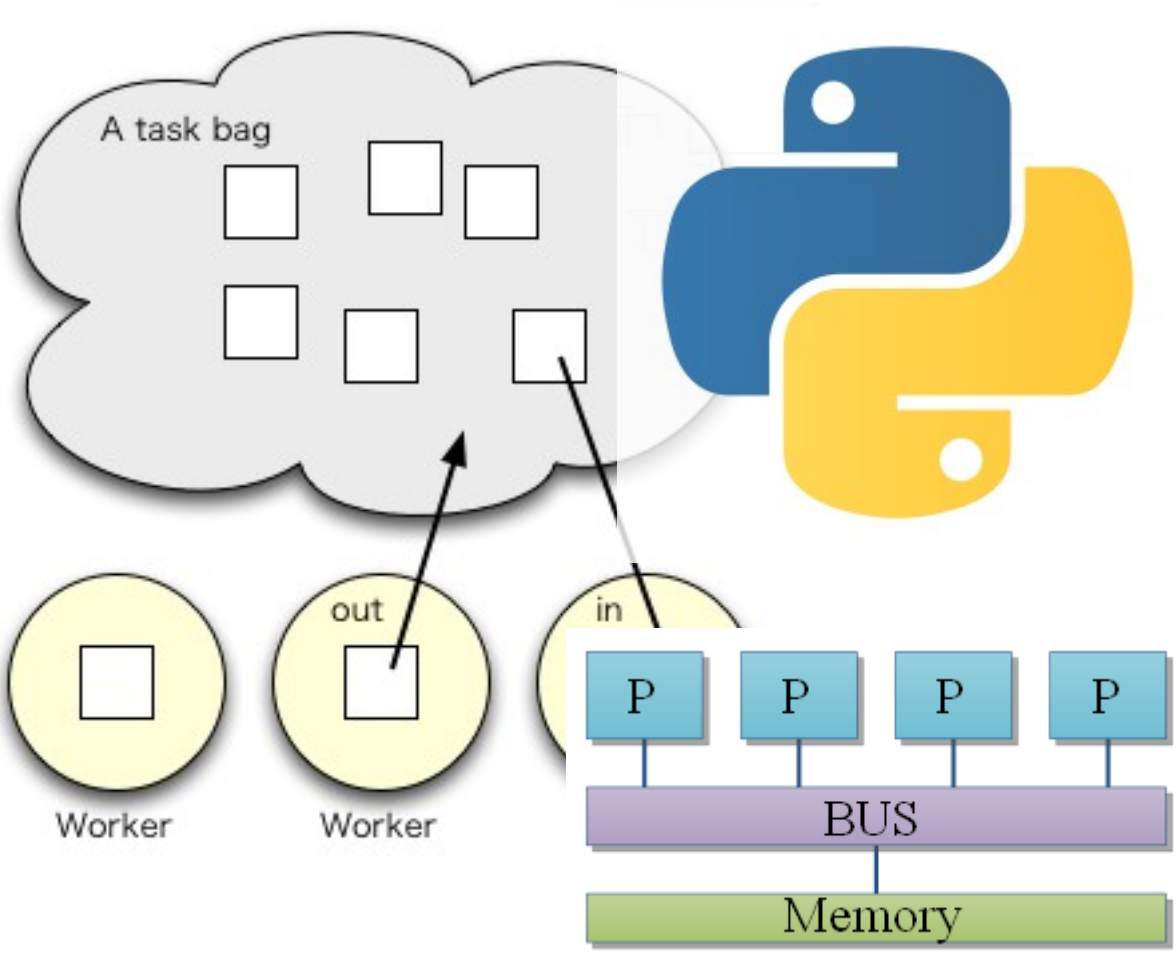
Matthew Hixon
Rémi Mollard
Céline de Roland

25/04/2015

TP INFO 805

ARCHITECTURES LOGICIELLES



Objectifs du TP :

- Comprendre le comportement d'un style architectural
- Analyser et manipuler des échanges de données entre plusieurs agents
- Comprendre les différents types du modèle Linda
- Adapter plusieurs problèmes au modèle Linda

