# modelisad

## Plateforme RECORD, modélisation de la décision

Ronan Trépos – Patrick Chabrier

UBIA *(Unité de Biométrie et Intelligence Artificielle)*

# Plan

# Plan

# Modélisation des systèmes de culture

- Modèle de décision, modèle biophysique

# Extensions pour la modélisation

Equations aux différences

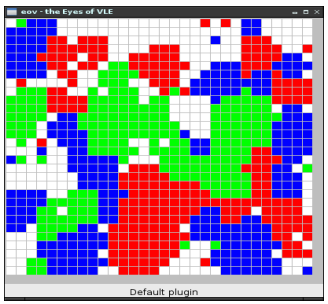
$$X_t = f(X_{t-1}, Y_{t-1})$$
$$Y_t = g(X_t, Y_{t-1})$$

Equations différentielles

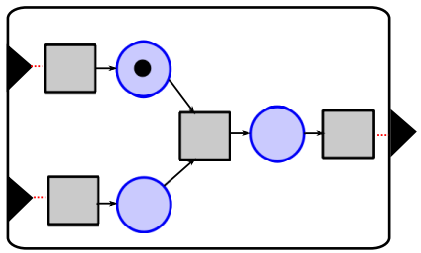
$$\frac{dx}{dt} = ax + y$$
$$\frac{dy}{dt} = by$$

DEVS

Automates cellulaires



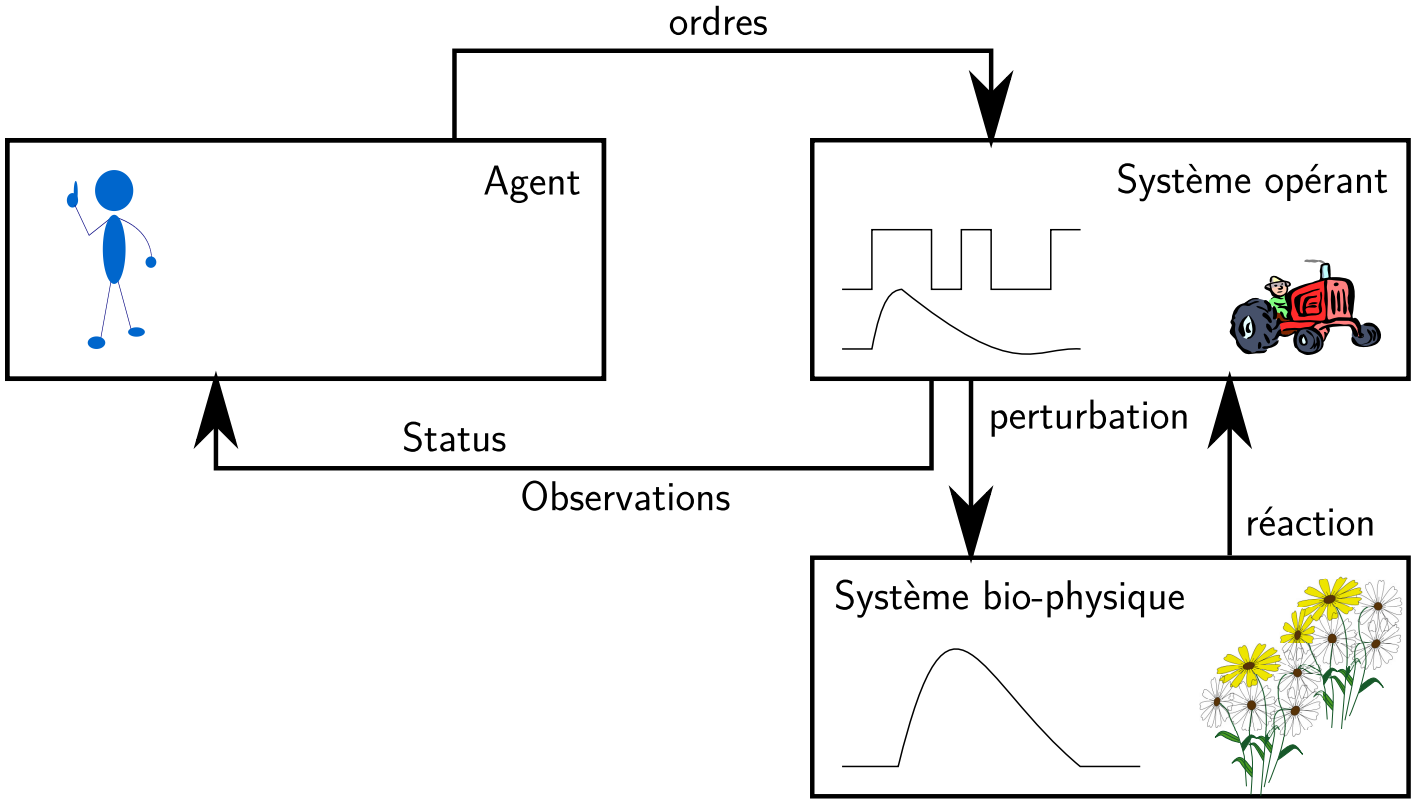

Réseaux de Pétri



```
activity {
 id=semis
}
activity {
 id=recolte
}
précédence {
 first=semis
 second=recolte
}
```

