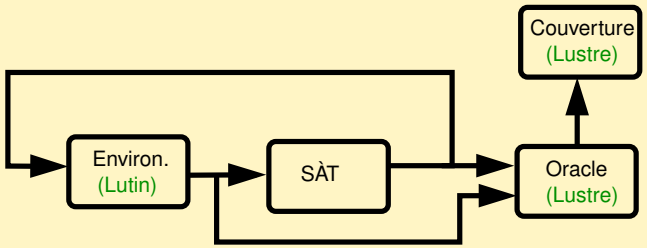
Outline

Plan

Lurette - Tests fonctionnels automatisés de systèmes réactifs (synchrones)

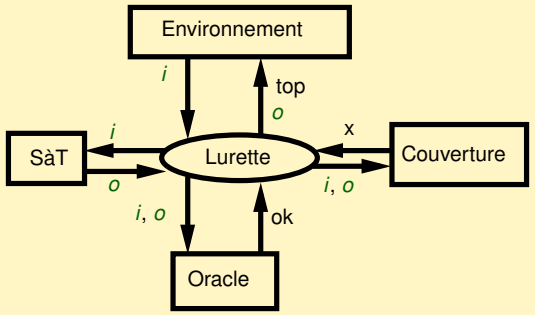
- Test fonctionnel (boîte noire)
  - confronter une implémentation et une spécification
- Test Automatisé
  - génération des stimuli (entrées du SàT)
  - dépouillement du résultat des tests (oracle)
- Basé sur une description formelle
  - des attendus du *Système à Tester* (SàT)
  - des hypothèses faites sur son environnement
- Systèmes réactifs
  - le SàT réagit à l'environnement qu'il cherche à contrôler (feedback)
  - un environnement *réaliste* doit faire de même

Test de systèmes bouclés

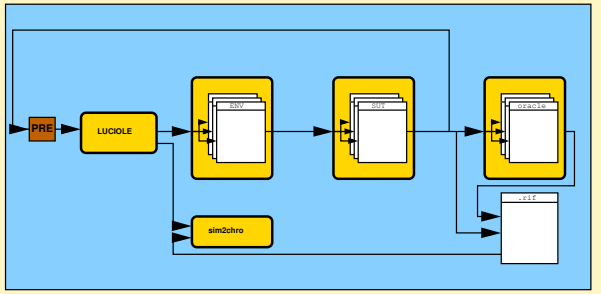


- L'environnement (Stimulateur) peut-être vu comme un système réactif non-déterministe
- Générateur de vecteurs de test sous contrainte

Vue Flot de données à chaque cycle



Vue synthétique de l'outil



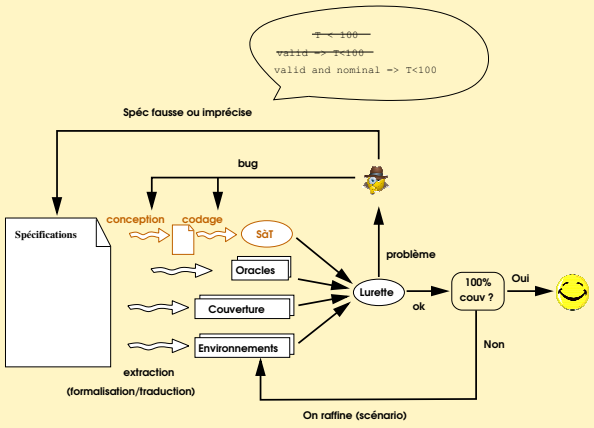
Expérimentations sur des systèmes «Synchrones»

- Hispano-Suiza (Scade)
  - Système de Contrôle-commande d'un moteur (M88)
  - Étude menée par un ingénieur en stage 2ème année (3 mois)
  - 17 bogues dont 13 nouveaux, et 1 dans un système en production
- Renault (Sildex/RT-Builder)
  - Objectif : réduire l'influence des conditions extérieures (pente, vent, etc.) sur le comportement du freinage
  - SUT : système de freinage+modèle simulink du vehicule
  - Env. : Le conducteur + les perturbations extérieures
  - Oracle : comme le SUT, mais sans pertubation
- Astrium (AADL et Scade)
  - ATV/PFS - Un systeme redondé avec un maître et un esclave
  - « Quand l'un des 2 systèmes abondonne son rôle de maître, l'autre le prend en moins de 2 ticks de l'horloge principale »

Expérimentations sur des systèmes hétérogènes

- Le projet Minalogic COMON (2009-2012) a été l'occasion d'élargir le domaine d'application des nos outils
  - Corys TESS, Alices (Simulateur énergie et transport)
  - Atos Origin, Scada (télésurveillance et acquisition de données)
  - Rolls-Royce, Scade
- La conclusion des expérimentations menées dans COMON est que Lurette est autant un outil de test qu'un outil d'ingénierie des spécifications (exigences fonctionnelles)

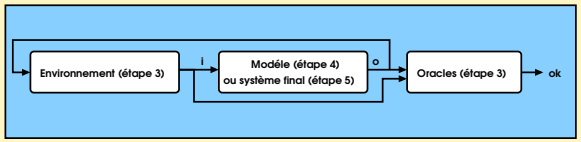
Processus de test Lurette



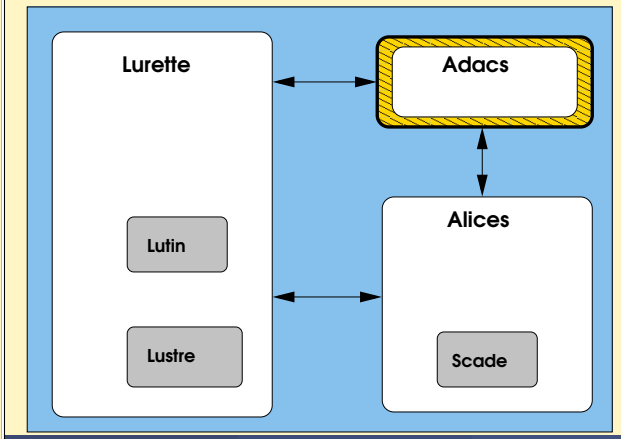
Plan

Test et Développement sans rupture dans COMON

1. Analyse des besoins
2. Spécification des exigences fonctionnelles en langue naturelle et à l'aide de dessins informels
3. Formalisation de ces exigences fonctionnelles (Oracles)
4. Obtention au plus tôt d'un modèle abstrait et exécutable satisfaisant ces exigences (Alices)
5. Obtention d'une implémentation finale satisfaisant ces exigences



Architecture logicielle du banc test

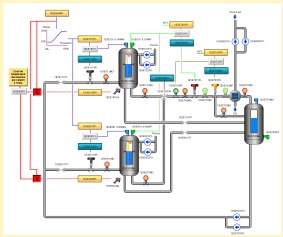


Expérimentations effectuées sur le cas d'étude

- Formalisation
  - Lustre pour les oracles
  - Lutin pour les environnements/stimulateurs
- Bibliothèque d'utilitaires génériques pour écrire les oracles et de stimulateurs
  - Vérifier qu'une valeur numérique est dans sa plage
  - Calculer les états du système
    - nominal, dégradé, 2/3-1/3, urgence
  - Détecter la stabilité d'une variable (`est_stable`)
  - Effectuer une action et attendre la stabilité
  - etc.

Le cas d'étude COMON

- Une étude de cas représentative
  - D'un circuit hydraulique nucléaire
    - Circuit physique
    - Capteurs et actionneurs
  - Et de son contrôle-commande
    - Redondance
    - Régulation
    - Sûreté
  - Son démonstrateur
    - Procédé simulé
    - N1 standard simulé
    - N1 classé émulé
    - N2 réel



Ce que l'on voit dans la Démo

- 4 ateliers hétérogènes communiquer
- Scénarios de stimulation (Environnement)
  - `n` (0, puis 1, puis 2) pannes au hasard, avec comme contrainte d'éviter (si possible) l'action de sûreté
  - Un opérateur virtuel bouclé
    - Une consigne cible est choisie au hasard
    - L'opérateur virtuel change la consigne d'au plus « `pas` » pour se rapprocher de la cible ; il attend la stabilité du sàt avant de la changer la consigne à nouveau (boucle 1)
    - quand la consigne atteint sa cible, on rechoisit une cible (boucle 2)
- Propriétés Invalidées (Oracles)
  - Transitions entre états du système (nominal <-> 2/3-1/3 <-> dégradé <-> urgence) : anomalie dans le calcul de la situation sur le N2
  - Seuil haut dépassé : anomalie dans le voteur N1 qui n'élimine pas le capteur en panne d'une fausse moyenne

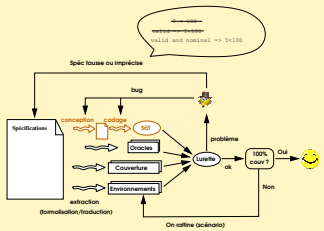
Un oracle surveillant la stabilisation du système

En mode nominal, après tout changement de consigne, toutes les valeurs issues des capteurs doivent être stables au bout de 2 minutes

```
C = vrai_depuis(aucun_chgt_consigne and nominal,120.0) ;
ok = (C => est_stable)
```

- Couvrir ce test, c'est générer une séquence où C est vraie
- D'où un besoin de scénarios où la consigne ne change pas toutes les secondes

Présentation des 2 stimulateurs de la démo



- Génération de pannes : contraintes qui génèrent plein de cas
- Modélisation d'un opérateur : illustre la capacité de Lutin à exprimer des scénarios assez évolués (couverture)

Un générateur de pannes (pseudo-)aléatoire

- Objectif : générer `n` pannes sans déclencher d'action classée
  - 2/2 ou 3/4 capteurs rédondés en défaut
- On tire au hasard parmi les solutions des contraintes
- Un scénario minimal pour
  - ne pas changer de panne à chaque cycle
  - en changer tout de même au bout de X cycles (ou si reset)

```
node(n:int; reset:bool) returns(P1,P2,P3,P4,P5,P6) =
let SOS=deux_vrai(P1,P2) or trois_vrai(P3,P4,P5,P6) in
let b2i(b:bool) = if b then 1 else 0 in
let nb_pannes = b2i(P1)+...+b2i(P6) in
loop {
 { (not SOS and nb_pannes = n) |> nb_pannes = n }
 fby loop X {not reset and P1=pre P1 and ... P6=pre P6}
}
```

[Pannes-i.lut ; Démo]

Un opérateur (pseudo-)aléatoire scénarisé

Change la consigne en surveillant la valeur de capteurs de niveaux

1. Choisit une consigne Cible (dans [0 ; 100])
2. Utilise un nœud qui va amener la consigne à la cible pas à pas
3. Quand la cible est atteinte, Recommence en 1.

Le nœud change_consigne_pas_a_pas est du même style

1. Maintient la consigne tant qu'on n'a pas la stabilité
2. Se rapproche de la consigne cible d'au plus pas
3. Si la consigne cible n'est pas atteinte, Recommence en 1

Cette modélisation illustre la capacité de Lutin à exprimer des scénarios assez évolués (couverture)
[operateur.lut ; Démo]

Le noeud operateur en Lutin

```
node operateur(est_stable:bool)
returns(C,Cible:real;phase:int)=
   phase=0 and between(Cible,0.0,100.0) and C=Cible
 fby
 loop {
   loop { phase=1 and -- Attend la stabilite du systeme
     not est_stable and
     maintient(C) and maintient(Cible)
   }
   fby phase=2 and -- Choisi la prochaine consigne
     between(Cible, 0.0, 100.0) and
     maintient(C)
   fby -- Vise la cible pas a pas
     assert phase=3 and maintient(Cible) in
     run C := change_consigne_pas_a_pas(
                        est_stable,pre C,pre Cible,2.0)
     in
       while (C <> Cible)
 }
```


Le noeud change_consigne_pas_a_pas en Lutin

```
node change_consigne_pas_a_pas(
  est_stable : bool; -- calcule par ailleurs
  C, Cible, pas : real)
returns (newC : real) =
 loop {
   loop { not est_stable and newC = C }
   fby
     if Abs(Cible - C) < pas then
       newC = Cible
     else if Cible < C then
       C - pas < newC and newC < C
     else
       C < newC and newC < C + pas
   fby
   -- on attend que le chgt de Consigne prenne effet
   loop [10] { newC = C }
 }
```


Plan

Conclusion

- Une approche basée sur des langages synchrones (versatilité)
  - Tests unitaires, intégration, etc.
  - Démarche itérative (raffinements successifs)
- Des langages pour des spécifications formelles lisibles
- Une mise en œuvre de l'approche « orientée par les modèles » pragmatique où l'on ne jette rien
  - le modèle n'est pas là pour palier à l'explosion de l'analyse du vrai système mais pour faire de la simulation et de la validation précoce (à toutes les étapes)
- Une façon d'introduire (et de tirer partie) des méthodes formelles dans l'industrie sans se heurter au mur de l'explosion combinatoire
- La société Argosim va être créée (juin 2013) dans le but d'industrialiser ces idées/outils

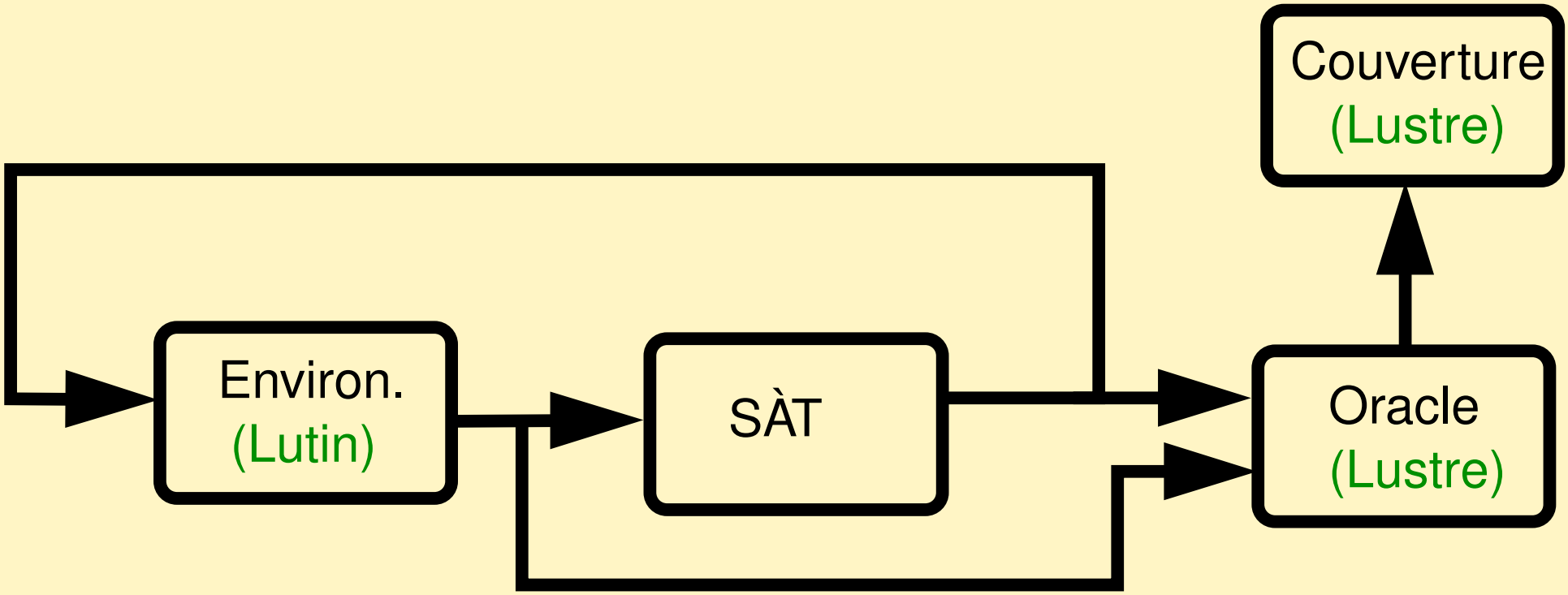
Outline

Plan

Lurette - Tests fonctionnels automatisés de systèmes réactifs (synchrones)