TABLE DES MATIÈRES

I)Introduction, Rappel du Sujet....3
1)Introduction....3
2)Rappel du sujet....3
II)Problème de la mine vu en cours....4
1)Master....4
2)Capteur CH4....4
3)Capteur CO....5
4)Capteur H2O....5
5)Scrutateur Gaz Haut....5
6)Scrutateur Gaz Bas....5
7)Scrutateur H2O Haut....5
8)Scrutateur H2O Bas....5
9)Logique Gaz Bas....6
10)Logique Gaz Haut....6
11)Logique H2O Bas....6
12)Logique H2O Haut....6
13)Pompe....7
14)Ventilateur....7
III)Problème de la mine revisité....7
1) Master....8
2)Capteur_CO, Capteur_CH4, Capteur_H2O....8
3)Horloge....8
4)Capteur_Personnes....8
5)Scrutateurs....9
6)Scrutateur Heure Creuse....9
7)Scrutateur Heure Pleine....10
8)Logiques Gaz et H2O....10
9)Logique Heure Creuse....10
10)Logique Heure Pleine....10
11)Travailleur....10
12)Ascenseur....11
13)Pompe, Ventilateur, Lampe....11
IV)Problème du train (Examen 2013/2014)....12
1) Trains....12
2)Capteur demandes entrées....13
3)Capteur demandes sorties....13
4)Capteur d'acquittement entrées....13
5)Capteur d'acquittement sorties....14
6)Opérateur....14
V)Conclusion....15

I) INTRODUCTION, RAPPEL DU SUJET

1) Introduction

L'architecture logicielle décrit d'une manière symbolique et schématique les différents éléments d'un ou de plusieurs systèmes informatiques, leurs interrelations et leurs interactions.

De manière plus globale une architecture logicielle peut être vue comme :

→ Un ensemble de briques de base : composants

→ Un ensemble de règles d'utilisation.

→ Un ensemble de recettes pour combiner les interactions entre les différents composants.

De nos jours il existe un nombre considérable de styles architecturaux. Ce sont des modèles de référence de résolution des problèmes courants d'architecture. Un système informatique peut utiliser plusieurs styles selon sa complexité.

Dans ce TP nous allons aborder le style architécturale de type : « données partagées », grâce au modèle Linda.

Linda n'est pas un langage de programmation mais uniquement un langage de coordination qui définit un modèle de communication et de synchronisation.

Il se défini de la façon suivante :

- Une collection de tuple, appelé espace des tuples. Un tuple est une suite finie et ordonnée de champs typés.

- Un ensemble d'opérations : Rajout, retrait et lectures de tuples.

- Un mécanisme d'unification qui permet d'accéder aux tuples.

2) Rappel du sujet

Deux sujets sont proposés :

- Le premier sujet est d'implémenter le modèles Linda ainsi que le problème de la mine vue en cour.

- Le deuxième sujet est d'implémenter la mine vue en cours et de trouver un autre problème à implémenter.

Nous avons pris la décision de prendre le deuxième sujet pour la simple raison que nous avons trouvé une implémentation du modèle Linda en Python. Cette implémentation est regroupée dans une bibliothèque qui se nomme : PyLinda. Son point fort est sa simplicité d'utilisation.

Il est possible d'obtenir une version (Projet abandonnée) à l'adresse suivante :

http://freecode.com/projects/pylinda.

Nous avons utilisé cette librairie afin de réaliser trois travaux :

- Implémenter le problème de la mine vu en cours
- Implémenter une mine plus complexe, en ajoutant de nouvelles spécifications
- Implémenter le problème du train de l'examen 2013/2014

II) PROBLÈME DE LA MINE VU EN COURS

Le graphique ci-dessous montre les événements qui font passer le système d'un état à un autre.

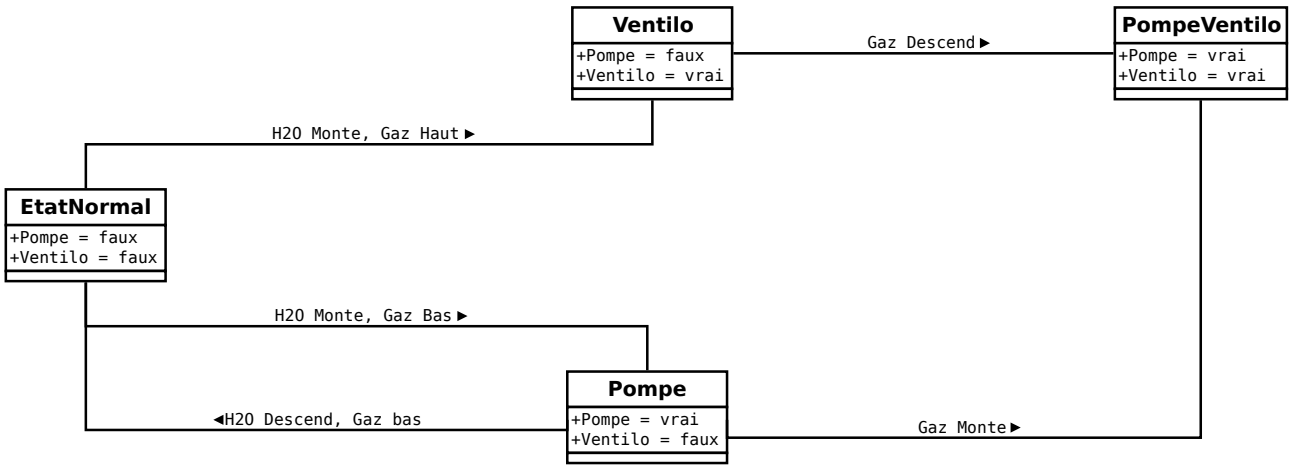
L'état EtatNormal correspond à Ventilateur et Pompe inactifs.