# Plan

## Modélisation : Modèle décision / Système opérant / Système bio-physique



## Le système Agent est composé :

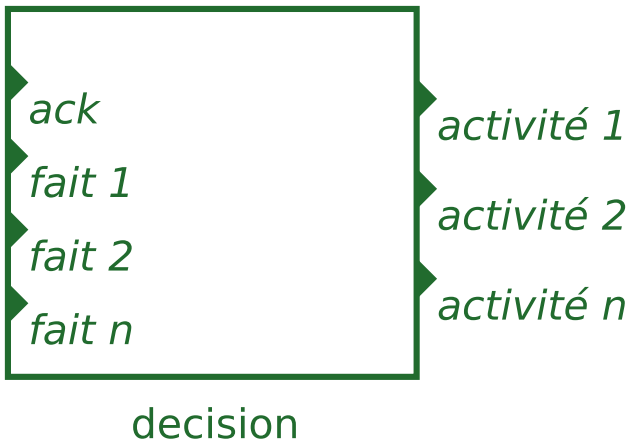
- d'une base de connaissances sous forme d'ensemble de variables, de structures, de classes :
  - Elle se représente un modèle des systèmes opérants et bio-physiques
  - Elle est mise à jour :
    - à partir des observations issues du système opérant
    - sur sa propre autonomie
- d'un graphe d'activités qui :
  - s'appuie sur la base de connaissances pour activer ou invalider des opérations techniques (ordre à envoyer au système opérant)
  - peut déclencher des opérations en parallèle
  - peut dynamiquement manipuler le graphe d'activités

## Le système Agent est composé :

- d'une base de connaissances sous forme d'ensemble de variables, de structures, de classes :
  - Elle se représente un modèle des systèmes opérants et bio-physiques
  - Elle est mise à jour :
    - à partir des observations issues du système opérant
    - sur sa propre autonomie
- d'un graphe d'activités qui :
  - s'appuie sur la base de connaissances pour activer ou invalider des opérations techniques (ordre à envoyer au système opérant)
  - peut déclencher des opérations en parallèle
  - peut dynamiquement manipuler le graphe d'activités

## Interface du système Agent

- La base de connaissances est mise à jour par les observations du système sur les ports d'entrée (faits *i*)
- Les sorties sont connectées au système opérant indiquant les déclenchements et arrêts des activités (activité *i*)
- Un port d'entrée ack pour la réception des événements de fins d'activité du système opérant



# Activité

## Caractéristiques

### Une activité est caractérisée par :

- un nom
- un état (*Wait*, *Started*, *FF*, *Done*, *Failed*)
- des contraintes temporelles
- des pré-conditions
- une fonction de changement d'état (wait à start, start à done, etc.) (reliée au port d'entrée ack) (*optionnelle*).
- une fonction de sortie (pour générer des événements complexes avec paramètres par ex.) (*optionnelle*).

Défini une plage pendant laquelle une activité peut démarrer :

## Contraintes temporelles simples

$$[-\infty, +\infty]$$
$$[\text{minstart}, +\infty]$$
$$[-\infty, \text{maxfinish}]$$
$$[\text{minstart}, \text{maxfinish}]$$

## Contraintes temporelles par intervalles

$$[-\infty \text{ ou minstart}, [\text{minfinish}, \text{maxfinish}]]$$
$$[[\text{minstart}, \text{maxfinish}], +\infty \text{ ou date}]$$
$$[[\text{minstart}, \text{maxstart}], [\text{minfinish}, \text{maxfinish}]]$$

# Activité

## Contraintes temporelles

Défini une plage pendant laquelle une activité peut démarrer :

### Contraintes temporelles simples

$$[-\infty, +\infty]$$
$$[\text{minstart}, +\infty]$$
$$[-\infty, \text{maxfinish}]$$
$$[\text{minstart}, \text{maxfinish}]$$

### Contraintes temporelles par intervalles

$$[-\infty \text{ ou minstart}, [\text{minfinish}, \text{maxfinish}]]$$
$$[[\text{minstart}, \text{maxfinish}], +\infty \text{ ou date}]$$
$$[[\text{minstart}, \text{maxstart}], [\text{minfinish}, \text{maxfinish}]]$$

# Activité

## Pré-conditions

### Un ensemble de prédicats assemblé sous forme de règles :

- un prédicat se traduit par un test effectué sur la base de connaissance du modèle.
- une règle est une conjonction de prédicats : pour qu'une règle soit valide, tous les prédicats doivent être valides.
- une activité est attachée à 0 ou *n* règles pour former les pré-conditions.
- une activité valide ses pré-conditions si au moins une des règles est valide.

# Activité

## Contraintes de précédence

Relier les activités entre-elle : les contraintes FS, SS et FF.

### Les contraintes sont valides si...

$F_iS_j$ l'activité $j$ démarre après la fin de l'activité $i$.

$S_iS_j$ l'activité $j$ démarre après le démarrage de l'activité $i$.

$F_iF_j$ l'activité $j$ finie après la fin de l'activité $i$.

### Avec des durées (*timelag*), les contraintes sont valides si...

$F_iS_j(tl_{min}, tl_{max})$ l'activité $j$ démarre entre $tl_{min}$ et $tl_{max}$ unités de temps après la fin de l'activité $i$.

$S_iS_j(tl_{min}, tl_{max})$ l'activité $j$ démarre entre $tl_{min}$ et $tl_{max}$ unités de temps après le démarrage de l'activité $i$.

$F_iF_j(tl_{min}, tl_{max})$ l'activité $j$ finie entre $tl_{min}$ et $tl_{max}$ unités de temps après la fin de l'activité $i$.

# Activité

## Contraintes de précédence

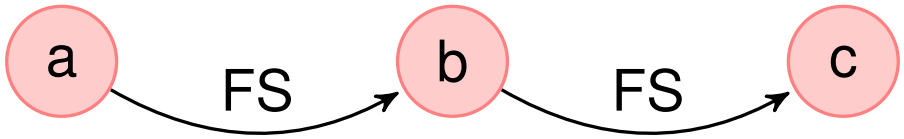



FIGURE: *a* démarre. Quand *a* fini, *b* peut démarrer, quand *b* fini, *c* peut démarrer.