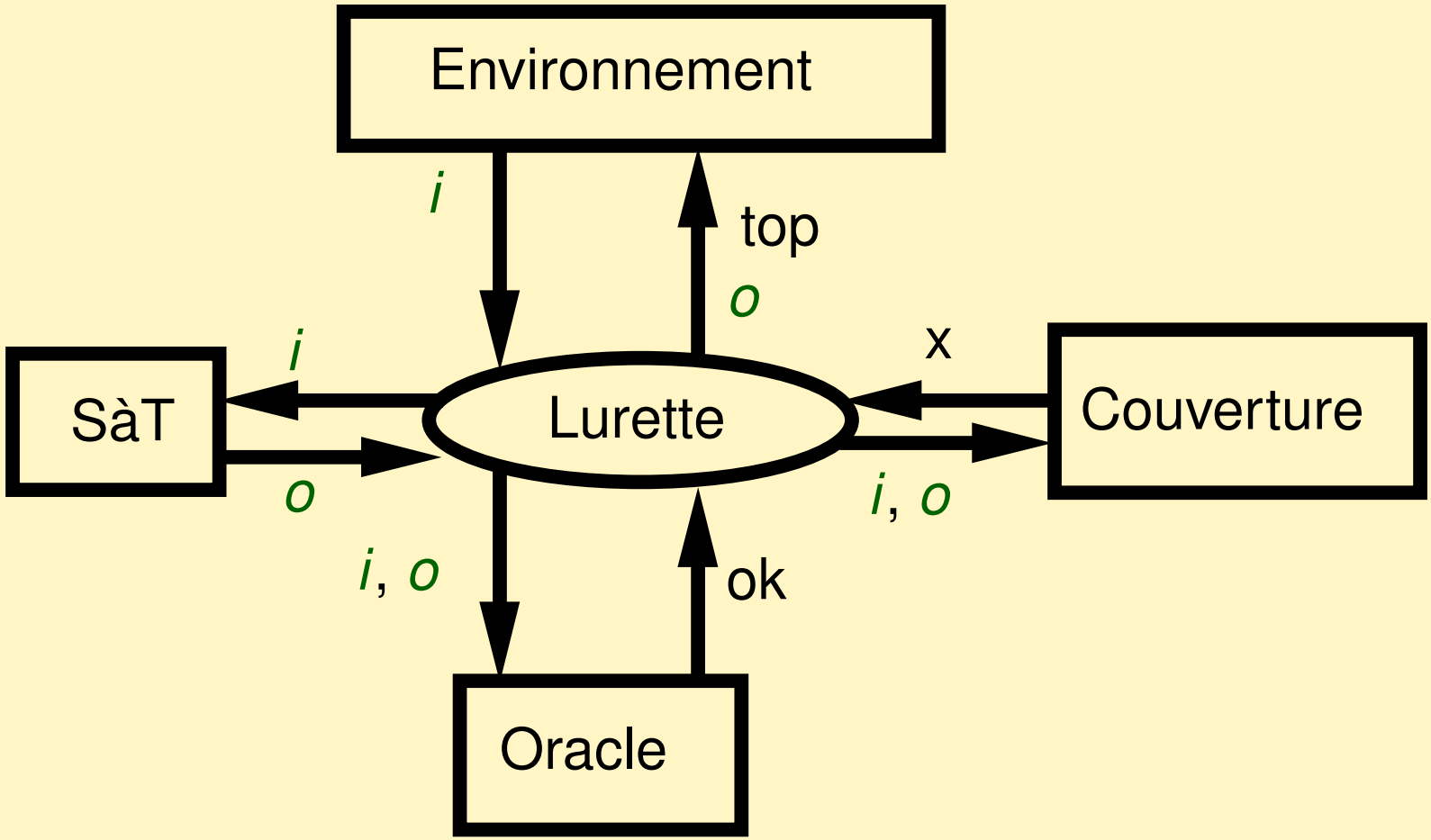
- Test fonctionnel (boîte noire)
  - confronter une implémentation et une spécification
- Test Automatisé
  - génération des stimuli (entrées du SàT)
  - dépouillement du résultat des tests (oracle)
- Basé sur une description formelle
  - des attendus du *Système à Tester* (SàT)
  - des hypothèses faites sur son environnement
- Systèmes réactifs
  - le SàT réagit à l'environnement qu'il cherche à contrôler (feedback)
  - un environnement *réaliste* doit faire de même

Test de systèmes bouclés

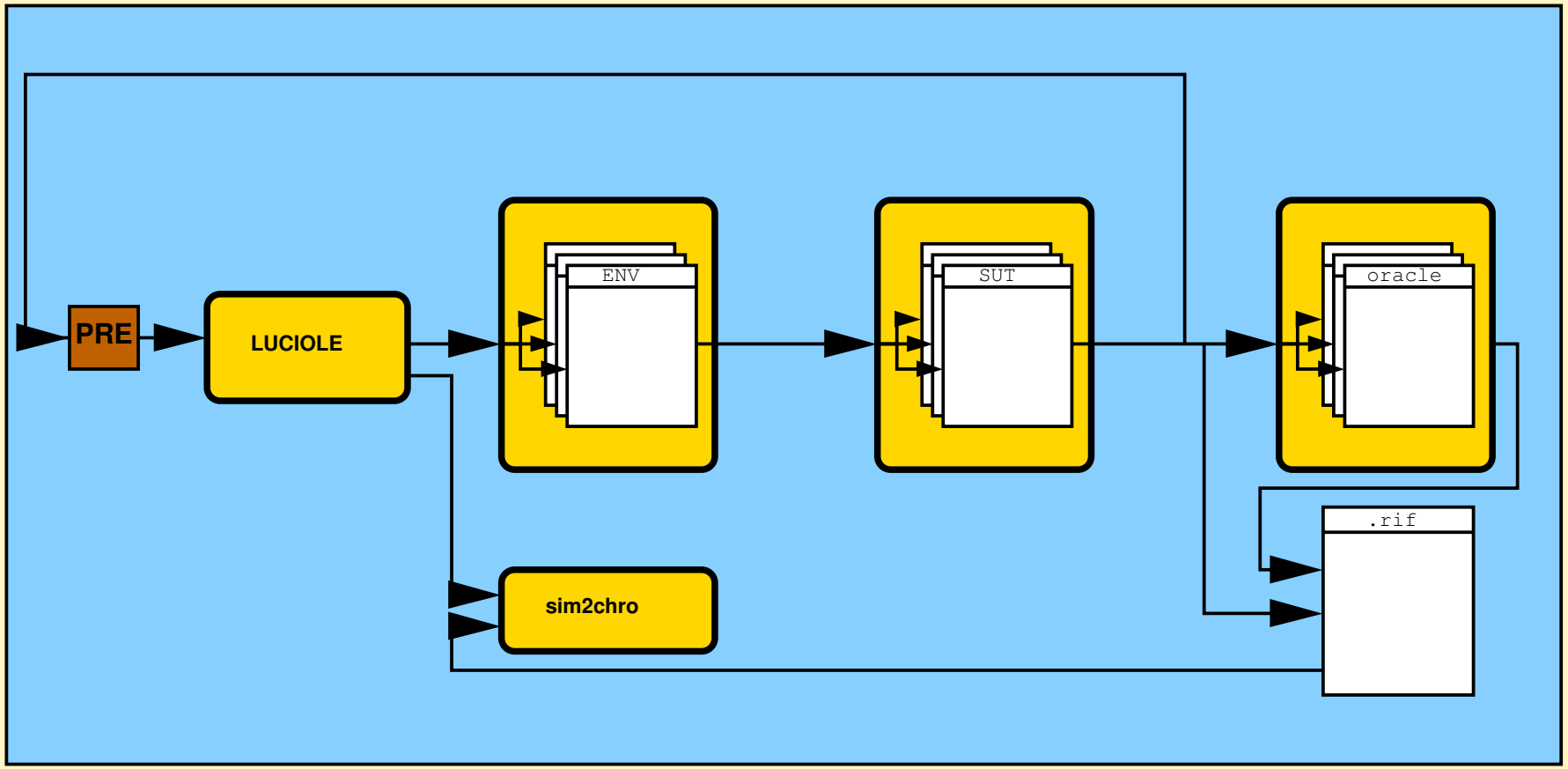


- L'environnement (Stimulateur) peut-être vu comme un système réactif non-déterministe
- Générateur de vecteurs de test sous contrainte

Vue Flot de données à chaque cycle



Vue synthétique de l'outil



Expérimentations sur des systèmes «Synchrones»

- Hispano-Suiza (Scade)
  - Système de Contrôle-commande d'un moteur (M88)
  - Étude menée par un ingénieur en stage 2ème année (3 mois)
  - 17 bogues dont 13 nouveaux, et 1 dans un système en production
- Renault (Sildex/RT-Builder)
  - Objectif : réduire l'influence des conditions extérieures (pente, vent, etc.) sur le comportement du freinage
  - SUT : système de freinage+modèle simulink du vehicule
  - Env. : Le conducteur + les perturbations extérieures
  - Oracle : comme le SUT, mais sans pertubation
- Astrium (AADL et Scade)
  - ATV/PFS - Un systeme redondé avec un maître et un esclave
  - « Quand l'un des 2 systèmes abondonne son rôle de maître, l'autre le prend en moins de 2 ticks de l'horloge principale »

Expérimentations sur des systèmes hétérogènes

- Le projet Minalogic COMON (2009-2012) a été l'occasion d'élargir le domaine d'application des nos outils
  - Corys TESS, Alices (Simulateur énergie et transport)
  - Atos Origin, Scada (télésurveillance et acquisition de données)
  - Rolls-Royce, Scade
- La conclusion des expérimentations menées dans COMON est que Lurette est autant un outil de test qu'un outil d'ingénierie des spécifications (exigences fonctionnelles)

Processus de test Lurette

~~T < 100~~

~~valid => T<100~~

valid and nominal => T<100

Spéc fausse ou imprécise

bug

conception

codage

Spécifications

SàT

Oracles

Couverture

Environnements

problème

Lurette

ok

100% couv ?

Oui

Non

extraction

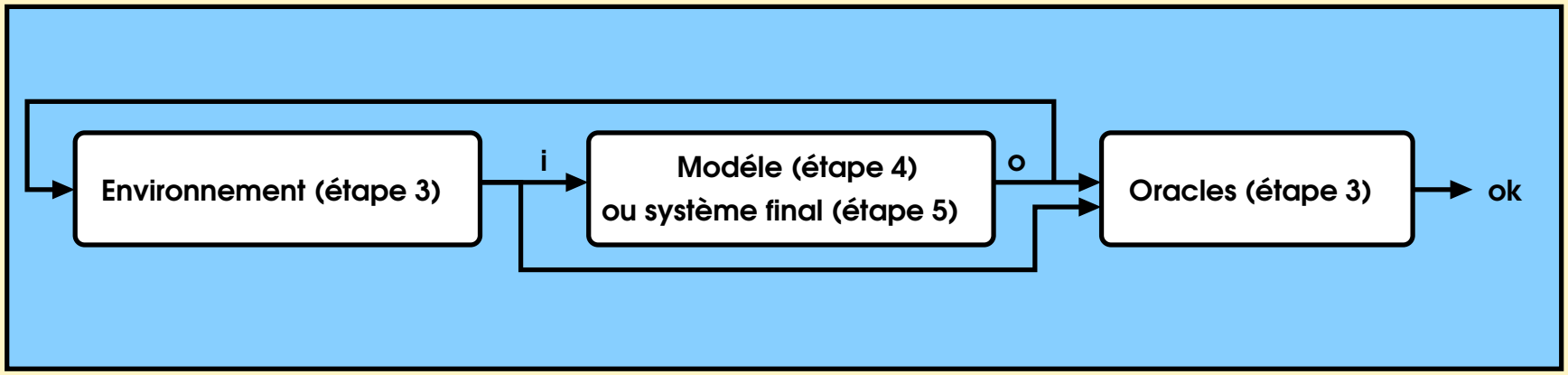
(formalisation/traduction)

On raffine (scénario)

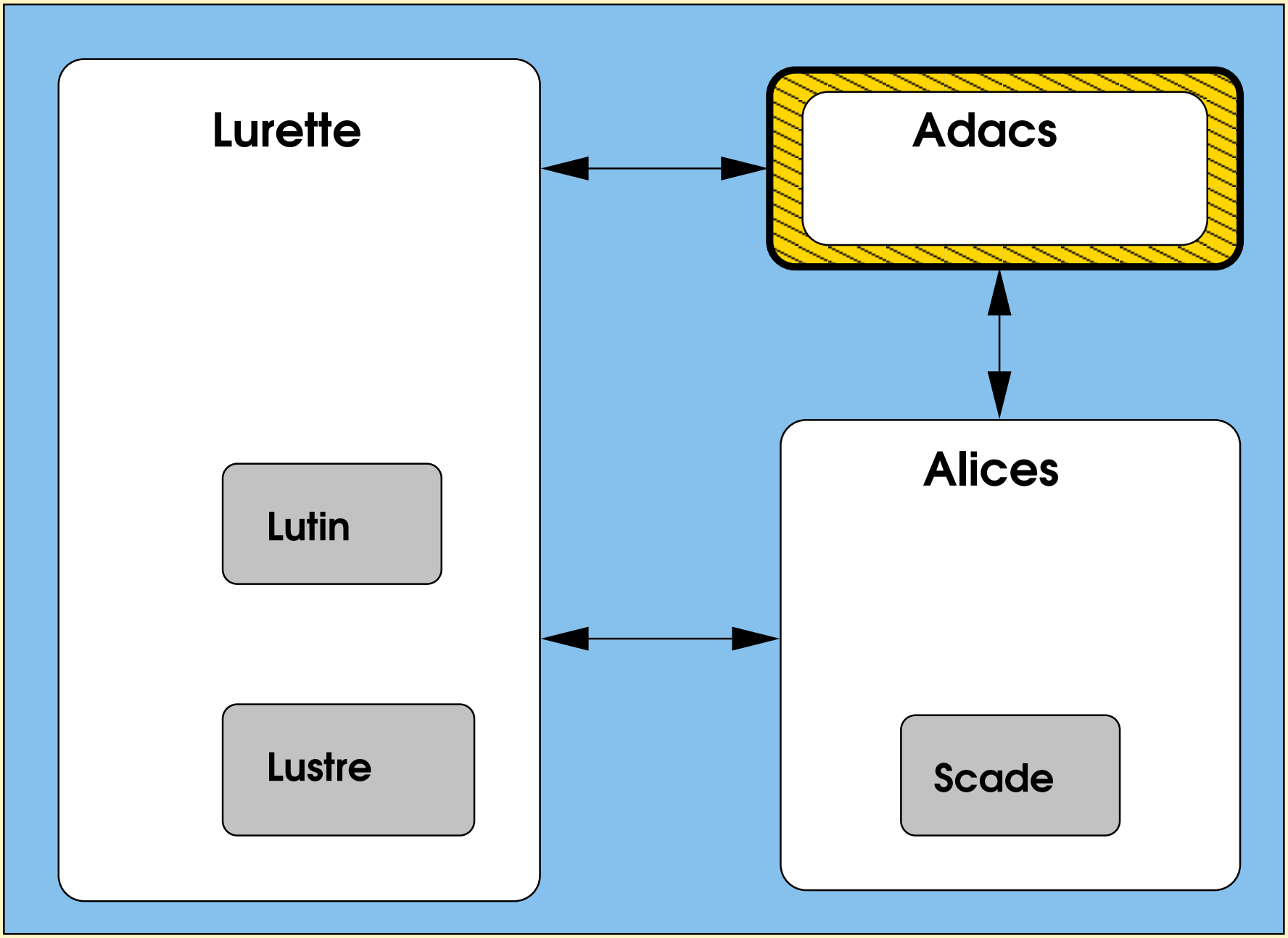
Plan

Test et Développement sans rupture dans COMON

1. Analyse des besoins
2. Spécification des exigences fonctionnelles en langue naturelle et à l'aide de dessins informels
3. Formalisation de ces exigences fonctionnelles (Oracles)
4. Obtention au plus tôt d'un modèle abstrait et exécutable satisfaisant ces exigences (Alices)
5. Obtention d'une implémentation finale satisfaisant ces exigences