L'état Ventilo correspond à Ventilateur actif, Pompe inactive.

L'état Pompe correspond à Ventilateur inactif, Pompe active.

L'état PompeVentilo correspond à Pompe et Ventilateur actifs.

Les événements Monte et Descend correspondent à passer respectivement au dessus et en dessous des seuils fixés.



1) Master

Cet agent initialise le système.

2) Capteur CH4

Cet agent mesure à intervalles réguliers le niveau de méthane dans la mine.

```
Capteur_CH4 =
      // On regarde l'état de la pompe uniquement dans un but de simulation.
      // En effet, si la pompe est en route, le niveau de gaz augmente plus vite
      // Si le ventilateur est en route, le niveau de gaz diminue
      READ (<| "etat_pompe" string ;; ?pompe_en_route string |>)
      . READ (<| "etat_ventilateur" ;; ?ventilateur_en_route string |>)
      . établir le nouveau niveau de CH4 (variation aléatoire du niveau)
      . IN (<| "Niveau_CH4" string ;; ?_ float |>)
      . OUT (< "Niveau_CH4" string ;; !nouveau_niveau float >)
      . Capteur_CH4
```

3) Capteur CO

Cet agent mesure à intervalles réguliers le niveau de monoxyde de carbone dans la mine.

Même fonctionnement que Capteur_CH4.

4) Capteur H2O

Cet agent mesure à intervalles réguliers le niveau d'eau dans la mine.

Même fonctionnement que Capteur_CH4.

5) Scrutateur Gaz Haut

Lorsqu'il est actif, cet agent détecte si les niveaux de gaz dépassent le seuil fixé.

```
Scrutateur_Gaz_Haut =
      // Le tuple detection_gaz_haut est présent si la surveillance de la montée
du gaz est activée
      READ (< "detection_gaz_haut" string >)
      // On lit les niveaux et on prévient les autres agents si ils dépassent
les seuils fixés
      . READ (<| "Niveau_CH4" string ;; ?niveau_CH4 float |>)
      . READ (<| "Niveau_CO" string ;; ?niveau_CO float |>)
      . (
            [niveaux de gaz > seuils fixés]
                  IN (< "detection_gaz_haut" string >)
                  . OUT (< "Gaz_haut_detecté" string >)
      )
      . Scrutateur_Gaz_Haut
```

6) Scrutateur Gaz Bas

Lorsqu'il est actif, cet agent détecte si les niveaux de gaz descendent en dessous du seuil fixé.

Même fonctionnement que scrutateur gaz haut.

7) Scrutateur H2O Haut

Lorsqu'il est actif, cet agent détecte si le niveau d'eau dépasse le seuil fixé.

Même fonctionnement que scrutateur gaz haut.

8) Scrutateur H2O Bas

Lorsqu'il est actif, cet agent détecte si le niveau d'eau descend en dessous du seuil fixé.

Même fonctionnement que scrutateur gaz haut.