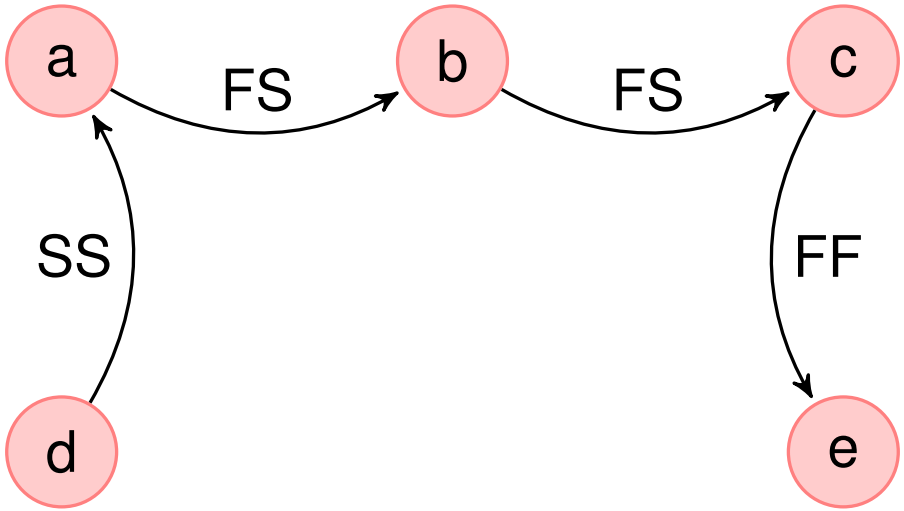



FIGURE: *a* doit démarrer en même temps que *d*. Quand *a* fini, *b* peut démarrer, quand *b* fini, *c* peut démarrer. Quand *c* fini, *e* doit finir.

# Activité

## Contraintes de précédence

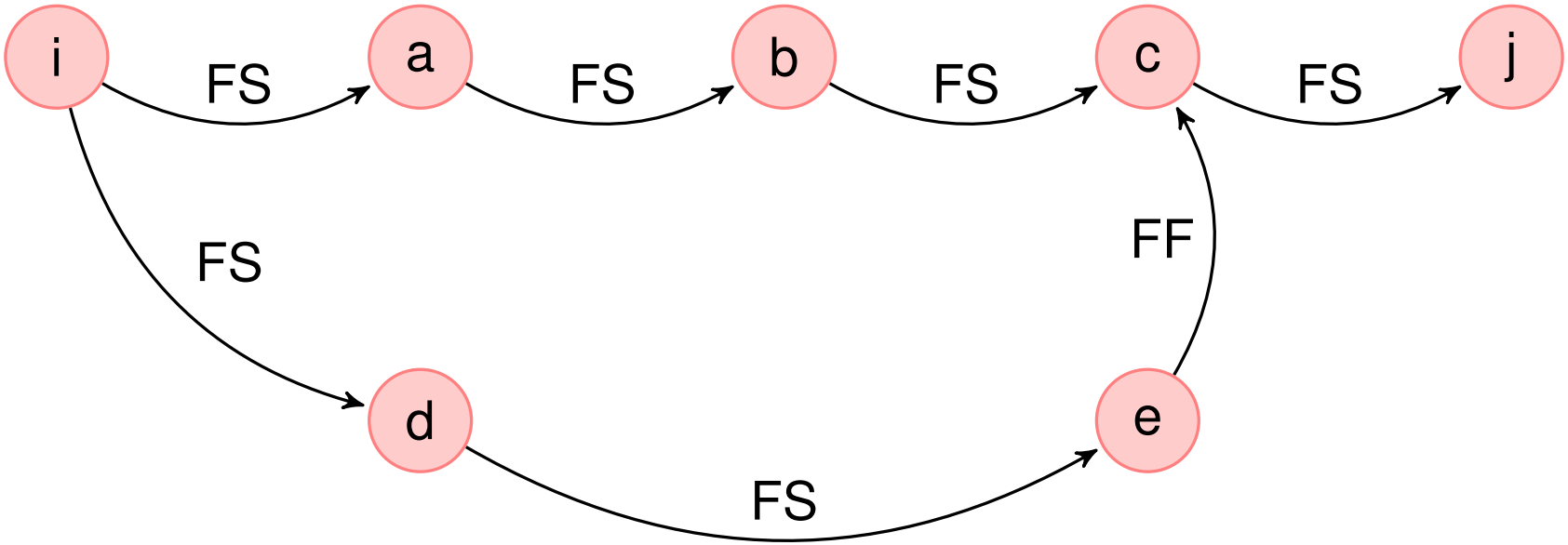



FIGURE: lorsque *i* fini, *a* et *d* peuvent démarrer etc.

# Activité

## Description des états

**Wait** l'activité est en attente : au moins une de ses contraintes (temporelles, conditions ou précédence) n'est pas valide

**Started** l'activité est démarrée : toutes les contraintes sont valides (contraintes temporelles, contraintes de précédences et pré-conditions)