Architecture logicielle du banc test

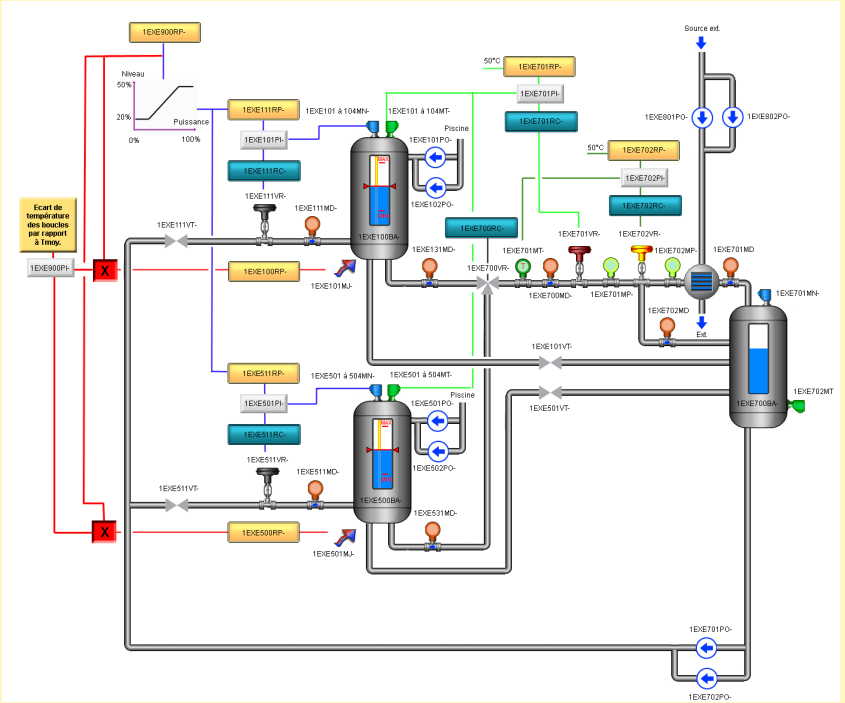


Expérimentations effectuées sur le cas d'étude

- Formalisation
  - Lustre pour les oracles
  - Lutin pour les environnements/stimulateurs
- Bibliothèque d'utilitaires génériques pour écrire les oracles et de stimulateurs
  - Vérifier qu'une valeur numérique est dans sa plage
  - Calculer les états du système
    - nominal, dégradé, 2/3-1/3, urgence
  - Détecter la stabilité d'une variable (`est_stable`)
  - Effectuer une action et attendre la stabilité
  - etc.

Le cas d'étude COMON

- Une étude de cas représentative
  - D'un circuit hydraulique nucléaire
    - Circuit physique
    - Capteurs et actionneurs
  - Et de son contrôle-commande
    - Redondance
    - Régulation
    - Sûreté
  - Son démonstrateur
    - Procédé simulé
    - N1 standard simulé
    - N1 classé émulé
    - N2 réel



Ce que l'on voit dans la Démo

- 4 ateliers hétérogènes communiquer
- Scénarios de stimulation (Environnement)
  - `n` (0, puis 1, puis 2) pannes au hasard, avec comme contrainte d'éviter (si possible) l'action de sûreté
  - Un opérateur virtuel bouclé
    - Une consigne cible est choisie au hasard
    - L'opérateur virtuel change la consigne d'au plus « `pas` » pour se rapprocher de la cible ; il attend la stabilité du sàt avant de la changer la consigne à nouveau (boucle 1)
    - quand la consigne atteint sa cible, on rechoisit une cible (boucle 2)
- Propriétés Invalidées (Oracles)
  - Transitions entre états du système (nominal <-> 2/3-1/3 <-> dégradé <-> urgence) : anomalie dans le calcul de la situation sur le N2
  - Seuil haut dépassé : anomalie dans le voteur N1 qui n'élimine pas le capteur en panne d'une fausse moyenne

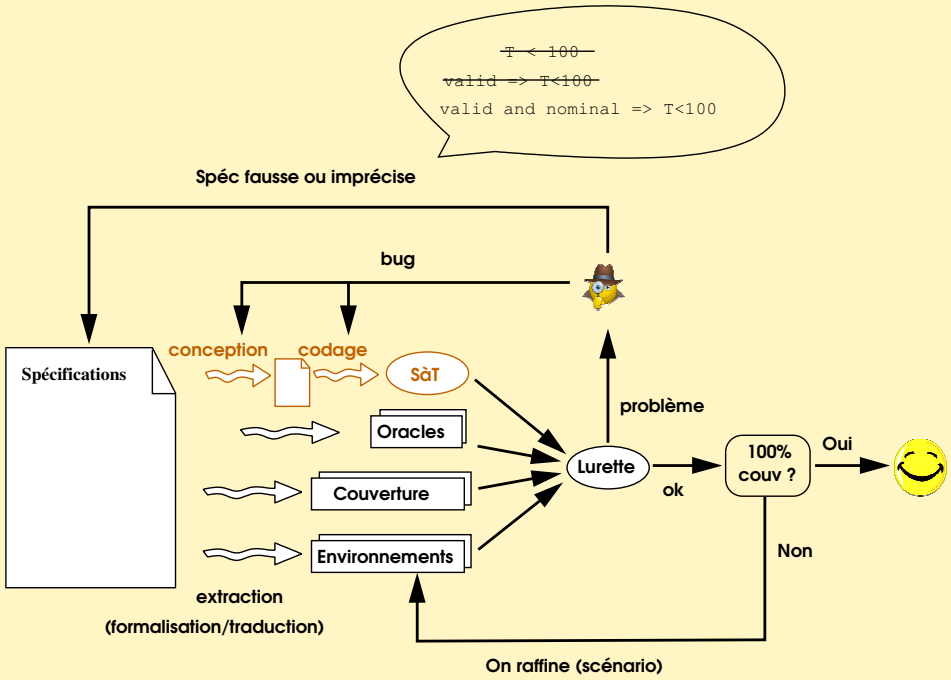
Un oracle surveillant la stabilisation du système

En mode nominal, après tout changement de consigne, toutes les valeurs issues des capteurs doivent être stables au bout de 2 minutes

```
C = vrai_depuis(aucun_chgt_consigne and nominal,120.0) ;
ok = (C => est_stable)
```

- Couvrir ce test, c'est générer une séquence où C est vraie
- D'où un besoin de scénarios où la consigne ne change pas toutes les secondes

Présentation des 2 stimulateurs de la démo



- Génération de pannes : contraintes qui génèrent plein de cas
- Modélisation d'un opérateur : illustre la capacité de Lutin à exprimer des scénarios assez évolués (couverture)

Un générateur de pannes (pseudo-)aléatoire

- Objectif : générer `n` pannes sans déclencher d'action classée
  - 2/2 ou 3/4 capteurs rédondés en défaut
- On tire au hasard parmi les solutions des contraintes
- Un scénario minimal pour
  - ne pas changer de panne à chaque cycle
  - en changer tout de même au bout de X cycles (ou si reset)

```
node(n:int; reset:bool) returns(P1,P2,P3,P4,P5,P6) =
let SOS=deux_vrai(P1,P2) or trois_vrai(P3,P4,P5,P6) in
let b2i(b:bool) = if b then 1 else 0 in
let nb_pannes = b2i(P1)+...+b2i(P6) in
loop {
 { (not SOS and nb_pannes = n) |> nb_pannes = n }
 fby loop X {not reset and P1=pre P1 and ... P6=pre P6}
}
```

[Pannes-i.lut ; Démo]

Un opérateur (pseudo-)aléatoire scénarisé

Change la consigne en surveillant la valeur de capteurs de niveaux

1. Choisit une consigne Cible (dans [0 ; 100])
2. Utilise un nœud qui va amener la consigne à la cible pas à pas
3. Quand la cible est atteinte, Recommence en 1.

Le nœud change_consigne_pas_a_pas est du même style

1. Maintient la consigne tant qu'on n'a pas la stabilité
2. Se rapproche de la consigne cible d'au plus `pas`
3. Si la consigne cible n'est pas atteinte, Recommence en 1

Cette modélisation illustre la capacité de Lutin à exprimer des scénarios assez évolués (couverture)
[operateur.lut ; Démo]

Le noeud operateur en Lutin

```
node operateur(est_stable:bool)
returns(C,Cible:real;phase:int)=
   phase=0 and between(Cible,0.0,100.0) and C=Cible
 fby
 loop {
   loop { phase=1 and -- Attend la stabilite du systeme
     not est_stable and
     maintient(C) and maintient(Cible)
   }
   fby phase=2 and -- Choisi la prochaine consigne
     between(Cible, 0.0, 100.0) and
     maintient(C)
   fby -- Vise la cible pas a pas
     assert phase=3 and maintient(Cible) in
     run C := change_consigne_pas_a_pas(
                        est_stable,pre C,pre Cible,2.0)
     in
       while (C <> Cible)
  }
```

Le noeud change_consigne_pas_a_pas en Lutin

```
node change_consigne_pas_a_pas(
  est_stable : bool; -- calcule par ailleurs
  C, Cible, pas : real)
returns (newC : real) =
 loop {
   loop { not est_stable and newC = C }
   fby
     if Abs(Cible - C) < pas then
       newC = Cible
     else if Cible < C then
       C - pas < newC and newC < C
     else
       C < newC and newC < C + pas
   fby
   -- on attend que le chgt de Consigne prenne effet
   loop [10] { newC = C }
 }
```

Plan

Conclusion

- Une approche basée sur des langages synchrones (versatilité)
  - Tests unitaires, intégration, etc.
  - Démarche itérative (raffinements successifs)
- Des langages pour des spécifications formelles lisibles
- Une mise en œuvre de l'approche « orientée par les modèles » pragmatique où l'on ne jette rien
  - le modèle n'est pas là pour palier à l'explosion de l'analyse du vrai système mais pour faire de la simulation et de la validation précoce (à toutes les étapes)
- Une façon d'introduire (et de tirer partie) des méthodes formelles dans l'industrie sans se heurter au mur de l'explosion combinatoire
- La société Argosim va être créée (juin 2013) dans le but d'industrialiser ces idées/outils

Outline

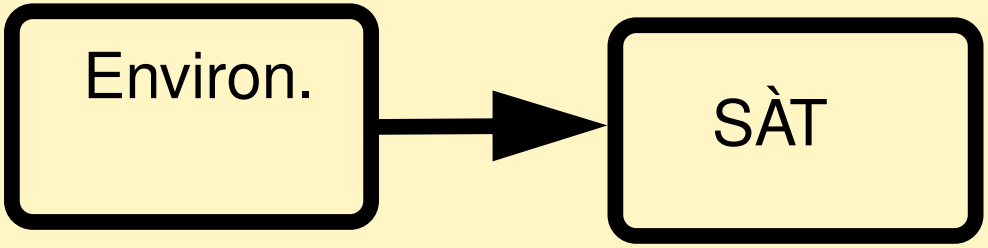
Plan

Lurette - Tests fonctionnels automatisés de systèmes réactifs (synchrones)

- Test fonctionnel (boîte noire)
  - confronter une implémentation et une spécification
- Test Automatisé
  - génération des stimuli (entrées du SàT)
  - dépouillement du résultat des tests (oracle)
- Basé sur une description formelle
  - des attendus du *Système à Tester* (SàT)
  - des hypothèses faites sur son environnement
- Systèmes réactifs
  - le SàT réagit à l'environnement qu'il cherche à contrôler (feedback)
  - un environnement *réaliste* doit faire de même

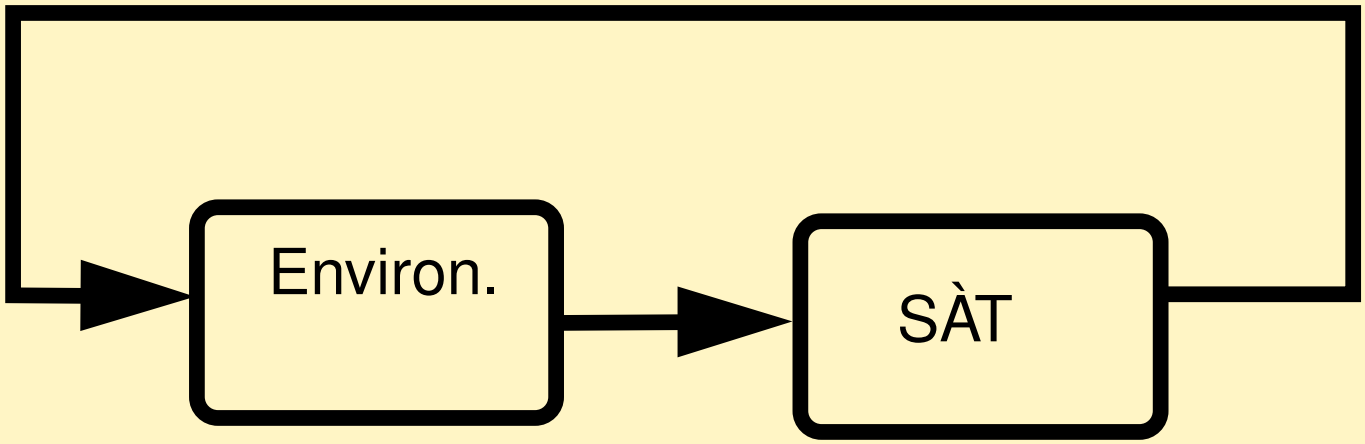
Test de systèmes bouclés

Test de systèmes bouclés



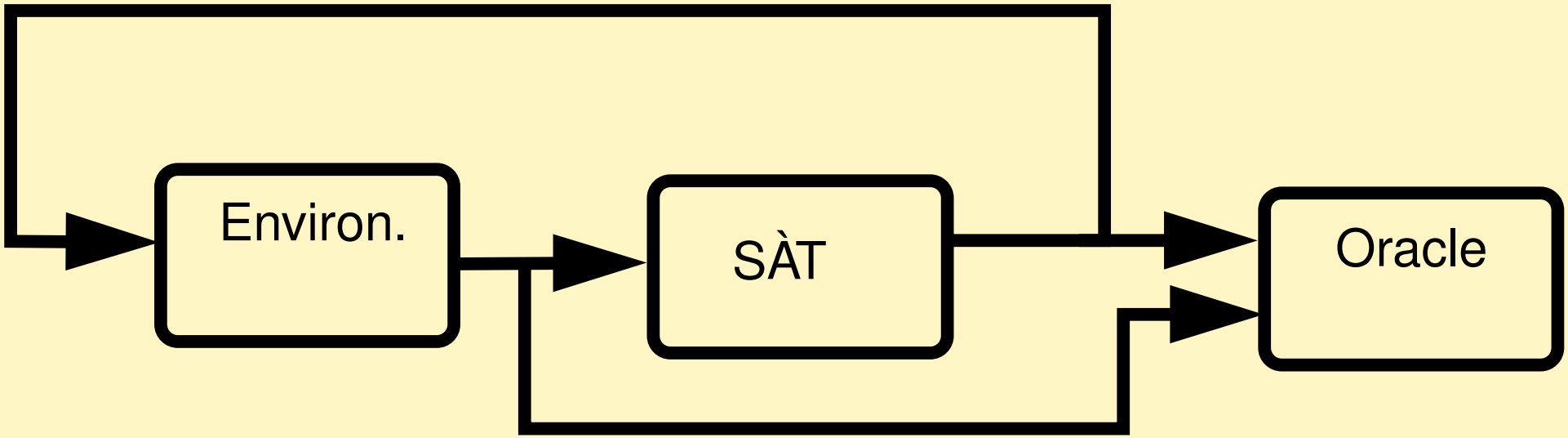
- L'environnement (Stimulateur) peut-être vu comme un système réactif non-déterministe
- Générateur de vecteurs de test sous contrainte