9) Logique Gaz Bas

Lorsqu'il reçoit un signal du scrutateur Gaz Bas, cet agent décide quoi faire.

```
Logique_Gaz_Bas =
      IN (<| "Gaz_Bas_Détecté" string |>)
      // Le gaz est revenu à un niveau bas, on peut lancer la pompe
      // comme on lance la pompe, on doit aussi détecter quand le niveau d'eau
      // redescendra en dessous du seuil fixé
      . OUT (< "Pompe_En_Route" string >)
      . OUT (< "detection_H2O_bas" string >)
      . Logique_Gaz_Bas
```

10) Logique Gaz Haut

Lorsqu'il reçoit un signal du scrutateur Gaz Haut, cet agent décide quoi faire.

```
Logique_Gaz_Haut =
      IN (<| "Gaz_Haut_Détecté" string |>)
      // Si on est dans cette situation, c'est que la pompe est en route,
      // on doit activer le ventilateur jusqu'à arrêt de la pompe
      . OUT (< "Ventilateur_En_Route" string >)
      . Logique_Gaz_Haut
```

11) Logique H2O Bas

Lorsqu'il reçoit un signal du scrutateur H2O Bas, cet agent décide quoi faire.

```
Logique_H2O_Bas =
      IN (<| "H2O_Bas_Détecté" string |>)
      // L'eau est revenue en dessous du seuil fixé
      // Si on a eu à activer le ventilo pendant le pompage, il faut le
désactiver
      // Sinon il faut arrêter de surveiller la montée du gaz
      . READ (<| "etat_ventilateur" ;; ?ventilateur_en_route string |>
      . (
            [ventilateur_en_route = "activé"]
                  OUT (< "Ventilateur_Arret" string >)
            + [else]
                  IN (<| "detection_gaz_haut" string |>)
      )
      // Dans tous les cas on arête la pompe et on recommence à surveiller la
montée de l'eau
      . OUT (< "Pompe_Arret" string >)
      . OUT (< "detection_H2O_haut" string >)
      . Logique_H2O_Bas
```

12) Logique H2O Haut

Lorsqu'il reçoit un signal du scrutateur H2O Haut, cet agent décide quoi faire.

```
Logique_H2O_Haut =
      IN (<| "H2O_Haut_Détecté" string |>)
      // On commence par regarder les niveaux de gaz afin de décider si on doit
d'abord activer le ventilateur avant de pomper l'eau
      . READ (<| "Niveau_CH4" string ;; ?niveau_CH4 float |>)
      . READ (<| "Niveau_CO" string ;; ?niveau_CO float |>)
      . (
            [niveaus de gaz < seuils fixés]
```

```
                    OUT (< "Pompe_En_Route" string >)
                    . OUT (< "detection_H2O_bas" string >)
                    . OUT (< "detection_gaz_haut" string >)
              + [else]
                    OUT (< "Ventilateur_En_Route" string >)
                    . OUT (< "detection_gaz_bas" string >)
      )
      . Logique_H2O_Haut
```

13) Pompe

Cet agent s'active ou se désactive, selon les ordres reçus.

```
Pompe =
      READ (<| "etat_pompe" string ;; ?etat_pompe string |>
      // Si l'état actuel est actif, on attend un ordre de désactivation
      // et réciproquement si l'état est inactif
      . (
            [etat_pompe = "activé"]
                  IN (<| "Pompe_En_Route" string |>)
                  . etat_pompe = "désactivé"
            + [else]
                  IN (<| "Pompe_Arret" string |>)
                  . etat_pompe = "activé"
      )
      . IN (<| "etat_pompe" string ;; ?etat_pompe string |>
      . OUT (< "etat_pompe" string ;; !etat_pompe string |>
      . Pompe
```

14) Ventilateur

Cet agent s'active ou se désactive, selon les ordres reçus.

Même fonctionnement que la pompe.

III) PROBLÈME DE LA MINE REVISITÉ

Afin d'expérimenter l'intérêt de cette technologie dans une situation complexe, nous avons choisi d'ajouter des spécifications nouvelles au problème de la mine plutôt que de recommencer à zéro un nouveau problème.

Nous avons donc ajouté les fonctionnalités suivantes :

- Des mineurs travaillent dans la mine, répartis sur deux équipes.
- La première équipe travaille de 5h à 13h, et la seconde de 15h à 23h.
- Pendant les heures de travail (heures pleines), les seuils sont plus restrictifs qu'en dehors des heures de travail (heures creuses).
- Un ascenseur permet aux travailleurs de descendre dans la mine au début de leur service, et remonter à la fin. Il ne peut prendre qu'une seule personne à la fois.
- Une lampe est présente dans la mine, allumée pendant les heures pleines, éteinte pendant les heures creuses.