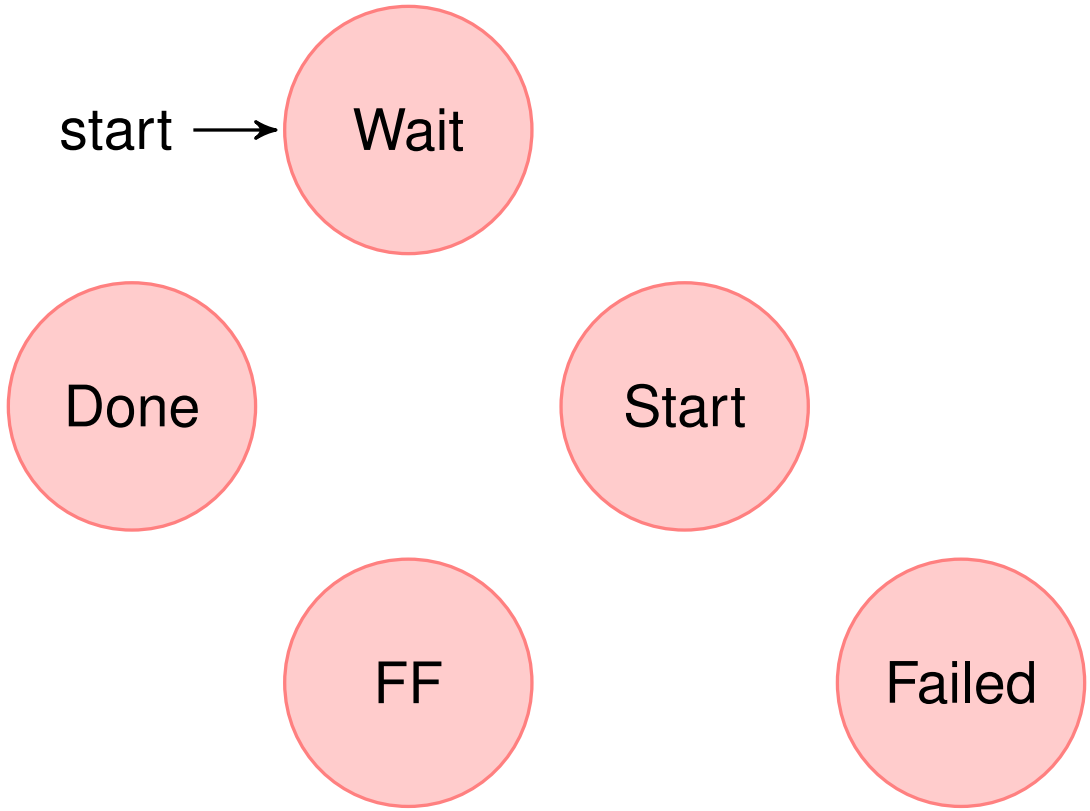
**FF** l'activité est close par le système opérant (réception d'un événement sur le port ack) cependant les contraintes de précédence de type FF peuvent encore la faire passer en état Failed

**Done** l'activité est finie : un ack de type "done" a été reçu et les contraintes de précédence de type FF sont valides

**Failed** l'activité a raté : l'activité est sortie de ses contraintes temporelles, le jeu du graphe de précédence l'a fait passé dans cet état ou le système opérant l'a invalidé.