Test de systèmes bouclés



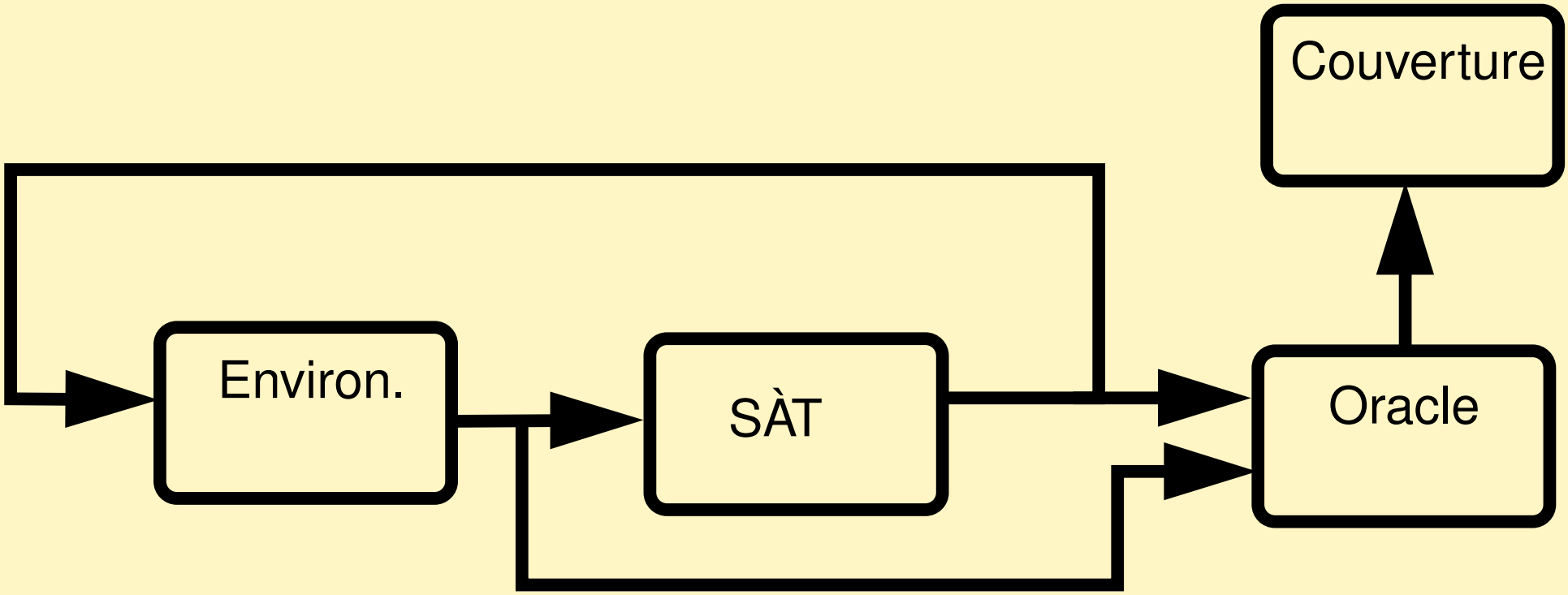
- L'environnement (Stimulateur) peut-être vu comme un système réactif non-déterministe
- Générateur de vecteurs de test sous contrainte

Test de systèmes bouclés



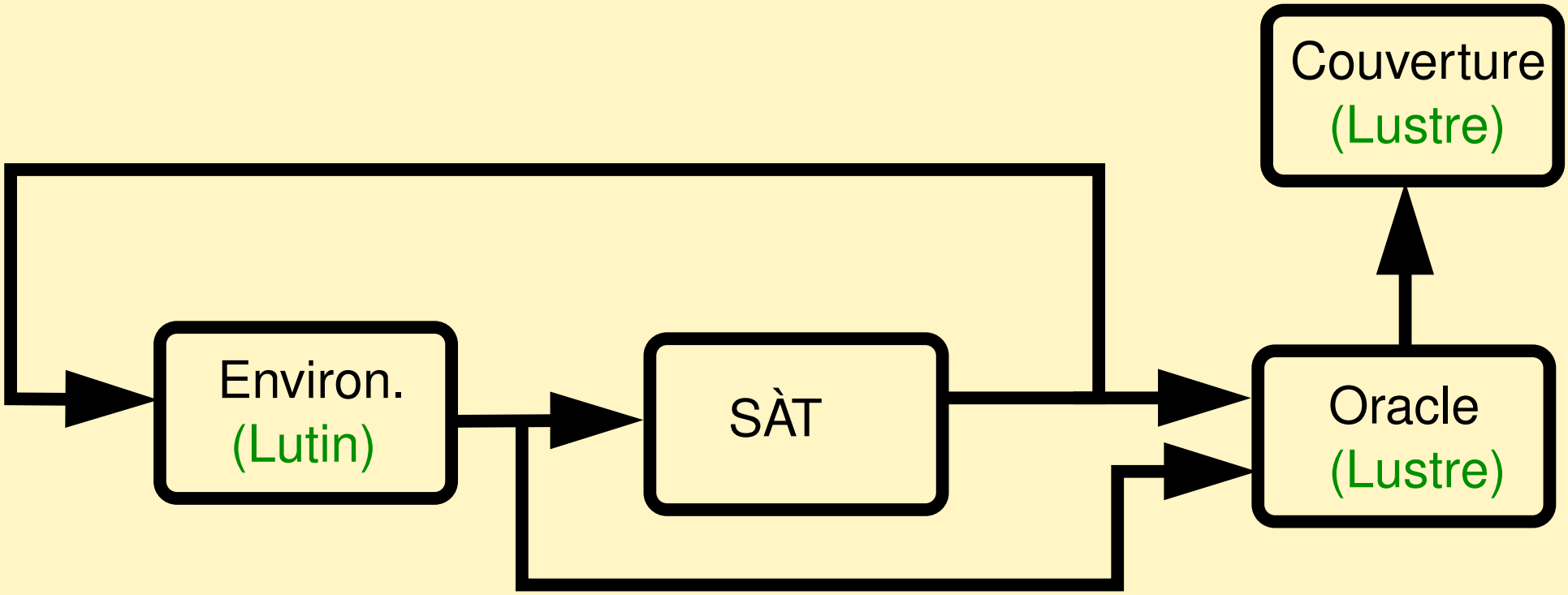
- L'environnement (Stimulateur) peut-être vu comme un système réactif non-déterministe
- Générateur de vecteurs de test sous contrainte

Test de systèmes bouclés



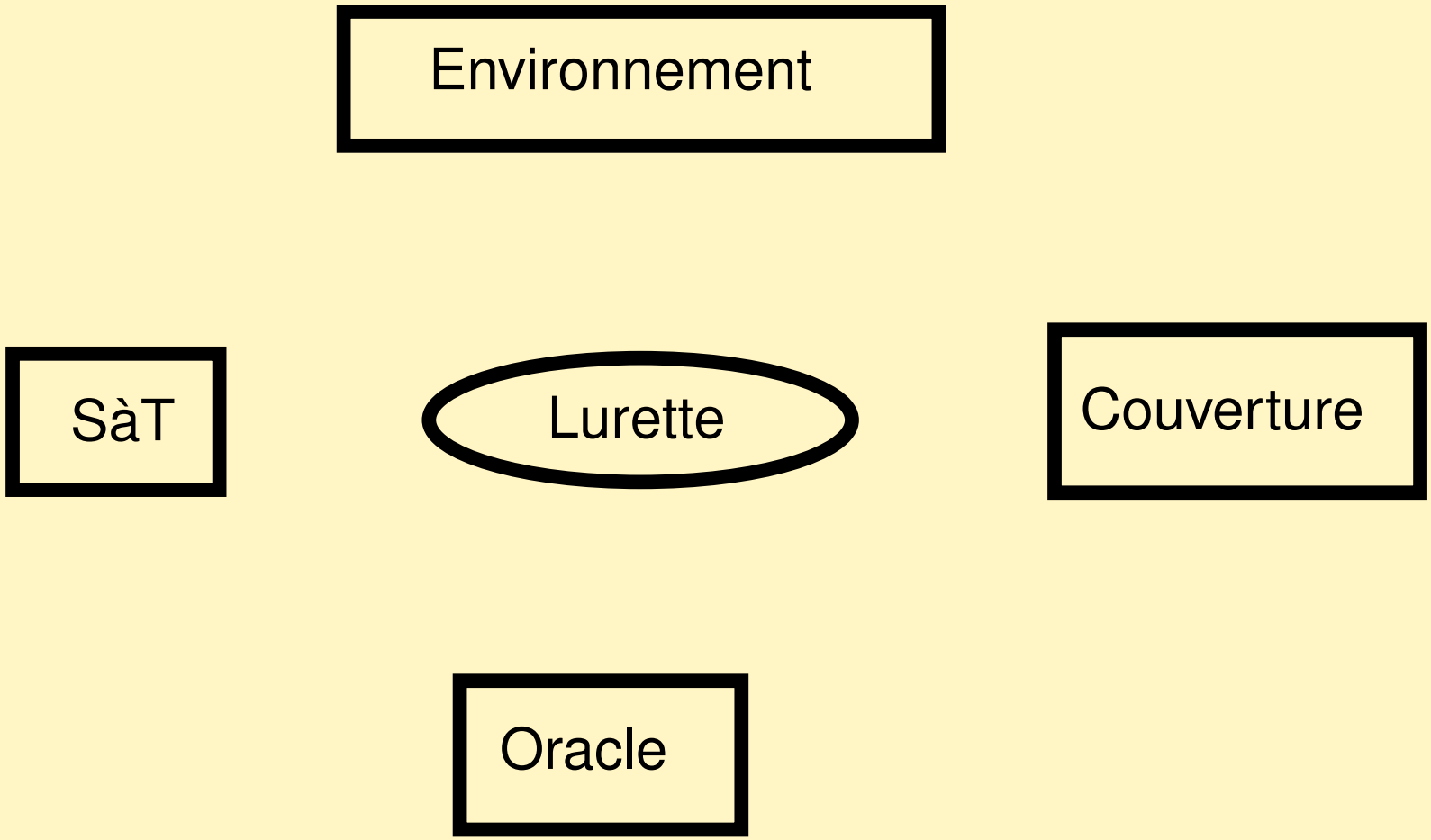
- L'environnement (Stimulateur) peut-être vu comme un système réactif non-déterministe
- Générateur de vecteurs de test sous contrainte

Test de systèmes bouclés

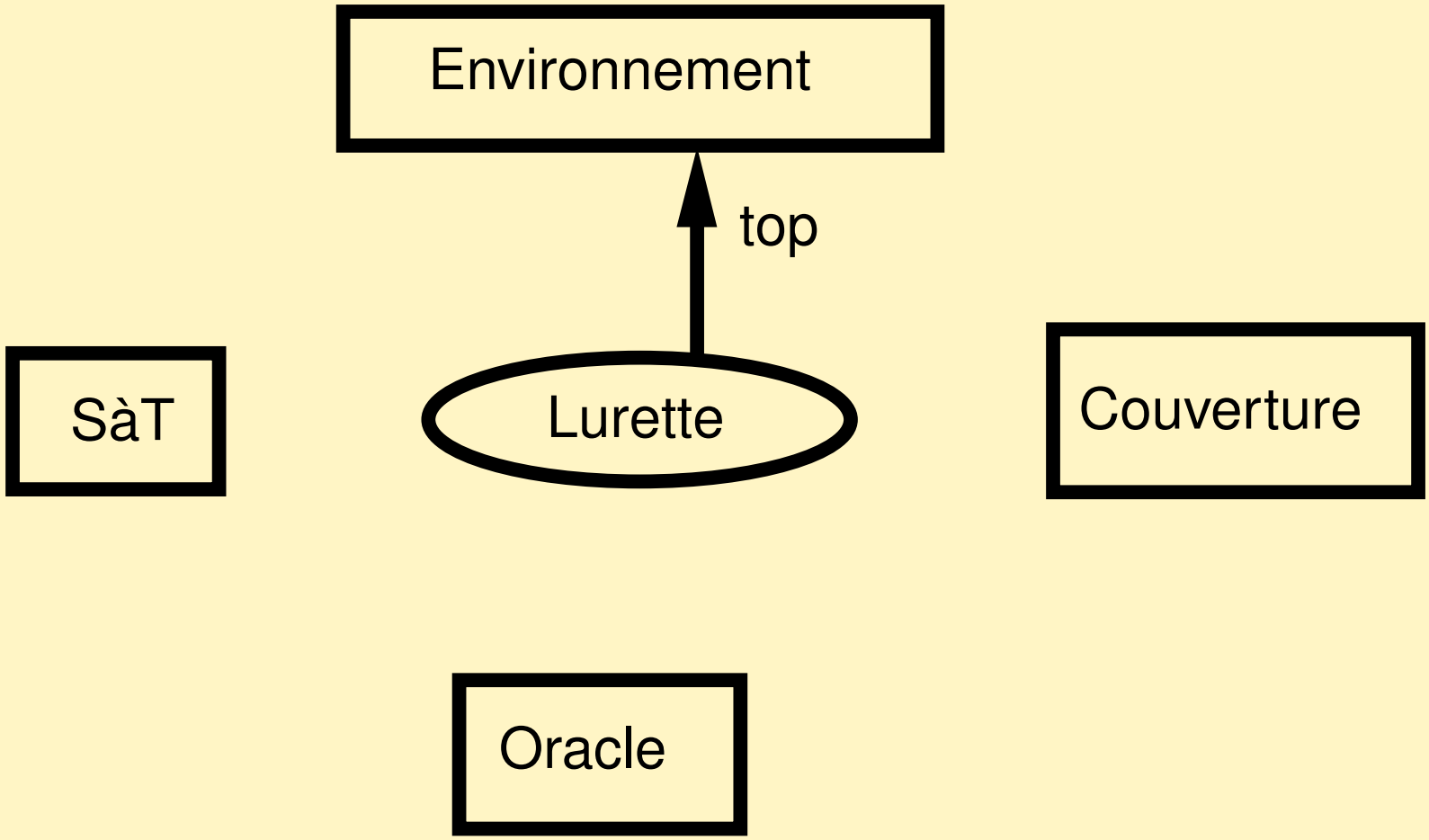


- L'environnement (Stimulateur) peut-être vu comme un système réactif non-déterministe
- Générateur de vecteurs de test sous contrainte

