.Type.perform("sub", quantity, val);
        }

        // ----- paramètre -----
        public Value getContent () { return quantite; }
        public void setContent(Value val) { quantite = val; }
}
```

Le service *ajouter* est déclenché avec une fonction de commande de type *posedge* sur le port *ajouter*. Le paramètre *val* est associé au port d'entrée *in*. La valeur de retour est associée au port de sortie *quantité*.

Le service *retirer* est déclenché avec une fonction de commande de type *posedge* sur le port *retirer*. Le paramètre *val* est associé au port d'entrée *out*. La valeur de retour est associée au port de sortie *quantité*.

Les deux services ont la même priorité, ce qui signifie que s'ils sont déclenchés simultanément au même instant il faut utiliser une fonction de résolution pour résoudre le conflit écriture-écriture. La fonction de résolution utilisée garde la dernière valeur calculée :

```
public class Last implements Resolution {
        public Value solve(Value [] values) { return values[values.length-1] ; }
}
```

Remarques :

1. Quel que soit l'ordre d'exécution des services le résultat est le même : (quantité + in) – out = (quantité – out) + in.
2. Un composant placé sur le port de sortie *quantité* peut émettre un signal d'alerte à l'intention du concepteur lorsque la quantité est inférieure à zéro ou supérieure à une valeur représentant le volume du récipient. Cet observateur ne voit que le résultat de la fusion.
3. Le type utilisé est *Value*. Il est ainsi possible d'utiliser ce composant avec des données de n'importe quel type.

2-2. Le composant *robinet*.

Le composant *Robinet* est un module (cf. Figure 5). Il a deux entrées *on* et *off*, et une sortie *out*. Le robinet s'ouvre par un front montant sur l'entrée *on* et se ferme par un front montant sur l'entrée *off*. Lorsqu'il est en position ouverte le robinet délivre sur sa sortie un signal périodique (horloge) de période paramétrable (ici 2×10=20 ms).

Les actions associées à l'ouverture du robinet peuvent alors être déclenchée sur les fronts montants du signal périodique de sortie.

Lorsque le robinet est fermé, la sortie est stable et conserve sa dernière valeur.

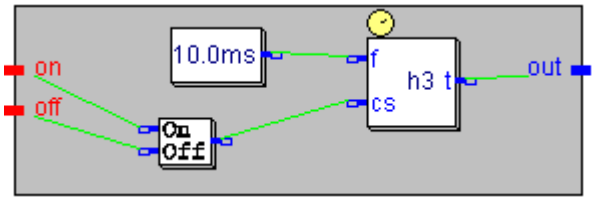

Figure 5 – Le composant robinet.

Ce module est constitué d'un composant *OnOff* et d'un composant *horloge*.

Le composant *OnOff* est un composant séquentiel à deux services déclenchés par une fonction de commande de type *posedge*. Son service *on* force la sortie à la valeur *High*, son service *off* force la sortie à la valeur *Low*. Le service *off* a une priorité supérieure au service *on*. Néanmoins, l'activation simultanée des deux services n'a pas beaucoup de sens. Cette activation étant générée par le contrôleur Esterel, elle peut être validée formellement par model-checking avec les outils de l'environnement synchrone.

Le composant *horloge* est un composant séquentiel de type *TimeComponent*. Lorsque son entrée *cs* a la valeur *High*, il émet un signal périodique de période 2.f définie par son entrée *f*.

Voici le code Java du fournisseur de service *Horloge* :