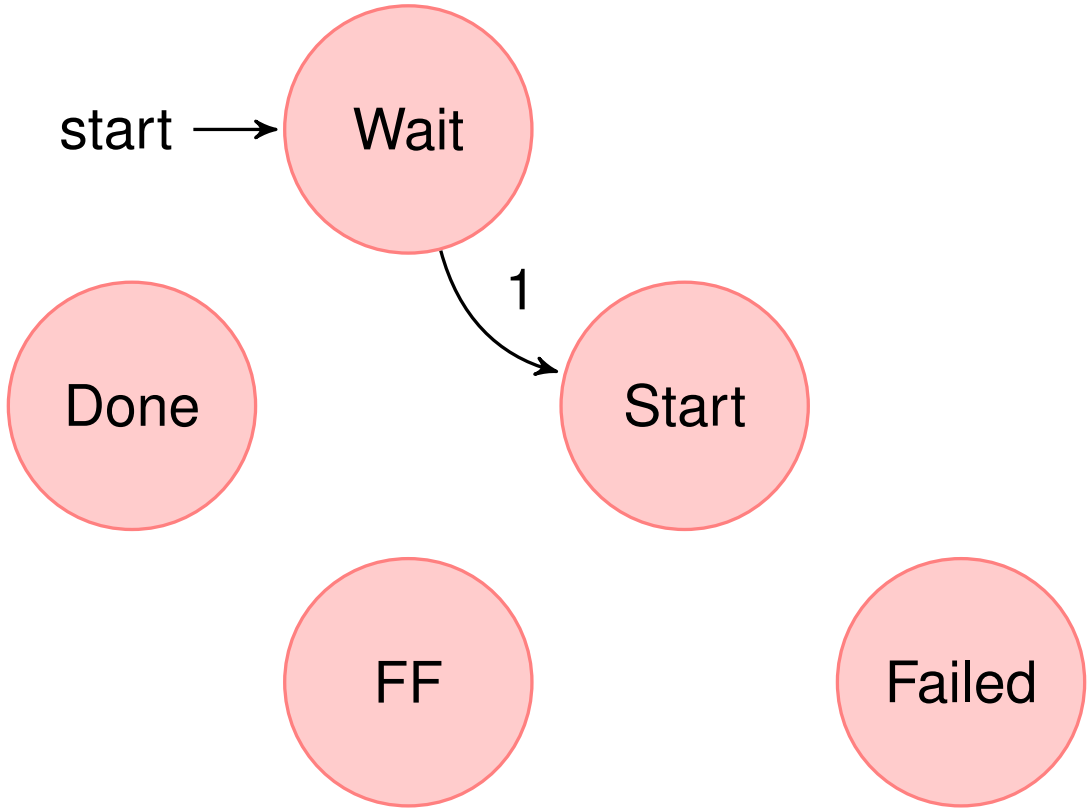
# Activité

## Graph d'états



# Activité

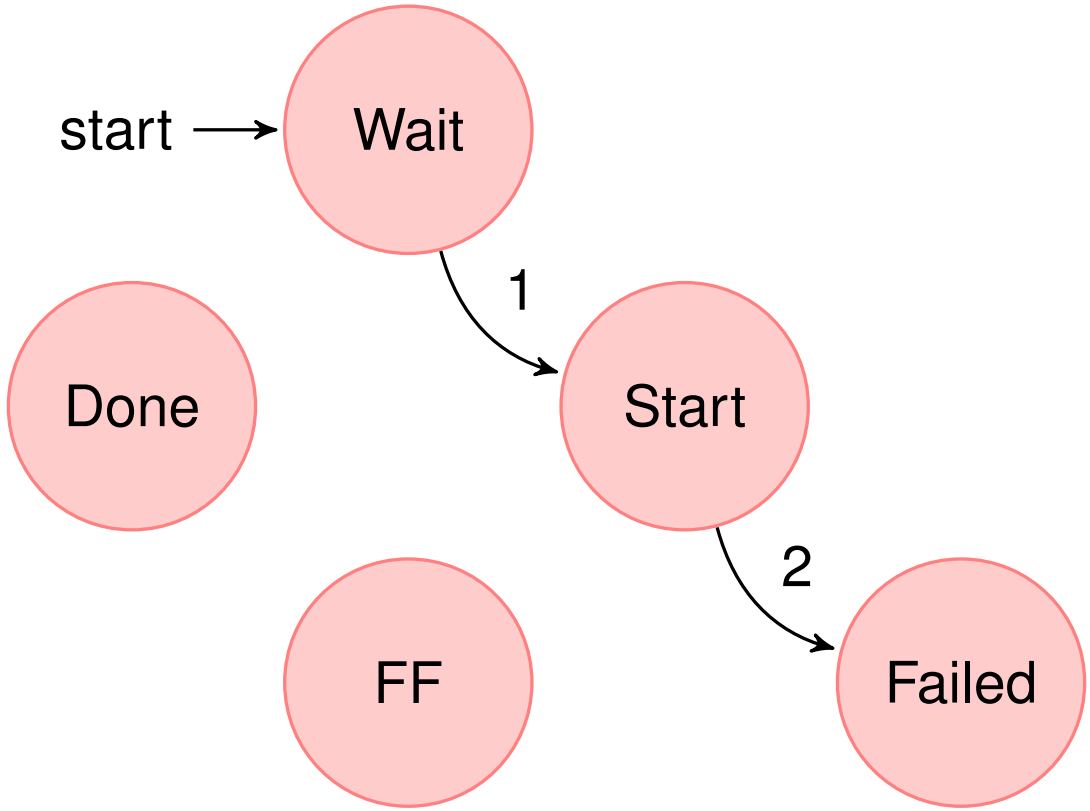
## Graph d'états




1. Si la tâche est en attente, la date de début d'activité est valide et les contraintes de précédence sont toutes valides ou au moins une est valide (paramètre). Des régles d'inactivation peuvent aussi s'appliquer.

# Activité

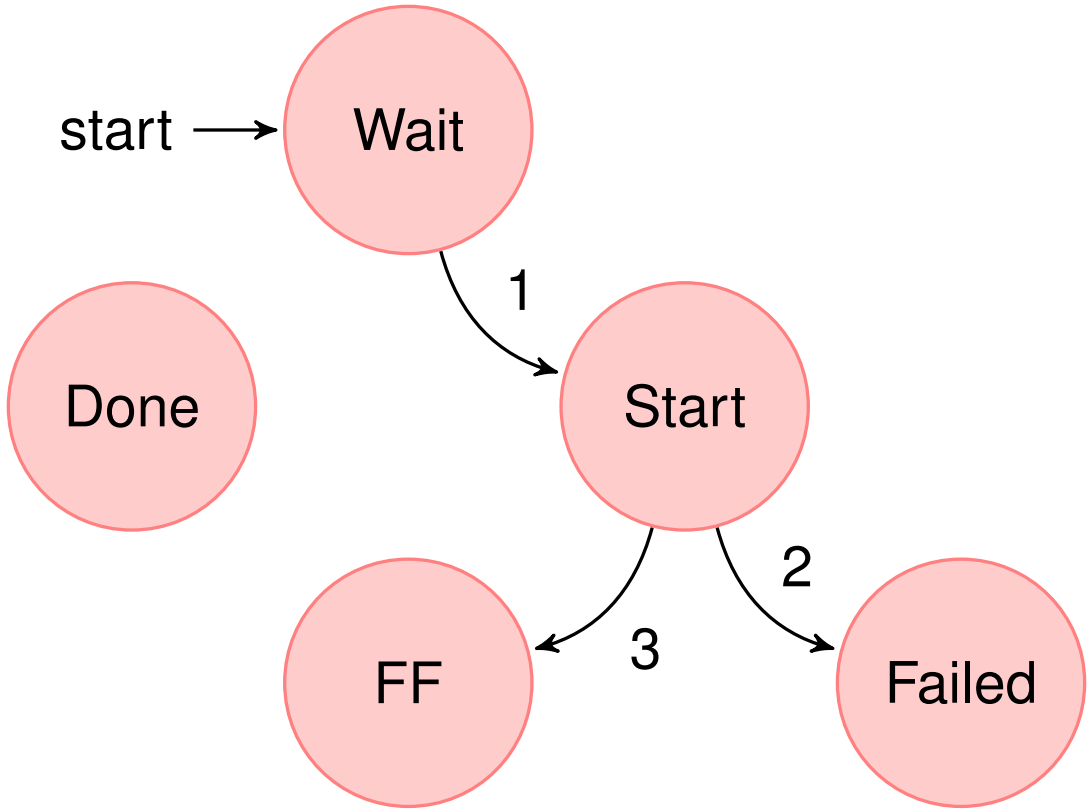
## Graph d'états




2. Si la tâche est en cours de simulation mais que la date de fin est dépassée ou si le système opérant invalide l'activité ou si des regles d'inactivation s'appliquent.