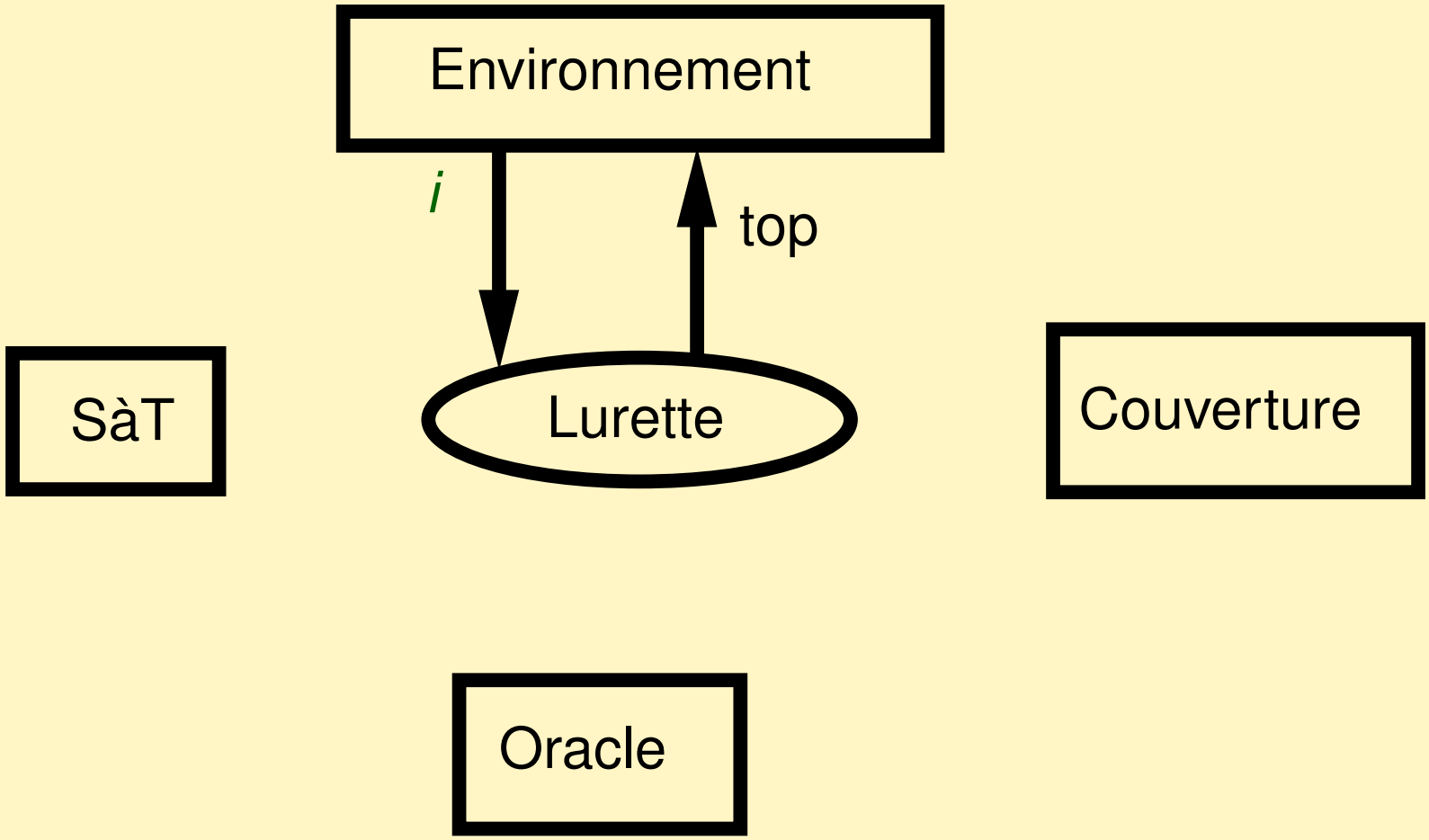
Vue Flot de données à chaque cycle



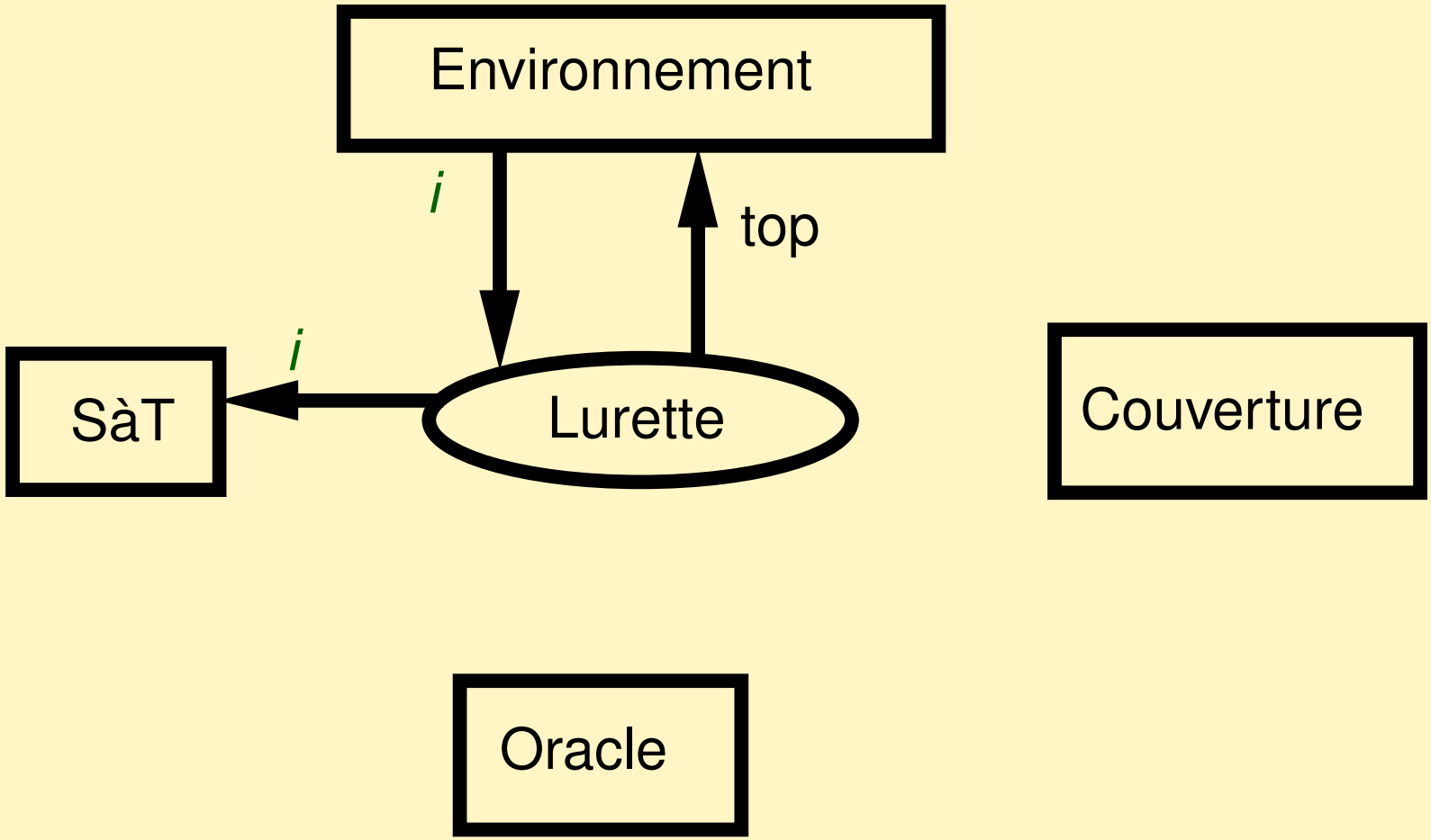
Vue Flot de données à chaque cycle



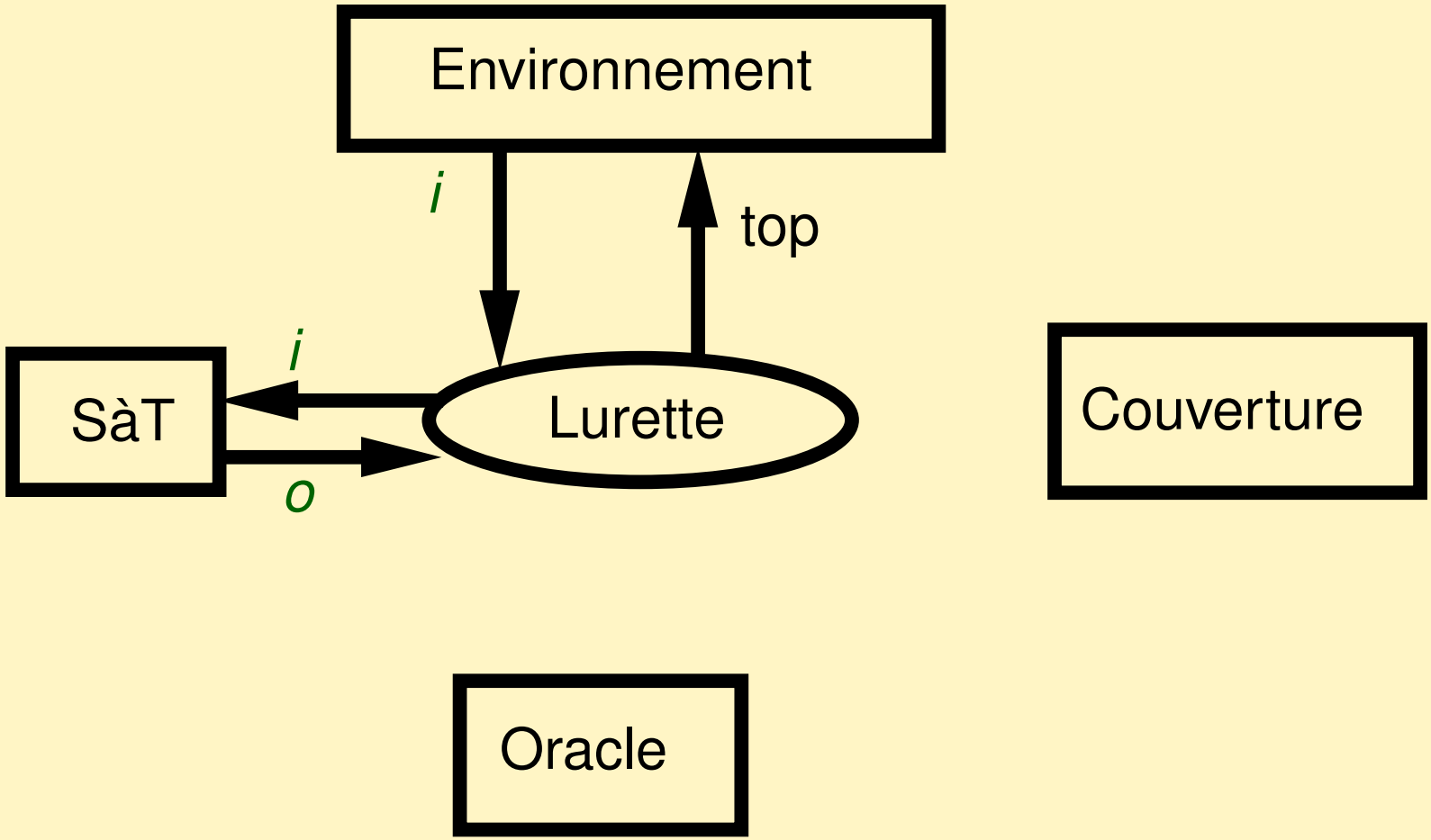
Vue Flot de données à chaque cycle



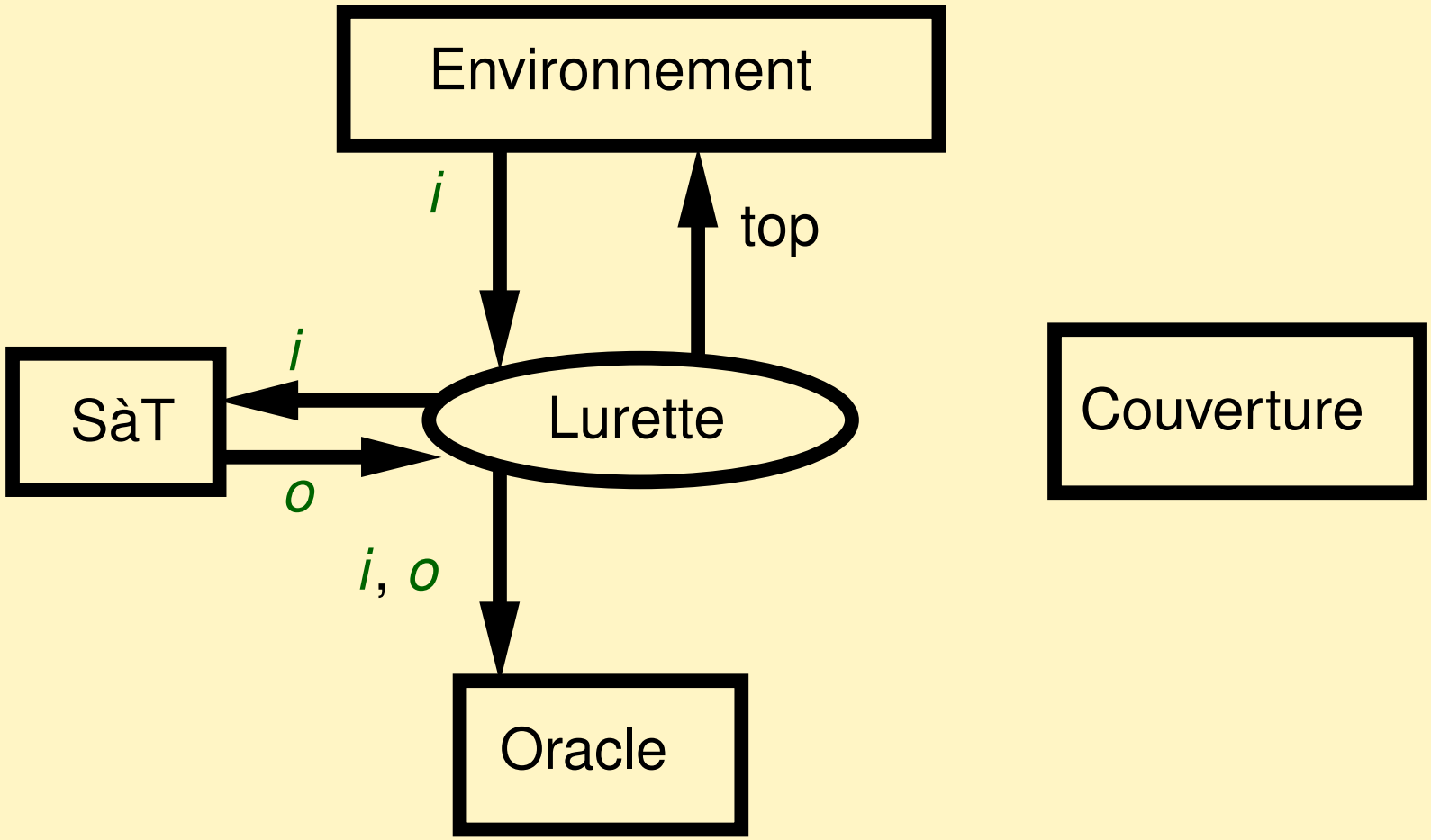
Vue Flot de données à chaque cycle



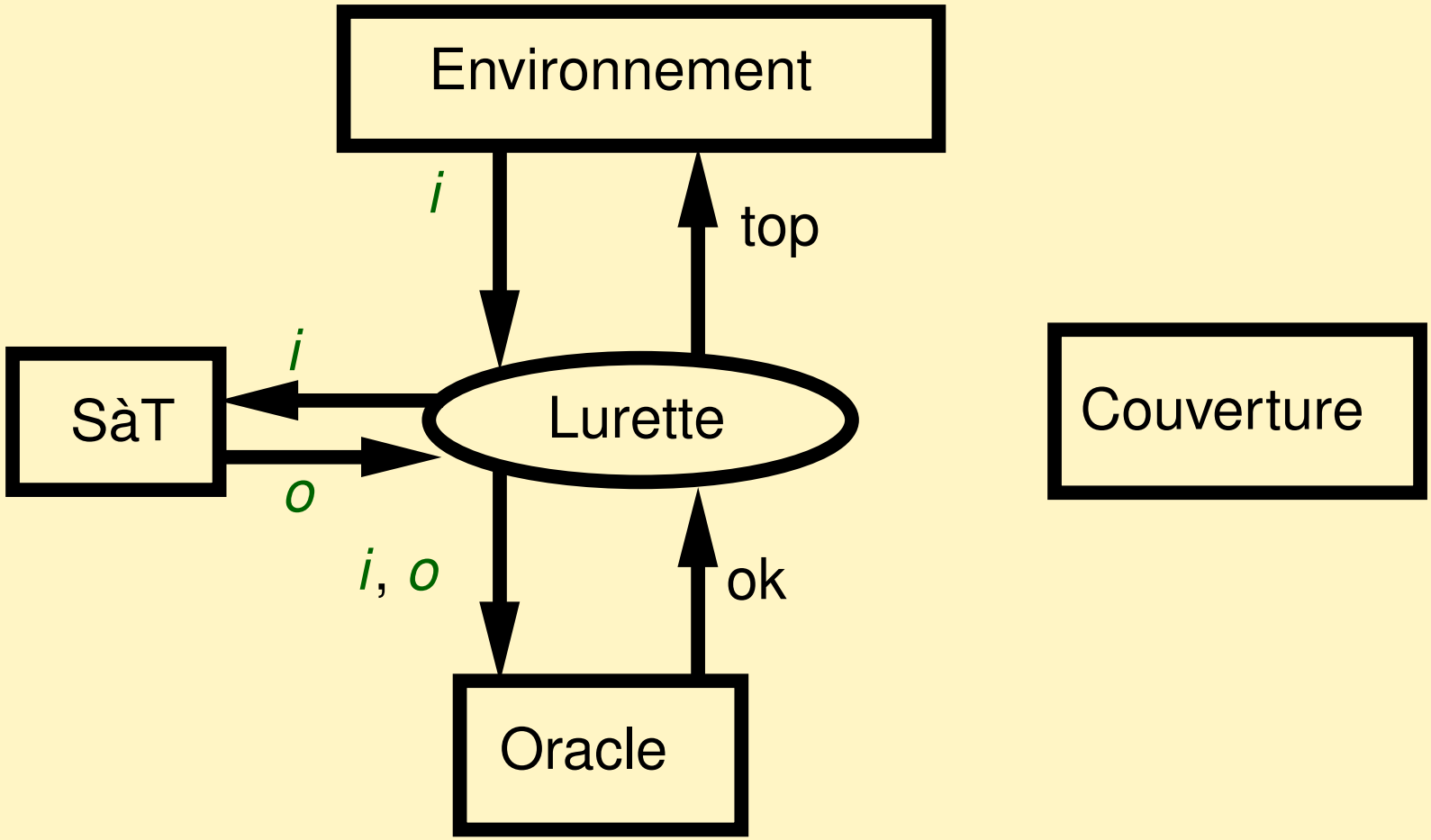
Vue Flot de données à chaque cycle



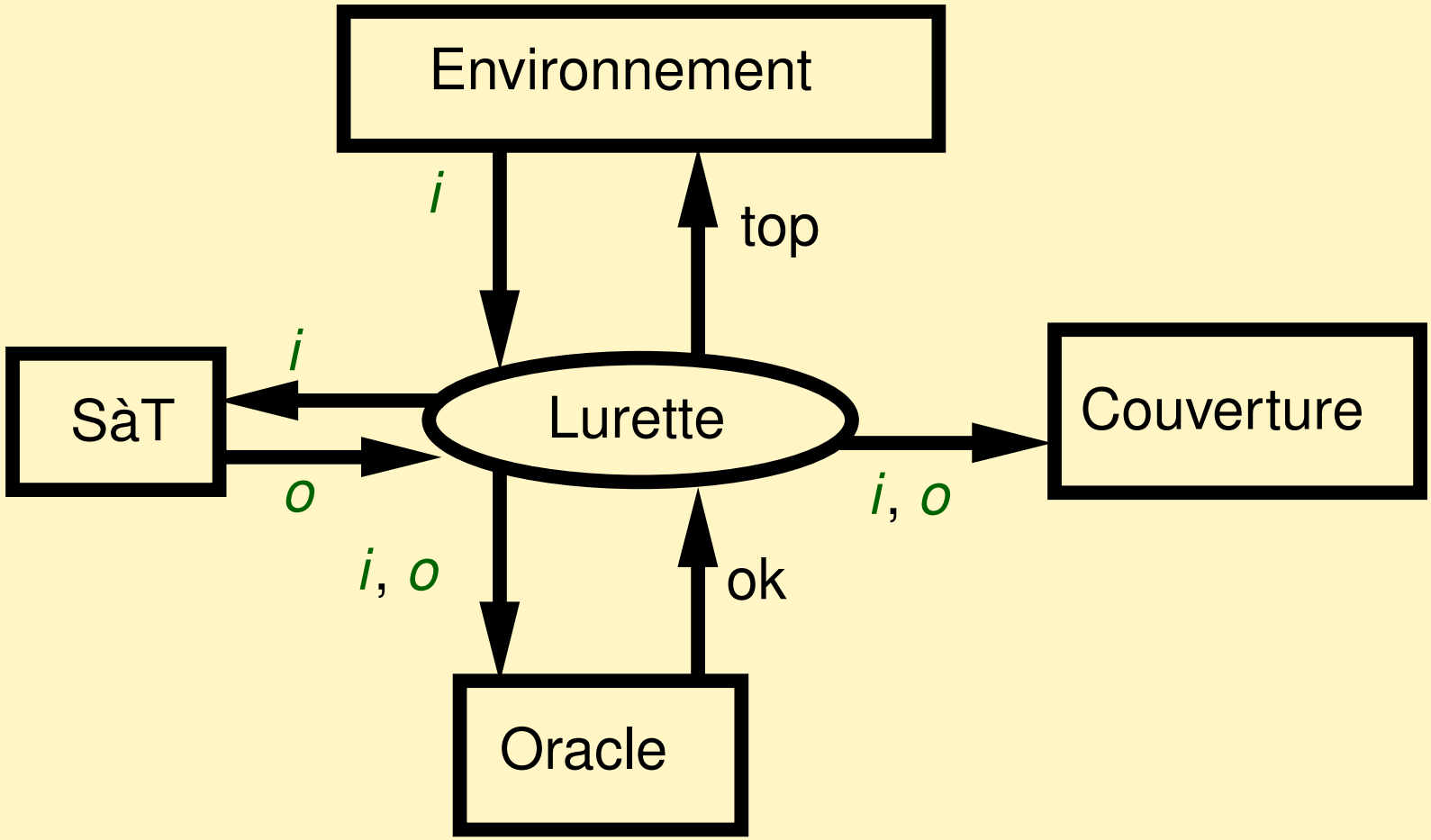
Vue Flot de données à chaque cycle



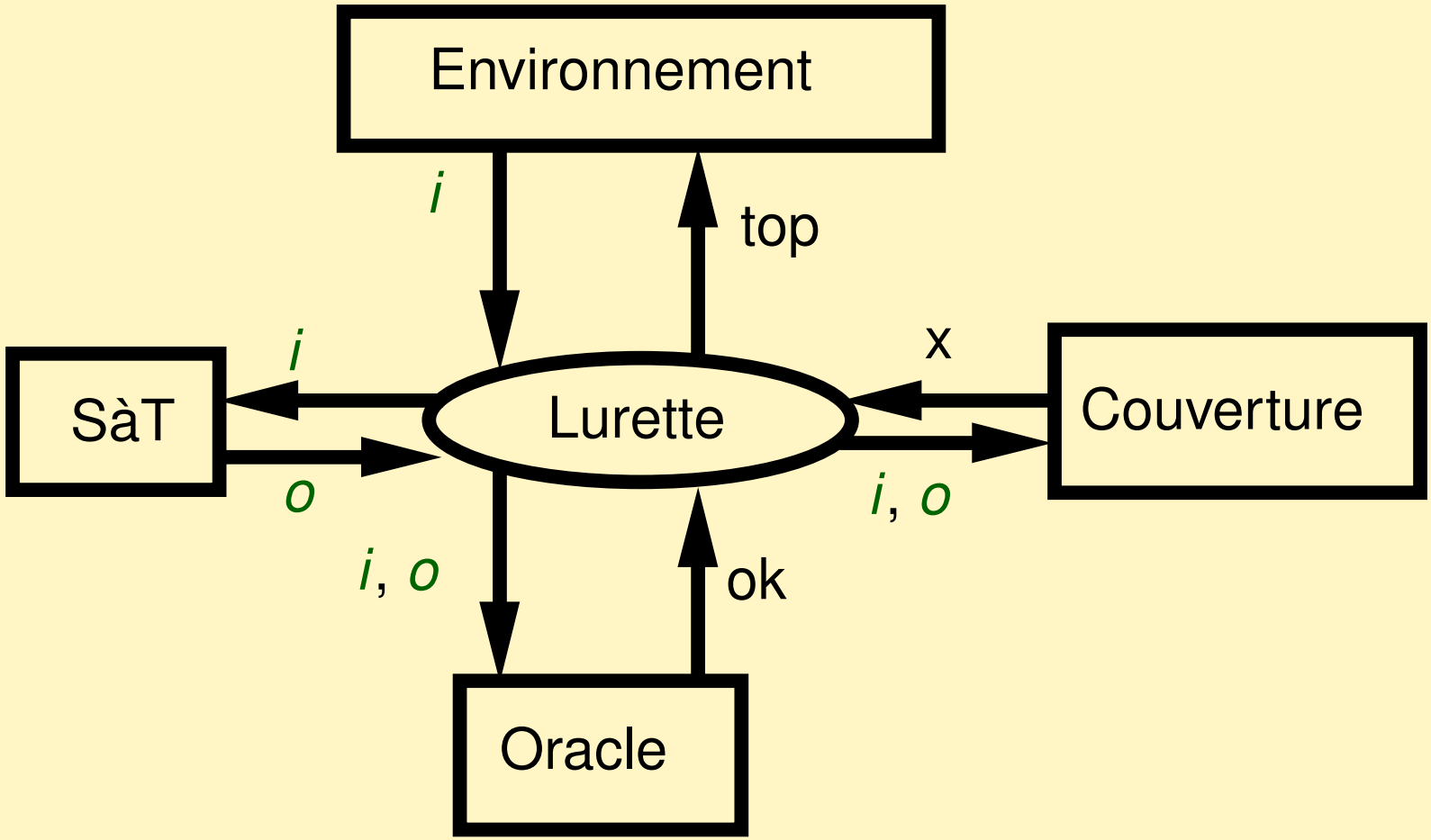
Vue Flot de données à chaque cycle



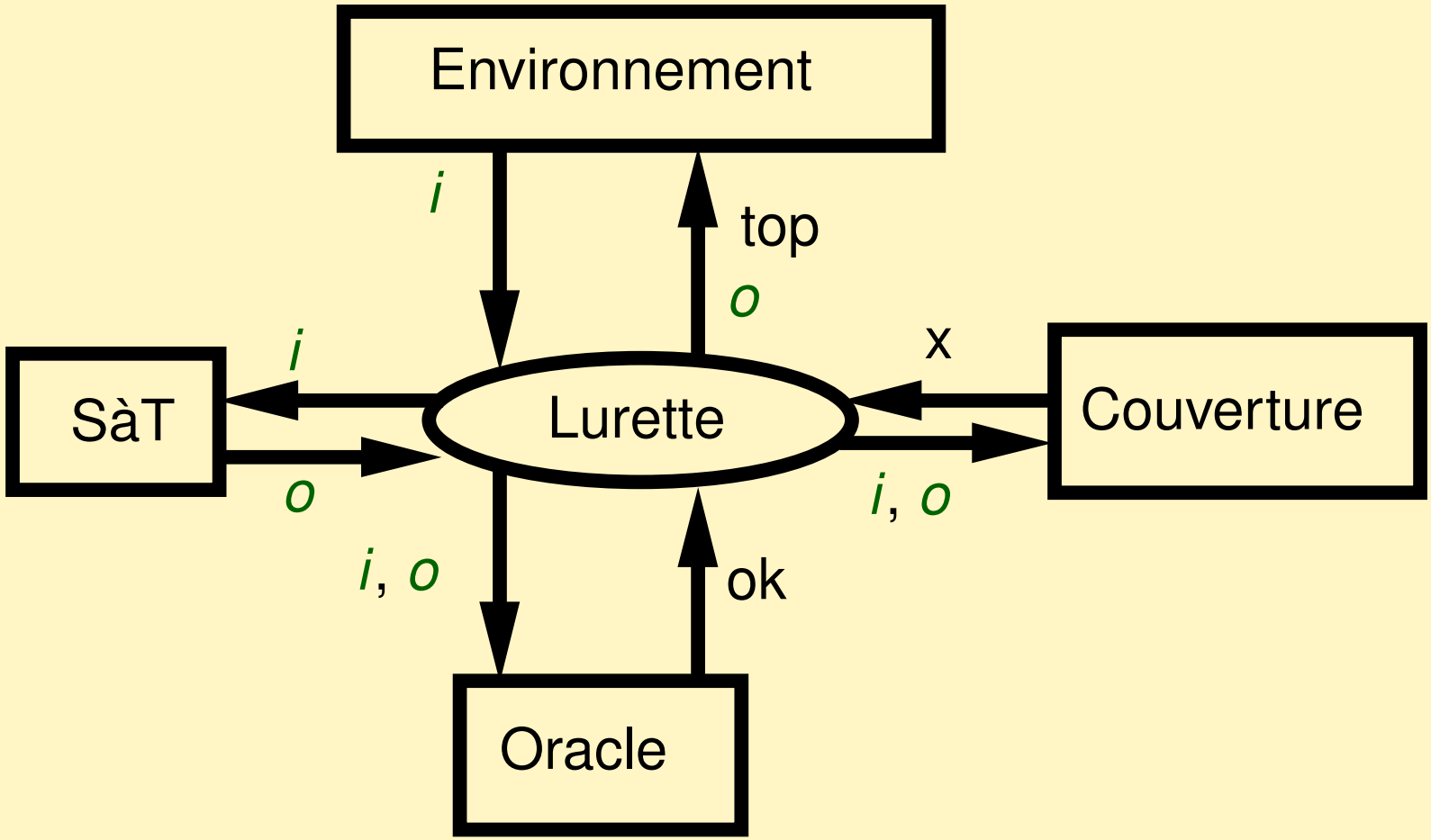
Vue Flot de données à chaque cycle



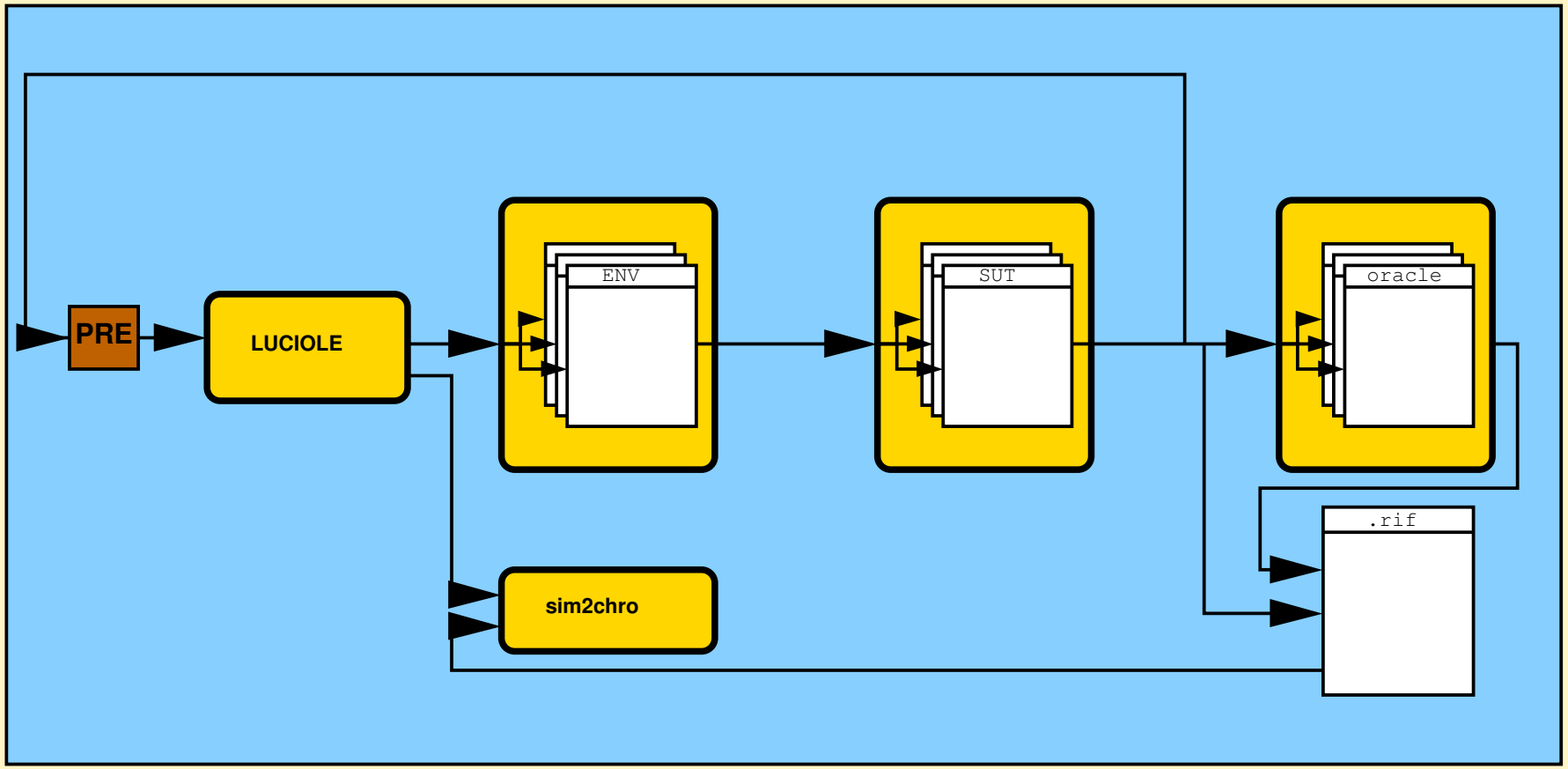
Vue Flot de données à chaque cycle



Vue Flot de données à chaque cycle



Vue synthétique de l'outil



Expérimentations sur des systèmes «Synchrones»