Au départ, nous avions prévu un ensemble de fonctionnalités bien plus important :

- Les travailleurs pouvaient, durant leur service, sortir un quart d'heure environ pour faire une pause.
- Nous avions fait une distinction entre l'ingénieur et les mineurs. Les mineurs pouvaient fumer une cigarette dans la mine, ce qui déclenchait le ventilateur. L'ingénieur était alors prévenu et tentait de faire arêter le mineur, s'il n'y parvenait pas le mineur s'arêtait tout seul au bout de 5 minutes.
- L'ingénieur pouvait également arêter ou activer manuellement la pompe et le ventilateur.
- L'ascenseur pouvait prendre 3 personnes à la fois.
- Durant les heures pleines, le forage de la mine pouvait ouvrir une poche de gaz ou une poche d'eau. Cela provoquait une montée très importante des niveaux concernés, qui passaient alors au dessus d'un seuil appelé seuil critique. Cela déclenchait une alarme et tous les travailleurs devaient sortir jusqu'à retour à la normale.
- Pendant ces périodes de catastrophe, l'ascenseur ne pouvait que monter des personnes, et interdisait donc la descente dans la mine.

Nous n'avons pas réussi à mettre en place ces fonctionnalités supplémentaires, mais essayer de le faire nous a permis d'expérimenter par nous même les tenants et les aboutissants du développement d'un système complexe.

Par exemple, nous nous sommes confrontés aux problèmes de modélisation et d'étude des cas. Le petit diagramme d'états que nous avons présenté à la partie précédente ne nous a pas semblé réalisable en amont et devrait plutôt être écrit en aval par une machine. La modélisation du système nécessite donc d'être réalisée différemment.

Dans une situation simple, nous avons pu commencer par spécifier le comportement du système, puis en déduire les agents. Dans une situation complexe, il nous a semblé plus réaliste de commencer par spécifier les agents et en déduire le système.

Dans la situation intermédiaire que nous avons implémentée, nous avons fait un mélangé les deux approches.

1) Master

Cet agent configure et initialise le système (nous avons utilisé les tuples pour partager les informations de configuration du système) .

2) Capteur_CO, Capteur_CH4, Capteur_H2O

Ces agents n'ont pas changé depuis la version précédente.

3) Horloge

L'horloge peut être vue comme un capteur (elle capte l'heure qu'il est). Pour notre simulation, l'heure qu'il est est un tuple dans la base de données, modifié à intervalles réguliers par l'horloge. Pour faciliter la simulation, nous configurons une variable duree_d_une_heure dans master, et l'horloge avance d'une heure toutes les duree_d_une_heure.

4) Capteur_Personnes

Cet agent reçoit des informations de la part de l'ascenseur lorsque des travailleurs entrent dans la

mine ou en sortent. Cela permet de savoir à tout moment combien de personnes se trouvent dans la mine.

En l'état, il n'est associé à aucun scrutateur ni aucune logique, mais nous l'avons laissé pour un éventuel usage ultérieur (par exemple, lorsque l'alarme retenti, il faut s'assurer que tous les travailleurs ont quitté la mine avant de faire certaines actions).

```
Capteur_Personnes(nbPersonnes) =
      // Si quelqu'un monte, on incrémente le nombre de personnes
      // Si quelqu'un descend, on le décrémente
      IN (<| "Entree/Sortie" string ;; ?action string |>)
      . (
            [action = "monter"]
                  nbPersonnes++
            + [else]
                  nbPersonnes--
      )
      . Capteur_Personnes(nbPersonnes)
```

5) Scrutateurs

A la version précédente, on connaissait les seuils dès le début. Maintenant, on doit les relire à chaque tour puisqu'ils changent selon l'horaire.

```
Scrutateur_Gaz_Haut =
      // Le tuple detection_gaz_haut est présent si la surveillance de la montée
du gaz est activée
      READ (< "detection_gaz_haut" string >)
      // On a ajouté la lecture des seuils
      . READ (<| "Seuil_CH4" string ;; ?seuil_CH4 float |>)
      . READ (<| "Seuil_CO" string ;; ?seuil_CO float |>)
      // On lit les niveaux et on prévient les autres agents si ils dépassent
les seuils lus
      . READ (<| "Niveau_CH4" string ;; ?niveau_CH4 float |>)
      . READ (<| "Niveau_CO" string ;; ?niveau_CO float |>)
      . (
            [niveaux de gaz > seuils lus]
                  IN (< "detection_gaz_haut" string >)
                  . OUT (< "Gaz_haut_detecté" string >)
      )
      . Scrutateur_Gaz_Haut
```