# Activité

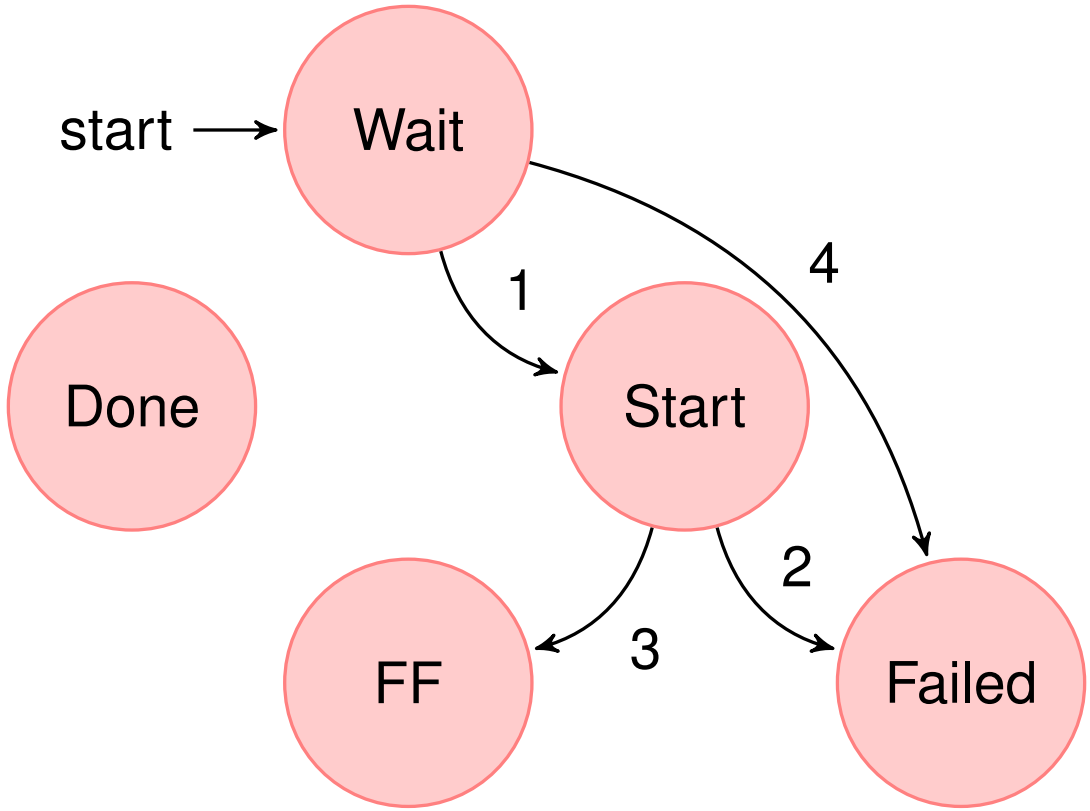
## Graph d'états




3. Si le système opérant valide l'activité

# Activité

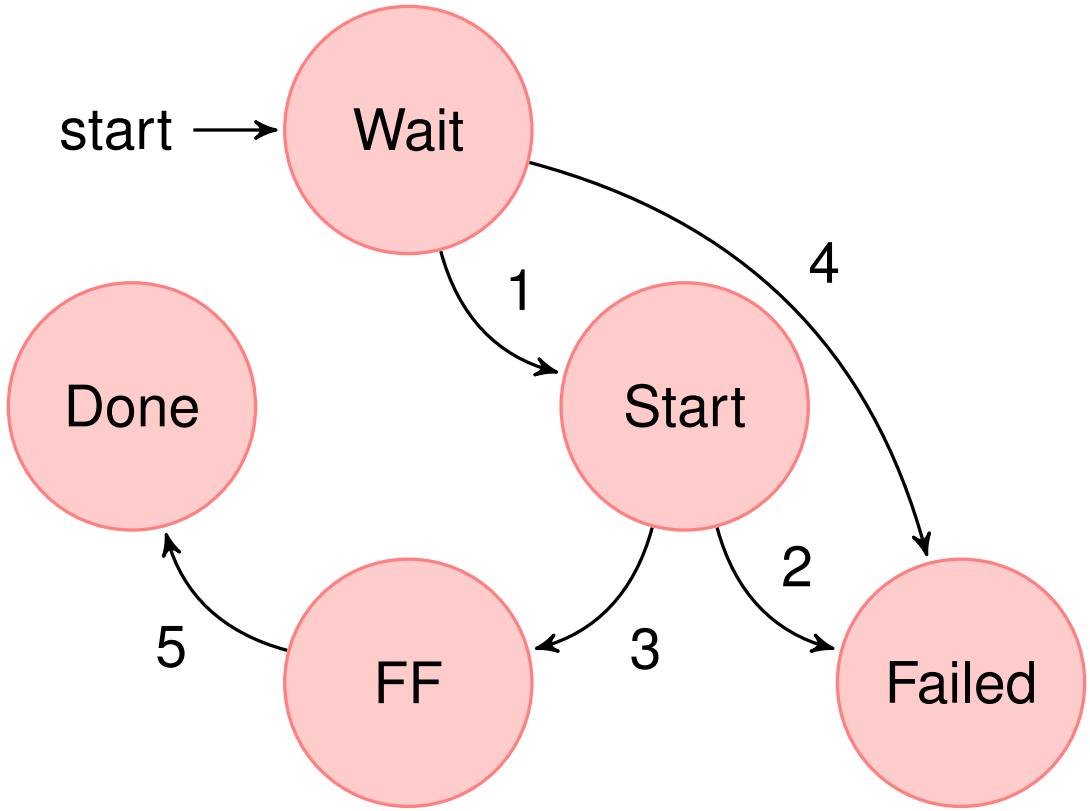
## Graph d'états




4. Si la tâche n'est jamais démarrée et que la date de fin est dépassée

# Activité

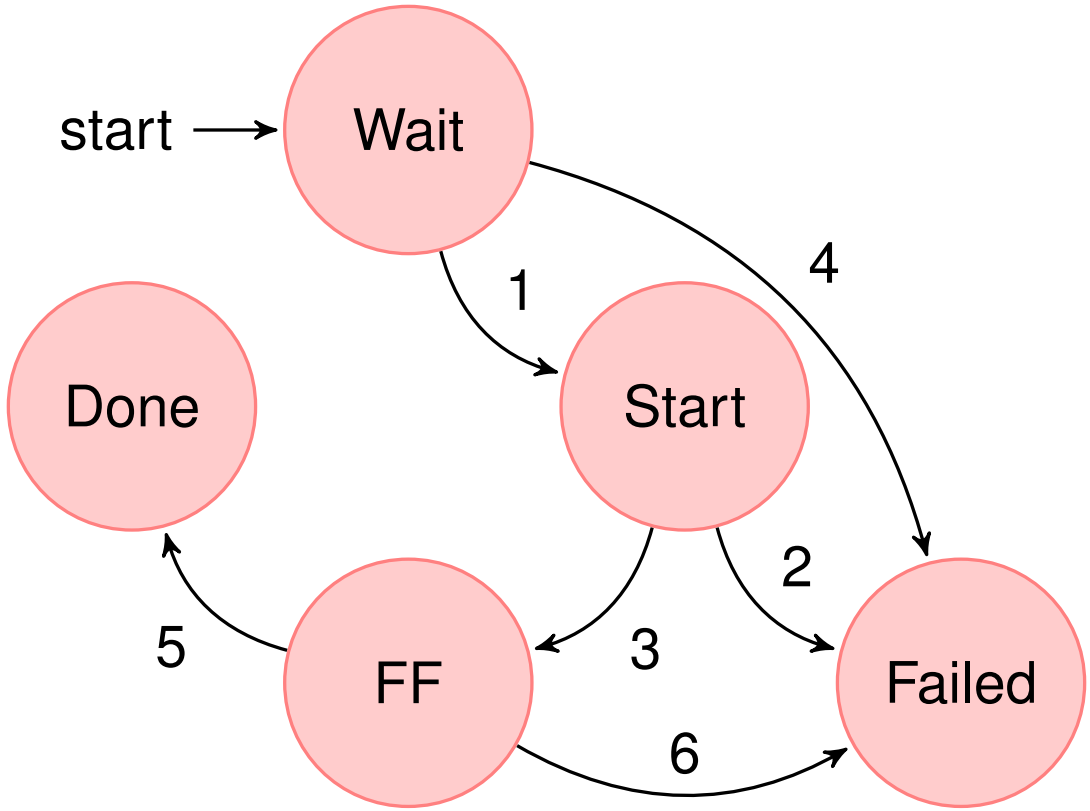
## Graph d'états




5 Si la tâche est finie et que les contraintes de type FF sont satisfaites.

# Activité

## Graph d'états




6 Si la tâche est finie mais que les containtes de type FF ne sont pas satisfaites.

# Plan

# Fichier de plan et c++

Pour bénéficier de plus souplesse :

- Les activités, les contraintes de précédences et les règles sont définies dans un fichier indépendant, (ou dans les conditions expérimentales ou dans le code C++).
- La base de connaissance reste dans les attributs de l'agent. La mise à jour des faits reste dans le C++. Les prédicats et les fonctions ack et output restent en C++.

# Fichier de planification

## Définition des règles

```
rules {
  rule {
    id = "identifiant" ;
    predicates = "pred1", "pred2", "pred3"; # optionnel, lien avec le C++
  }
  ... # liste de règles.
}
activities {
  ... # liste d'activités
}
precedences␣{
␣...␣#␣liste␣de␣contraintes␣de␣précédence
}
```

- une règle est une combinaison de prédicats (chaînes de caractères) définie dans le C++.

# Fichier de planification

## Définition des règles

```
rules {
  rule {
    id = "identifiant" ;
    predicates = "", "", "", ""; # optionnel, lien avec le C++
  }
  ... # liste de règles.
}
```

- une règle est une combinaison de prédicats (chaînes de caractères) définie dans le C++.

# Fichier de planification

## Définition des activités

```
activities {
  activity {
    id = "identifiant";
    rules = "", "", "", "";          # optionnel
    rules-fail = "", "", "", "";     # optionnel
    temporal {                       # optionnel
      minstart = 2;                  # (1) | une configuration :
      maxstart = 3;                  # (2) | (1, 4) ou (1, 2 , 4) ou
      minfinish = -infinity;         # (3) | (1, 3, 4) ou (1, 2 , 3, 4).
      maxfinish = infinity;          # (4) |
    }
    ack = "";                 # optionnel (lien avec le C++).