- Hispano-Suiza (Scade)
  - Système de Contrôle-commande d'un moteur (M88)
  - Étude menée par un ingénieur en stage 2ème année (3 mois)
  - 17 bogues dont 13 nouveaux, et 1 dans un système en production
- Renault (Sildex/RT-Builder)
  - Objectif : réduire l'influence des conditions extérieures (pente, vent, etc.) sur le comportement du freinage
  - SUT : système de freinage+modèle simulink du vehicule
  - Env. : Le conducteur + les perturbations extérieures
  - Oracle : comme le SUT, mais sans pertubation
- Astrium (AADL et Scade)
  - ATV/PFS - Un systeme redondé avec un maître et un esclave
  - « Quand l'un des 2 systèmes abondonne son rôle de maître, l'autre le prend en moins de 2 ticks de l'horloge principale »

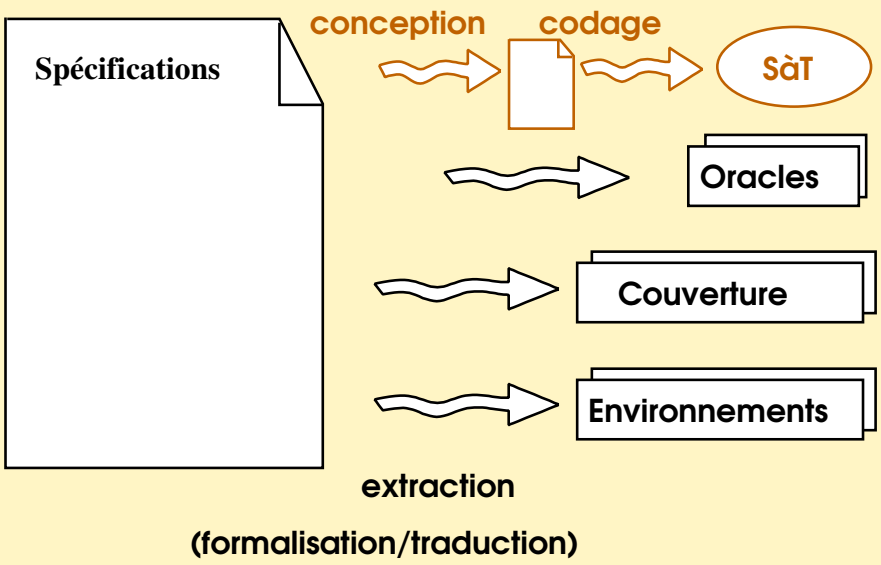