```
public class Horloge implements sep.model.ServiceProvider {
        protected LevelValue etat = LevelValue.Low ;
        public LevelValue tick( LevelValue cs) {
                if (cs.isHigh()) etat = etat.not();
                return etat;
 }
}
```

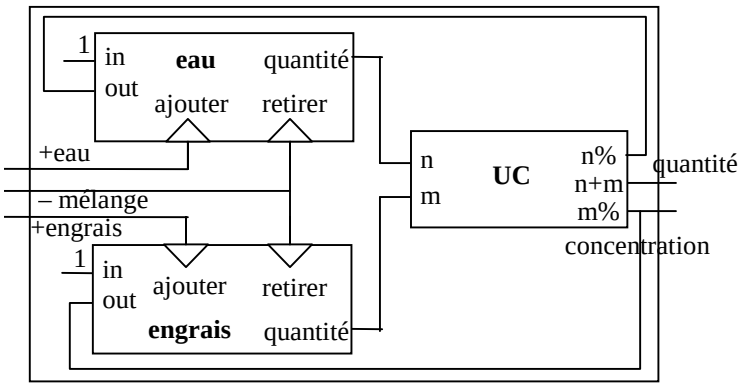

Figure 6 – Le composant mélangeur.

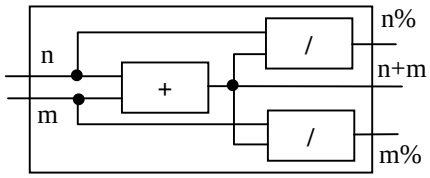

Figure 7 – L'unité de calcul du mélangeur.

2-3. Le composant mélangeur.

Le composant mélangeur modélise le récipient où l'engrais et l'eau sont mélangés. C'est un module SEP (cf. Figure 6). Ce module est constitué de deux récipients et d'une unité de calcul. Un récipient conserve la quantité d'eau présente dans le mélange, l'autre récipient conserve la quantité d'engrais. Notre loi de mélange est simple, cette loi est implémentée par l'unité de calcul, une loi plus proche de la réalité aurait pu être implémentée mais ce n'était pas l'objectif.

Loi de mélange : nous considérons que le volume de n litres d'engrais dilués dans m litres d'eau est n+m. La concentration d'engrais est alors n/(n+m). L'unité de calcul peut alors être réalisée par le module présenté par la Figure 7.

Le composant mélangeur a trois entrées de commande. L'entrée *+eau* permet d'ajouter de l'eau au mélange, l'entrée +engrais permet d'ajouter de l'engrais, l'entrée *–mélange* permet de vider du mélange. L'unité de calcul calcule la quantité de mélange ainsi que le taux d'engrais dans le mélange. De plus, elle permet de maintenir la même concentration lorsque la cuve est vidée. Pour une unité de mélange enlevée, il faut enlever *n/(n+m)* unité d'eau et *m/(n+m)* unité d'engrais pour conserver la même concentration.

2-4. Le composant Environnement.

C'est un composant qui modélise tout l'environnement 'physique' de l'arroseur, les récipients, les robinets, le mélangeur et les plantes. C'est un module Esterel composé des composants définis dans les sections précédentes. La Figure 8 présente ce module.

Lorsque le robinet d'eau est ouvert, un signal périodique est émis en sortie du composant WV (*Water Valve*), à chaque front montant de ce signal une quantité d'eau égale à 1 est retirée dans le récipient d'eau et ajoutée au mélangeur. La période du signal périodique représente le débit d'écoulement à travers le robinet d'eau.

Le robinet d'engrais FV (*Fertilizer Valve*) a un comportement similaire, mais son débit n'est pas nécessairement identique.

La pompe a aussi un comportement similaire, une unité de liquide (1 litre) est retirée du mélangeur et est ajoutée au récipient *plantes*. Le nombre de litres effectivement utilisé est calculé en utilisant le composant PC du processeur maîtrise, rappelons que celui-ci incrémente un compteur de 1 à chaque front montant d'un signal sur son port *inc*.

Des observateurs à la sortie du mélangeur comparent la quantité de mélange aux valeurs limites 10 litres et 90 litres afin de produire les signaux LOW_LEVEL et HIGH_LEVEL.

Le phénomène d'évaporation de l'eau et le fait que les plantes boivent de l'eau sont modélisés en utilisant un composant *Horloge* qui retire de l'eau dans le récipient *plantes* de façon périodique. La quantité d'eau présente dans le récipient *plantes* représente l'humidité des plantes.

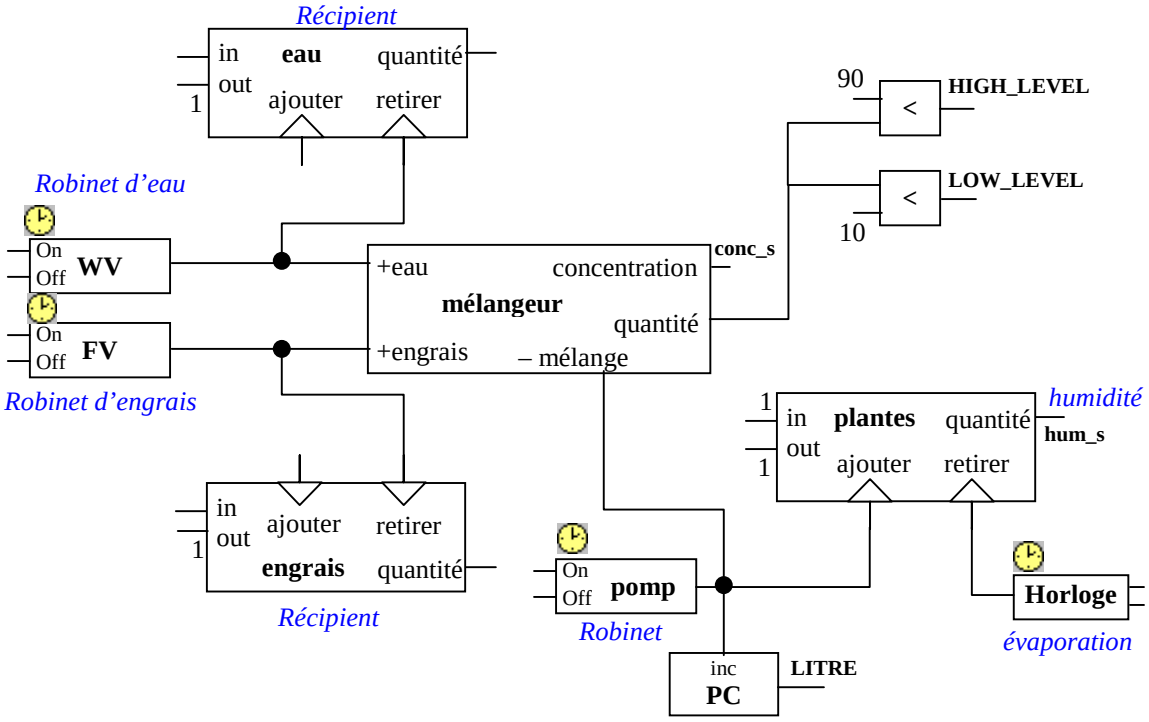

Figure 8 – Le composant environnement.

3- Modélisation des tâches asynchrones.

Le contrôleur synchrone utilise deux tâches asynchrones.

La première tâche nommée *PARAM* permet de lire dans une base de données la concentration et l'humidité de références des plantes à arroser. Le type de plantes à arroser est passé au contrôleur par le signal valué *START*. En fait, pour simplifier l'exemple, nous effectuons une opération arithmétique à partir du signal valué pour déterminer la concentration et l'humidité de référence. Toute autre opération serait autorisée.

La deuxième tâche nommée *REPORT* mémorise la consommation d'eau de l'arroseur automatique, cette consommation est mémorisée à chaque fois que la stabilité du système est atteinte, c'est-à-dire lorsque le signal *DONE* est émis.

Comme le montre le chapitre IX, les tâches asynchrones sont des composants SEP qui héritent de la classe *sep.sync.TaskStrl*. Ces composants sont des Thread Java, l'ordonnanceur SEP n'a donc aucun contrôle sur le rythme d'exécution de ces tâches. La méthode *run* constitue le comportement.

Le composant STRL instancie la classe dont le nom est *component.sync.Param* (respectivement *component.sync.Report*) et l'utilisera pour exécuter le comportement de la tâche *PARAM* (respectivement *REPORT*).

Voici le code Java écrit pour ces deux tâches :