    output = "";              # optionnel (lien avec le C++).
  } ... # liste d'activités
}
```

- un activité a un nom, des règles d'activation, des contraintes temporelles, une fonction de changement d'état et une fonction de sortie.

# Fichier de planification

## Définition des contraintes de précédence

```
precedences {
  precedence {
     type = SS | FF | FS;
     first = "activity source";
     second = "activity destination";
     mintimelag = 0;          # un réel positif ou égal à 0 ou infinity.
     maxtimelag = infinity; # un réel positif ou infinity.
  }
  ... # liste de contraintes de précédence.
}
```

- les contraintes de précédences sont de type FS, FF ou SS entre deux activités et avec ou sans *timelag*.

# Fichier de planification

## Outils supplémentaires, les séquence d'activite

```
activities {
  sequence-activity {
     id-prefix = "id";
     number = 2;                  # optionnel (defaut infinie)
     ...                          # optionnel, "rules", "rules-fail",
     ...                          #    "ack", "output" pour toutes les
     ...                          #    activités
     temporal {                   # optionnel, contraintes temporelles
       ...                        #    de la 1ere activite
     }                            #
     temporal-sequence {          # optionnel, contraintes de precedences
       precedence {               #    entre 2 activités de la séquence
...                     #
       }                          #
     }                            #
  }
}
```

- les contraintes de précédences sont de type FS, FF ou SF entre deux activités et avec ou sans *timelag*

# Exemple 1

```
class MaDecision: public ved::Agent {
public:
    MaDecision(const vd::DynamicsInit& mdl, const vd::InitEventList& evts)
        ved::Agent(mdl, evts)
    {
        addFacts(this) +=
                F("fait", &MaDecision::fait);
        addPredicates(this) +=
                P("pred1", &MaDecision::pred1);
;
        addOutputFunctions(this) +=
                O("out",&MaDecision::out);
        addAcknowledgeFunctions(this) +=
                A("ack", &MaDecision::ack);

        KnowledgeBase::plan().fill("monFichierPlan.txt");
    }
}
```