Expérimentations sur des systèmes hétérogènes

- Le projet Minalogic COMON (2009-2012) a été l'occasion d'élargir le domaine d'application des nos outils
  - Corys TESS, Alices (Simulateur énergie et transport)
  - Atos Origin, Scada (télésurveillance et acquisition de données)
  - Rolls-Royce, Scade

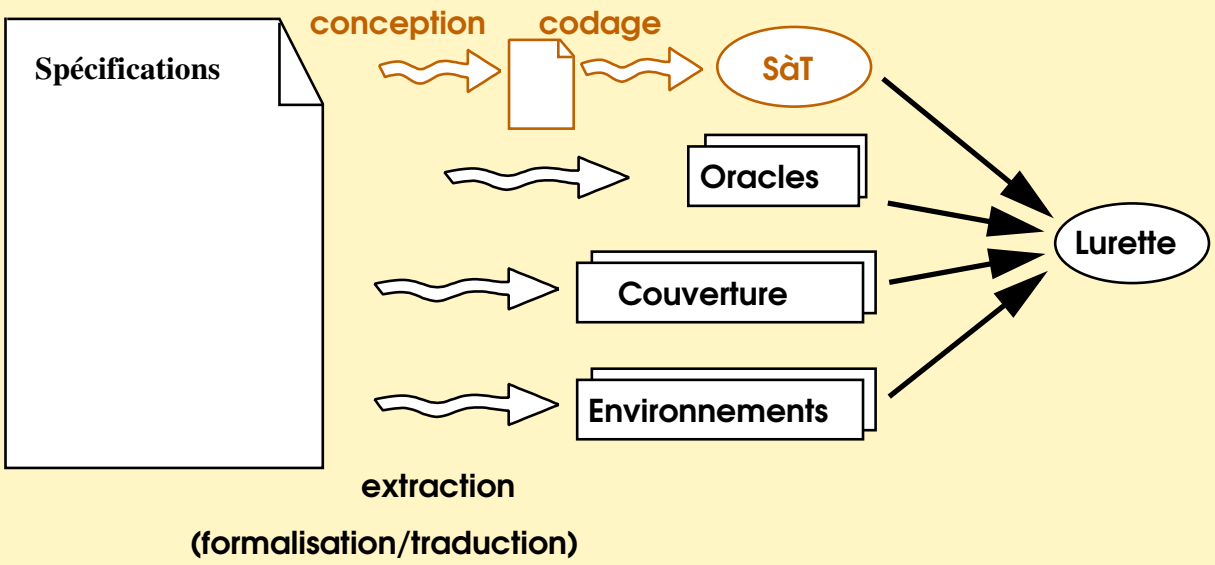
Expérimentations sur des systèmes hétérogènes

- Le projet Minalogic COMON (2009-2012) a été l'occasion d'élargir le domaine d'application des nos outils
  - Corys TESS, Alices (Simulateur énergie et transport)
  - Atos Origin, Scada (télésurveillance et acquisition de données)
  - Rolls-Royce, Scade
- La conclusion des expérimentations menées dans COMON est que Lurette est autant un outil de test qu'un outil d'ingénierie des spécifications (exigences fonctionnelles)