6) Scrutateur Heure Creuse

Comme les précédents scrutateurs, cet agent, lorsqu'il est actif, surveille l'heure qu'il est et envoie une alerte (un tuple) lorsqu'on dépasse 13h et 23h.

```
Scrutateur_Heure_Creuse =
      // Le tuple detection_heure_creuse est présent si la surveillance du
passage en heure creuse est activée
      READ (<| "detection_heure_creuse" string |>)
      // On lit l'heure qu'il est et on prévient les autres agents si on dépasse
13h ou 23h
      . READ (<| "heure" string;; ?heure int |>)
      . (
            [heure > 13 ou heure > 23]
                  IN (< "detection_heure_creuse" string >)
                  . OUT (< "Heure_creuse_detecté" string >)
      )
      . Scrutateur_Heure_Creuse
```

7) Scrutateur Heure Pleine

Même fonctionnement que Scrutateur Heure Creuse.

8) Logiques Gaz et H2O

Ces agents n'ont pas changé depuis la version précédente

9) Logique Heure Creuse

Lorsqu'on passe en heure creuse, il faut modifier les seuils, indiquer aux travailleurs qu'ils doivent sortir, et éteindre les lumières.

```
Logique_Heure_Creuse =
      IN (<| "Heure_creuse_detecté" string |>)
      . IN (< "Seuil_H2O_Haut" string ;; ?_ float >)
      . OUT (< "Seuil_H2O_Haut" string ;; !seuil_H2O_haut_heure_creuse float >)
      . OUT (< "Lampe_Arret" string >)
      // Si il est 13h, c'est l'équipe 1 qui sort.
      // Si il est 23h c'est l'équipe 2 qui sort.
      . READ (<| "heure" string;; ?heure int |>)
      . (
            [heure = 13]
                  numero_equipe = 1
            + [heure = 23]
                  numero_equipe = 2
      )
      // Il y a 4 travailleurs dans une équipe, donc on envoie 4 informations à
consommer.
      . OUT (< "Equipe_Sortir" string ;; !numero_equipe int>)
      . OUT (< "Equipe_Sortir" string ;; !numero_equipe int>)
      . OUT (< "Equipe_Sortir" string ;; !numero_equipe int>)
      . OUT (< "Equipe_Sortir" string ;; !numero_equipe int>)
      // On est en heure creuse, on doit donc maintenant surveiller le passage
en heure pleine
      . OUT (< "detection_heure_pleine" string >)
      . Logique_Heure_Creuse
```

10) Logique Heure Pleine

Même fonctionnement que logique heure creuse.

11) Travailleur

Nous avons 8 travailleurs (4 par équipe), qui fonctionnent tous exactement de la même manière. Nous donnons ici l'exemple d'une ingénieure nommée Isabelle.

```
Isabelle(etat_action, etat_location) =
      (
      [etat_action = "inactif", etat_location = "dehors"]
            // Attendre le signal indiquant d'aller travailler
            IN (<| "Equipe_Entrer" string ;; 1 int |>)
            // Appeler l'ascenseur
            . OUT (< "appel_ascenseur" string ;; "descendre" string ;;
"Isabelle" string >)
            // Lorsque l'ascenseur est arrivé à destination, le travail
commence
            . IN (<| "ascenseur_arrivé" string ;; "Isabelle" string |>)
            . etat_location = "mine"
            . etat_action = "travail"
      + [etat_action = "travail"]
            // Attendre le signal indiquant la fin du travail
```

```
            IN (<| "Equipe_Sortir" string ;; 1 int |>)
            // Appeler l'ascenseur
            . OUT (< "appel_ascenseur" string ;; "monter" string ;; "Isabelle"
string >)
            // Lorsque l'ascenseur est arrivé à destination, le travail est
terminé
            . IN (< "ascenseur_arrivé" string ;; "Isabelle" string >
            . etat_location = "dehors"
            . etat_action = "inactif"
      )
      . Isabelle(etat_action, etat_location)
```

12) Ascenseur

```
Ascenseur(etat_position) =
      IN (<| "appel_ascenseur" string ;; ?demande string ;; ?nom string |>)
      //Lorsqu'on a reçu une demande d'un travailleur, on doit d'abord aller
jusqu'à son niveau (ex : si la personne est en haut et l'ascenseur est en bas,
il faut d'abord monter l'ascenseur jusqu'à la personne)
      . (
            [ (etat_position = "haut" et demande = "monter") OU
              (etat_position = "bas" et demande = "descendre") ]
                  //Dans notre simulation, monter ou descendre = sleep(1min)
                  rejoindre le demandeur
      )
      //Dans notre simulation, monter ou descendre une personne = sleep(1min)
      . déplacer le demandeur
      //On prévient la personne qu'elle est arrivée
      . OUT (< "ascenseur_arrivé" string ;; !nom string >)
      //On met à jour la position de l'ascenseur
      . (
            [demande = "monter"]
                  etat_position = "haut"
            + [demande = "descendre"]
                  etat_position = "bas"
      )
      //On prévient le capteur de personnes
      . OUT (< "Entree/Sortie" string ;; !demande string >)
      . Ascenseur(etat_position)
```

13) Pompe, Ventilateur, Lampe

La pompe et le ventilateur n'ont pas changé, la lampe fonctionne sur le même modèle.