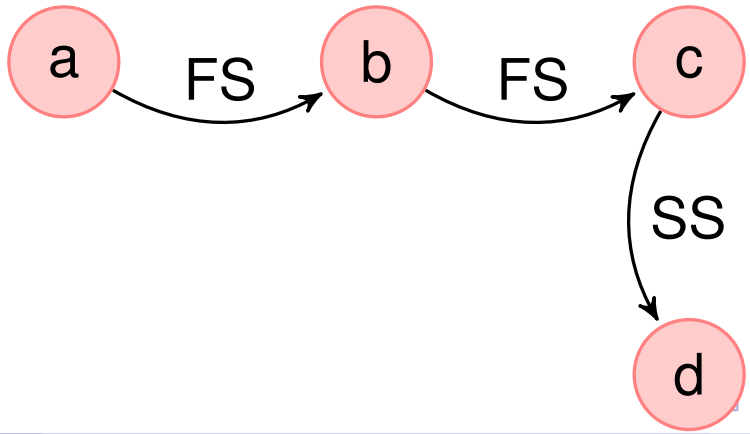
# Exemple 1

```
class Test : public vle::extension::decision::Agent
{
  Test() {
    addActivity("A");
    addActivity("B", vle::devs::negativeInfinity, vle::devs::infinity);
    addActivity("C", 12.0, vle::devs::infinity);
    addActivity("D", vle::devs::negativeInfinity, 12.0);
    addActivity("E", 12.0, 22.0);
  }
};
```

5 activités, avec uniquement des contraintes temporelles. Dans ce cas *a*, *b* et *d* vont démarrer en début de simulation, suivit de *c* et *d* lorsque la simulation atteint la date 12.0.

# Exemple 2

```
class Test : public vle::extension::decision::Agent
{
  Test() {
    Activity& a = addActivity("A");
    Activity& b = addActivity("B");
    Activity& c = addActivity("C");
    Activity& d = addActivity("D");
    addFinishToStartConstraint(a, b);
    addFinishToStartConstraint(b, c);
    addFinishToFinishConstrait(c, d);
  }
}
```



# Exemple 3

```
class Test : public vle::extension::decision::Agent
{
  Test() {
    addFact("pluie", boost::bind(&Test::majPluie, this, _1));
  }

  void majPluie(const vle::value::Value& value) {
    std::rotate(mPluies, mPluies + 1, mPluies + 5);
    mPluies[0] = value.toDouble().value();
  }

   double mPluies[5]; // vecteur de 5 réels : 5 jours de quantité
                      // de pluie : la base de connaissances.
};
```

- Lors de l'arrivée d'un événement sur sur le port Pluie du modèle Test la fonction majPluie est appelée et met à jour le vecteur des 5 dernières températures.

# Exemple 4

```
class Test : public vle::extension::decision::Agent {
  Test() {
    addFact("pluie", boost::bind(&Test::majPluie, this, _1));
    Activity& a = addActivity("a");
    Rule& r = a.addRule("RuleA");
    r.add(boost::bind(&Test::solPortant, this));
  }

  void majPluie(const vle::value::Value& value) {
    std::rotate(mPluies, mPluies + 1, mPluies + 5);
    mPluies[0] = value.toDouble().value();
  }

  bool solPortant() const {
    return std::accumulate(mPluies, mPluies + 5, 0.0) <= 15;
  }
                        // a demarre quand RuleA est valide
  double mPluies[5];    // c-a-d, lorsque la somme sur 5 jours
                        // est >= 15mm.
```

# Exemple 5


- un port Pluie pour la mise à jour de la base de connaissance,
- un port ack pour la réception des résultats du système opérant,
- enfin, un port de sortie a connecté au système opérant.

# Exemple 6

## Relation : durant

```
class During : public vle::extension::decision::Agent
{
  During()
  {
    addActivity("A");
    addActivity("B");

    // B needs to start at begin time of A + 1.0.
    // A needs to finished at end time of B + 1.0.
    addStartToStartConstraint("A", "B", 1.0);
    addFinishToFinishConstraint("B", "A", 1.0);
  }
};
```

# Exemple 7

## Relation : égale

```
class Equal : public vle::extension::decision::Agent
{
  Equal()
  {
    addActivity("A");
    addActivity("B");

    // A and B need to start and finish a the same time.
    addStartToStartConstraint("A", "B");
    addFinishToFinishConstraint("A", "B");
  }
};
```

# Exemple 8

## Réagir à la fermeture d'activité

Comment réagir à la fermeture d'une activité par le système opérant :

```
class React : public vle::extension::decision::Agent
{
  React()
  {
    Activity& a = addActivity("A");
    a.addAck(&onUpdateActivityA);
  }

  virtual void onUpdateActivity(const std::string& name,
                                const Activity& a,
                                const value::Value& value)
  {
    std::cout << "la tache " << name << " a change d etat."
              << "Elle se trouve dans l etat : " << a.state();
  }
};
```

# Gestion des durées d'activité

- durée effective de l'activité $=$ date de "done" - date "start"
- $\neq$ durée d'activité a priori
- Allocation de resources pour chaque activité par slot.

# Approche retenue ici

- L'allocation des ressources fait partie intégrante du système opérant (modèle fournit).
- Construire une Allocation à $t$,
  $a_t : \text{Ressources} \rightarrow \text{ActivitesEnCours} \cup \{null\}$
  - qui soit valide vis-à-vis des ressources nécessaires pour la réalisation des activités.
  - qui minimise les priorités des activités choisies.
  - qui maximise l'utilisation des ressources.

# Gestion des ressources