```
public class Param extends modele.sync.TaskStrl
{
   private String X, Y;
   private Value XValue, YValue;
   public void run() {
      X = nextParam();  // Concentration
      Y = nextParam();  // Humidité
      NextParam();       // Start
      Int number = getIntParam();

      XValue = new IntValue(number+2);
      YValue = new IntValue(number*2+1);
      WaitTick(); // On attend un tick avant de finir
      Terminate();
   }

   public void _return() {
      super._return();
      com.setData(X, XValue);
      com.setData(Y, YValue);
   }
}
```

```
public class Report extends modele.sync.TaskStrl
{
   public void run() {
// type de plantes
      nextParam();
      int number = getIntParam();

// Consommation
      nextParam();
      int cons = getIntParam();

// Affichage du résultat
      PrintConsole("Situation "+number+
                        " ->Consommation : "+cons);
      terminate();
   }
}
```

La classe *TaskStrl* définit des méthodes qui permettent de lire le nom des paramètres (*nextParam*) et leur valeur (*getIntParam*). Ici il y a deux paramètres de sortie (X et Y) qui sont les paramètres d'humidité et de concentration, et un paramètre d'entrée (*number*) qui est la valeur du signal *START*.

Lorsque le comportement se termine (appel de *terminate*), la tâche attend le signal return pour rendre la valeur des paramètres de retour (appel de la méthode *_return*). La primitive com.setData permet d'affecter les paramètres de sortie. Ici on affecte la valeur *XValue* pour le paramètre X et la valeur *YValue* pour le paramètre Y.

La tâche *REPORT* lit l'information concernant le type des plantes arrosées, la consommation et se contente d'afficher l'information sur la console de contrôle de SEP.

En conclusion, ce chapitre illustre le mécanisme d'encapsulation de la machine d'exécution Esterel dans un composant SEP et montre comment il est possible de simuler l'environnement complet composé d'un module Esterel, de tâches et de composants asynchrones.
De plus, il montre que SEP peut être utilisé pour des environnements hétérogènes sensiblement différents des architectures matérielles pour lesquelles il a été conçu à l'origine, ce qui montre ses possibilités d'évolution. Notamment par l'introduction de composants *TimeComponent* et d'une machine d'exécution Esterel.