IV) PROBLÈME DU TRAIN (EXAMEN 2013/2014)

Le problème du train repose sur la gestion d'un aiguillage à l'entrée d'une gare. Le principe est le suivant :

→ Nous avons un nombre n de places pour stocker les trains dans la gare.

→ Nous avons un nombre m de voies d'entrées dans la gare.

Chacune de ces n et m voies d'accès converge vers une unique voie. Ceci induit les aspects suivants :

→ Un seul train peut circuler sur cette unique voie.

→ Si il y'a de place pour stocker un train dans la gare, le train voulant entrée en gare est prioritaire.

→ Si il y pas de place dans la gare les trains voulant entrées sont bloqué en attente de libération d'un place dans la gare.

Afin de répondre au mieux au sujets proposé nous avons choisit d'implémenter les agents suivants :

1) Trains

Cet agent simule une entrée en gare ou une sortie de la gare. (A l'initialisation du systéme les trains sont tous à l'extérieur de la gare)

```
Train(etat_position) =
      //On vérifie la localisation du train dans le systéme
      [etat_position = "dehors"]
            //Le train est dehors, on dépose un tuple de demande entrée
            OUT(< "demande_entree" string |>)
            //On attend que l'opérateur accepte que l'on rentre
            . IN(<| "accord_entree" string |>)
            //On met à jour la variable de position
            etat_position = "dedans"
            //On acquitte notre entrée auprès de l'opérateur
            . OUT(< "je_suis_entre" string >)
            //On se rappel récursivement
            . Train(etat_position)
      //On effectue la même opération dans le cas d'une sortie du train
      + [position = "dedans"]
            OUT(< "demande_sortie" string >)
            . IN(<| "accord_sortie" string |>)
            etat_position = "dehors"
            . OUT(< "je_suis_sorti" string >)
            . Train(etat_position)
```

2) Capteur demandes entrées

Cet agent sert de compteur , il permet à l'opérateur de connaître le nombre de train en attente d'entrée.

```
c_demandes_entrees() =
  //On attend une demande d'entrée d'un train
  IN(<| "demande_entree" string |>)
  //On récupère le nombre de demande d'entrée en attente
  . IN(<| "nombre_demandes_entrees" string ;; ?nombre_demandes_entrees int |>)
  //On incrémente la valeur récupérée
  nombre_demandes_entrees =  nombre_demandes_entrees + 1
  //On remet la valeur incrémentée dans l'espace de tuple
  . OUT(< "nombre_demandes_entrees" string ;; ?nombre_demandes_entrees int >)
  //On se rappel récursivement
  . c_demandes_entrees()
```

3) Capteur demandes sorties

Cet agent sert de compteur , il permet à l'opérateur de connaître le nombre de train en attente de sortie..

```
c_demandes_sorties() =
  //On attend une demande de sortie d'un train
  IN(<| "demande_sortie" string |>)
  //On récupère le nombre de demande de sortie en attente
  . IN(<| "nombre_demandes_sorties" string ;; ?nombre_demandes_sorties int |>)
  //On incrémente la valeur récupérée
  nombre_demandes_sorties =  nombre_demandes_sorties + 1
  //On remet la valeur incrémentée dans l'espace de tuple
  . OUT(< "nombre_demandes_sorties" string ;; ?nombre_demandes_sorties int >)
  //On se rappel récursivement
  . c_demandes_sorties()
```

4) Capteur d'acquittement entrées

Cet agent sert de compteur, il permet décrémenter le nombre d'entrées en attente lorsqu'un train est entrée de la gare.

```
c_acquittement_entrees() =
  //On attend un acquittement d'entrée d'un train
  IN(<| "entree_acquittement" string |>)
  //On récupère le nombre de demande d'entrées en attente
  . IN(<| "nombre_demandes_entrees" string ;; ?nombre_demandes_entrees int |>)
  //On décrémente la valeur récupérée
  nombre_demandes_entrees =  nombre_demandes_entrees - 1
  //On remet la valeur décrémentée dans l'espace de tuple
  . OUT(< "nombre_demandes_entrees" string ;; ?nombre_demandes_entrees int >)
  //On se rappel récursivement
  . c_acquittement_entrees()
```

5) Capteur d'acquittement sorties

Cet agent sert de compteur, il permet décrémenter le nombre de sortie en attente lorsqu'un train est sorti de la gare.

```
c_acquittement_sorties() =
  //On attend un acquittement de sortie d'un train
  IN(<| "sortie_acquittement" string |>)
  //On récupère le nombre de demande de sorties en attente
  . IN(<| "nombre_demandes_sorties" string ;; ?nombre_demandes_sorties int |>)
  //On décrémente la valeur récupérée
  nombre_demandes_sorties =  nombre_demandes_sorties - 1
  //On remet la valeur décrémentée dans l'espace de tuple
  . OUT(< "nombre_demandes_sorties" string ;; ?nombre_demandes_ sorties int >)
  //On se rappel récursivement
  . c_acquittement_sorties()
```

6) Opérateur

Cet agent sert d'aiguilleur pour la gestion des trains. Il s'occupe d'autoriser/refuser l'entrée ou la sortie d'un train en fonction des places disponibles.

```
operateur(nbPlacesLibres) =
  //On récupère le nombre de demandes de sorties
  READ (<| "nombre_demandes_sorties" string ;; ?nombre_demandes_sorties int|>)
  //On récupère le nombre de demandes d'entrées
  . READ (<| "nombre_demandes_entrees" string;; ?nombre_demandes_ entrees int|>)
  //On test les valeurs récupérées pour traiter tous les cas possibles
  //Si aucune demande d'entrée et au moins une demande de sortie
  [nb_demandes_sorties > 0 et nb_demandes_entrees == 0]
     //On accorde la sortie pour le train
     OUT(< "accord_sortie" string >)
     //On attend que le train soit sorti de la gare
     . IN(<| "je_suis_sorti" string |>)
     //On acquitte la sortie du train
     . OUT(< "sortie_acquittement" string >)
     //On augmente le nombre de places libres dans la gare
     nbPlacesLibres =  nbPlacesLibres + 1
     //On se rappel récursivement
     . operateur(nbPlacesLibres)
  //Si au moins une demande d'entrée et au moins une place libre
  + [nb_demandes_entrees > 0 et nbPlacesLibres > 0]
     //On accorde l'entrée pour le train
     OUT(< "accord_entree" string >)
     //On attend que le train soit entré dans la gare
     . IN(<| "je_suis_entre" string |>)
     //On acquitte l'entrée du train
     . OUT(< "entree_acquittement" string >)
     //On diminue le nombre de places libres dans la gare
     nbPlacesLibres =  nbPlacesLibres – 1
     //On se rappel récursivement
     . operateur(nbPlacesLibres)
  //Si aucunes place et au moins une demande de sortie
  + [nb_demandes_sorties > 0 and nbPlacesLibres == 0]
     //On accorde la sortie pour le train
     OUT(< "accord_sortie" string >)
     //On attend que le train soit sorti de la gare
```

```
. IN(<| "je_suis_sorti" string |>)
//On acquitte la sortie du train
. OUT(< "sortie_acquittement" string >)
//On augmente le nombre de places libres dans la gare
nbPlacesLibres =  nbPlacesLibres + 1
//On se rappel récursivement
. operateur(nbPlacesLibres)
```

V) CONCLUSION

Tout d'abord nous avons apprécié le choix d'implémenter le systéme de la mine vu en TD. La première approche travaillé en groupe nous à permis de bien situer les choses et de mieux comprendre le fonctionnement de l'espace de tuples et des agents.

Nous avons cependant rencontré des problèmes lors de l'implémentation de la mine en plus complexe. La difficulté s'est fait ressentir lors de l'ajout de plusieurs nouveaux agents dans notre système. En effet le nombre de situations à prendre en compte à considérablement augmentées. Nous avons tout de même réussit à implémenter certains de ces nouveaux agents, tels que l'horloge présenté ci-dessus.

Pour finir, nous tenions également à vous remercier pour votre disponibilité à nous accorder de concevoir un autre système Linda afin d'avoir deux systéme entièrement implémenté pour le rendu du TP. Le problème du train à consolidé nos connaissances de Linda.