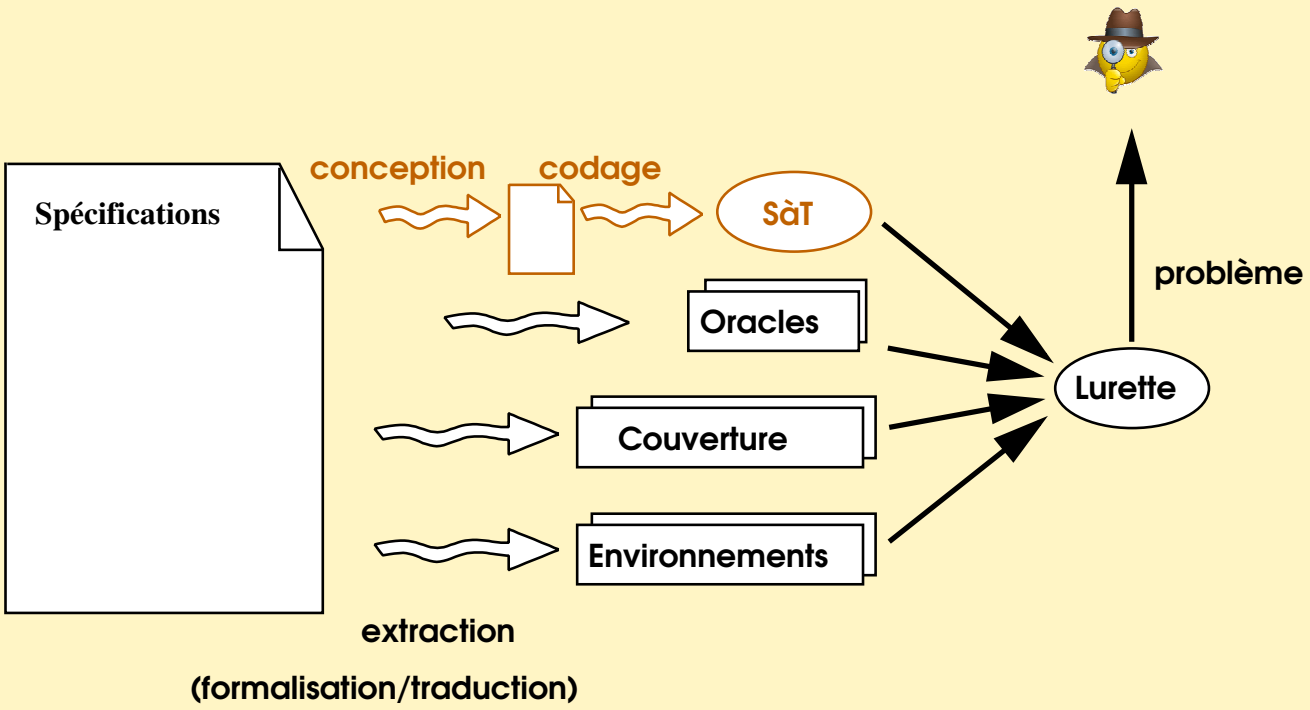
Processus de test Lurette



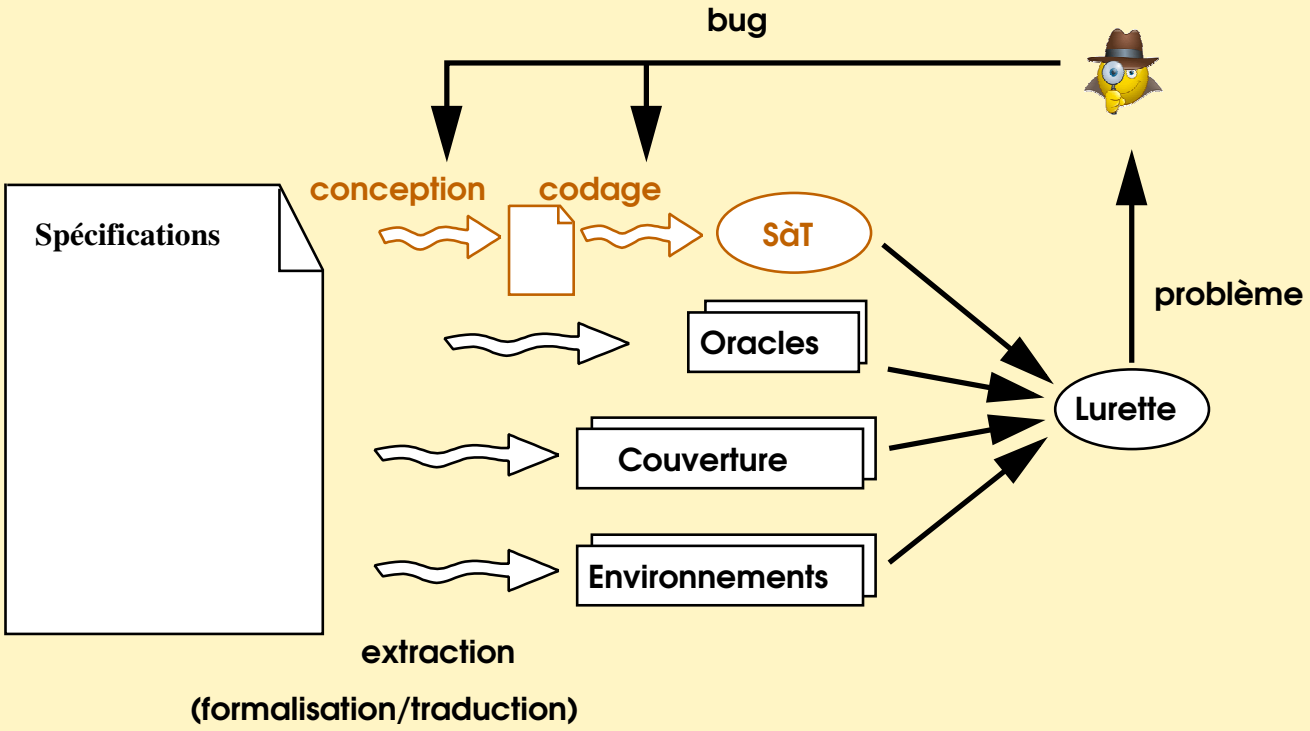
Processus de test Lurette



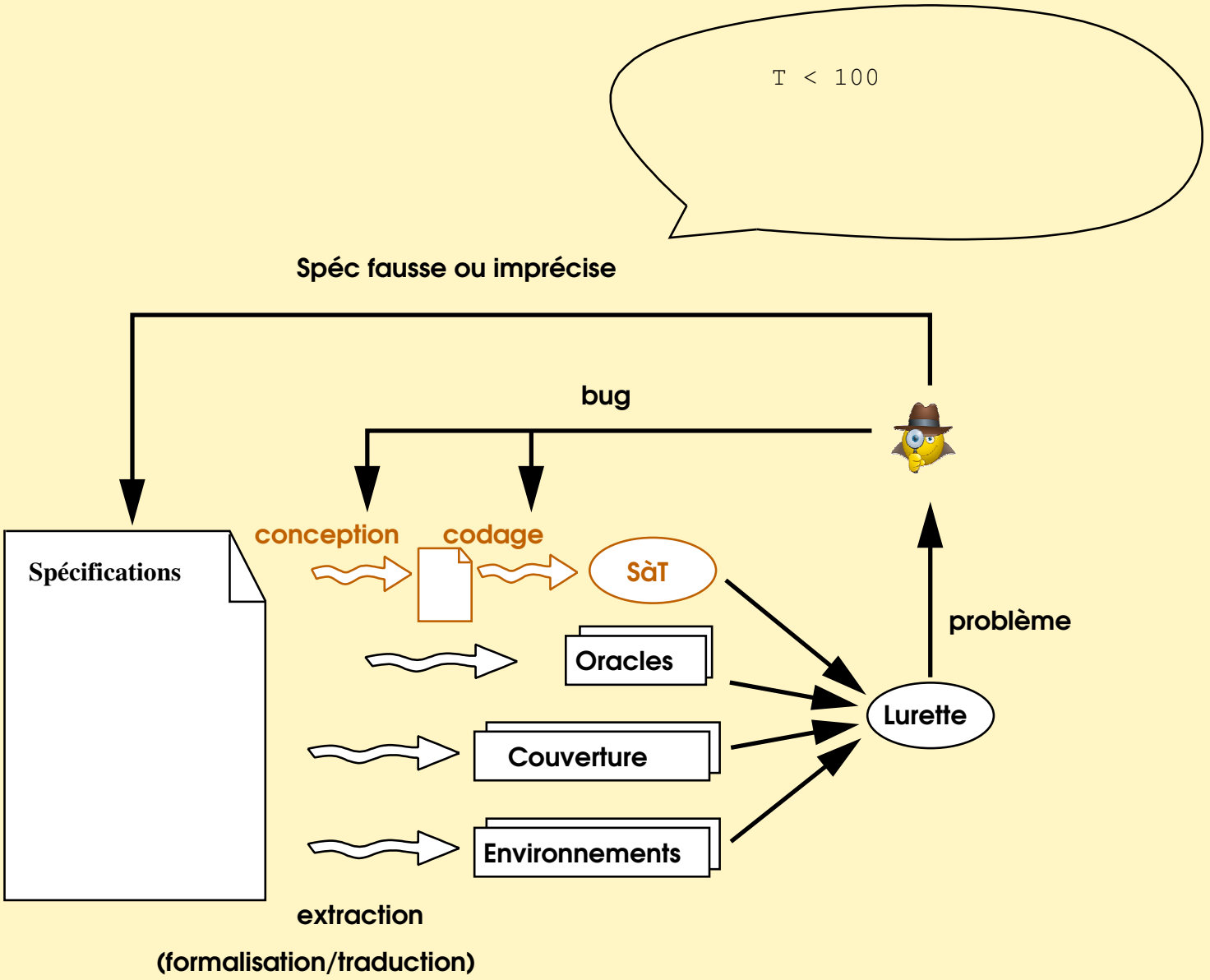
Processus de test Lurette



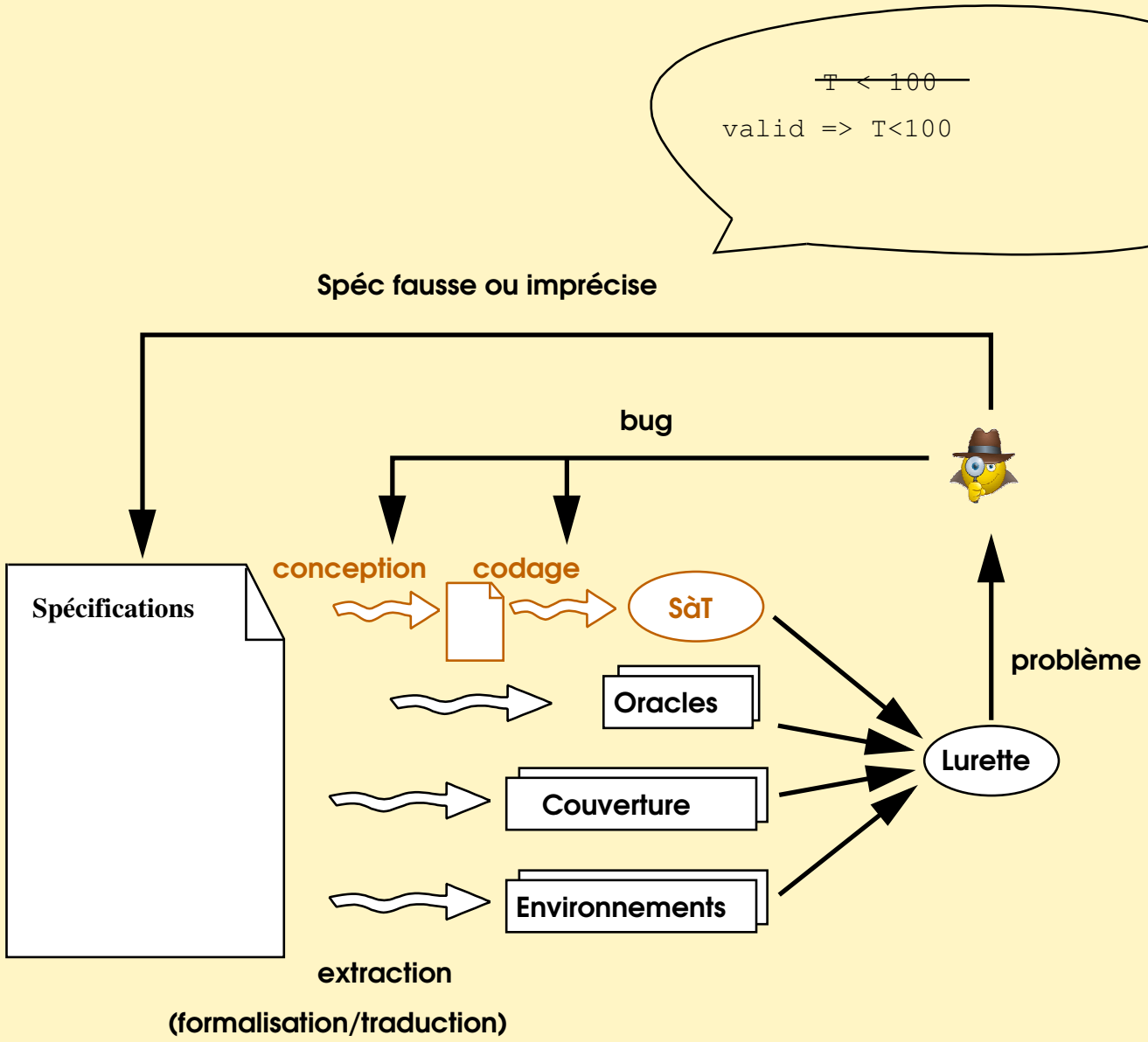
Processus de test Lurette



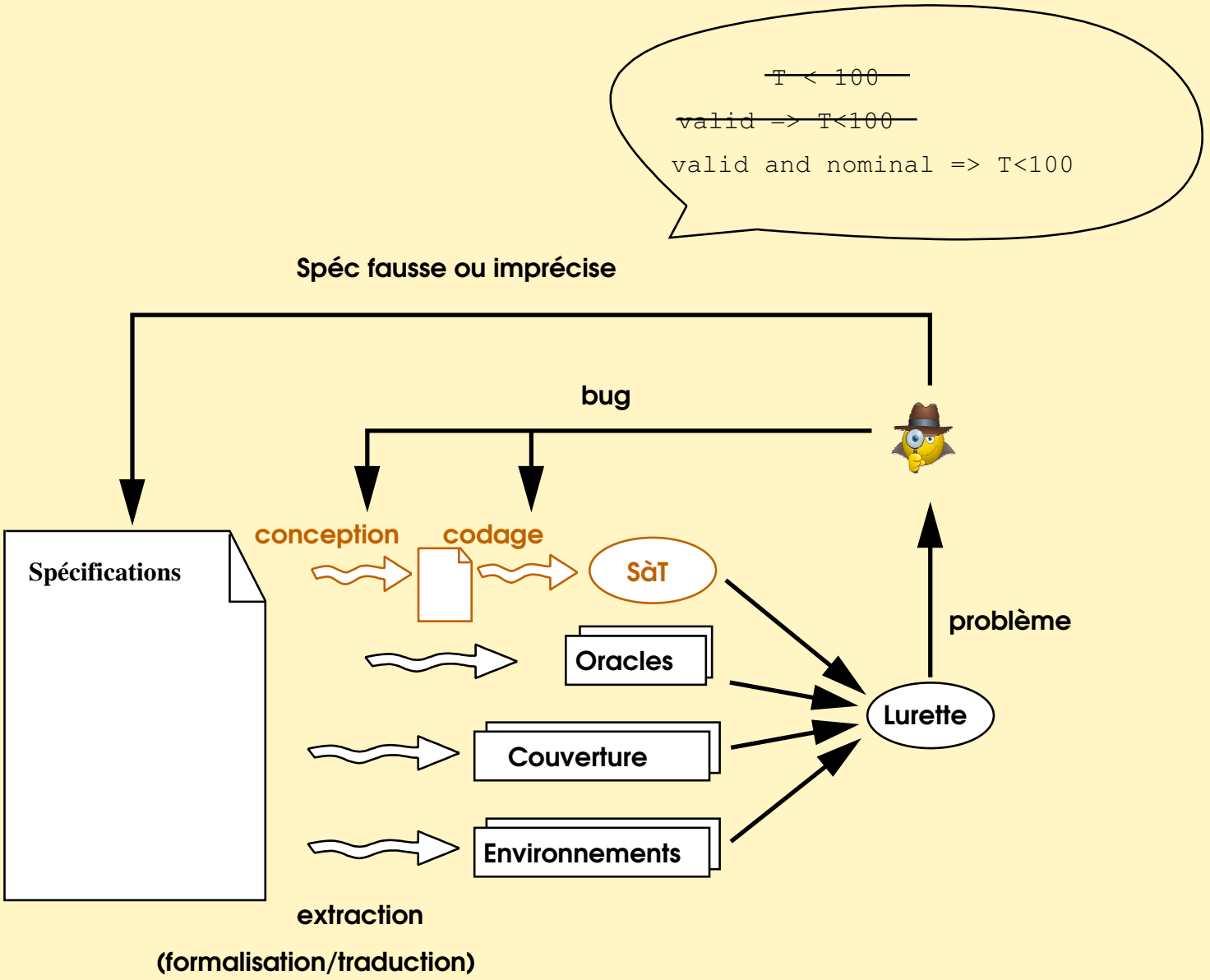
Processus de test Lurette



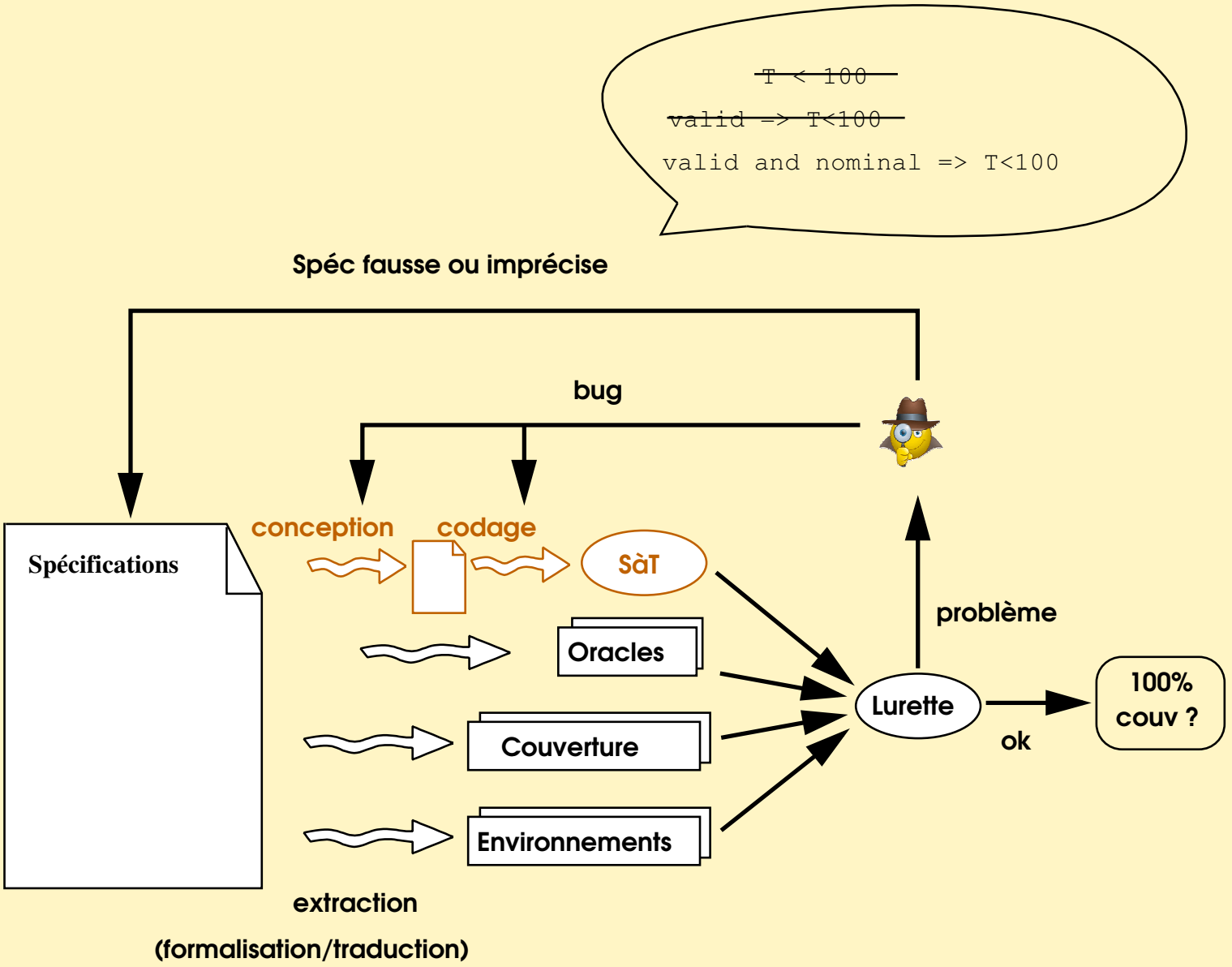
Processus de test Lurette



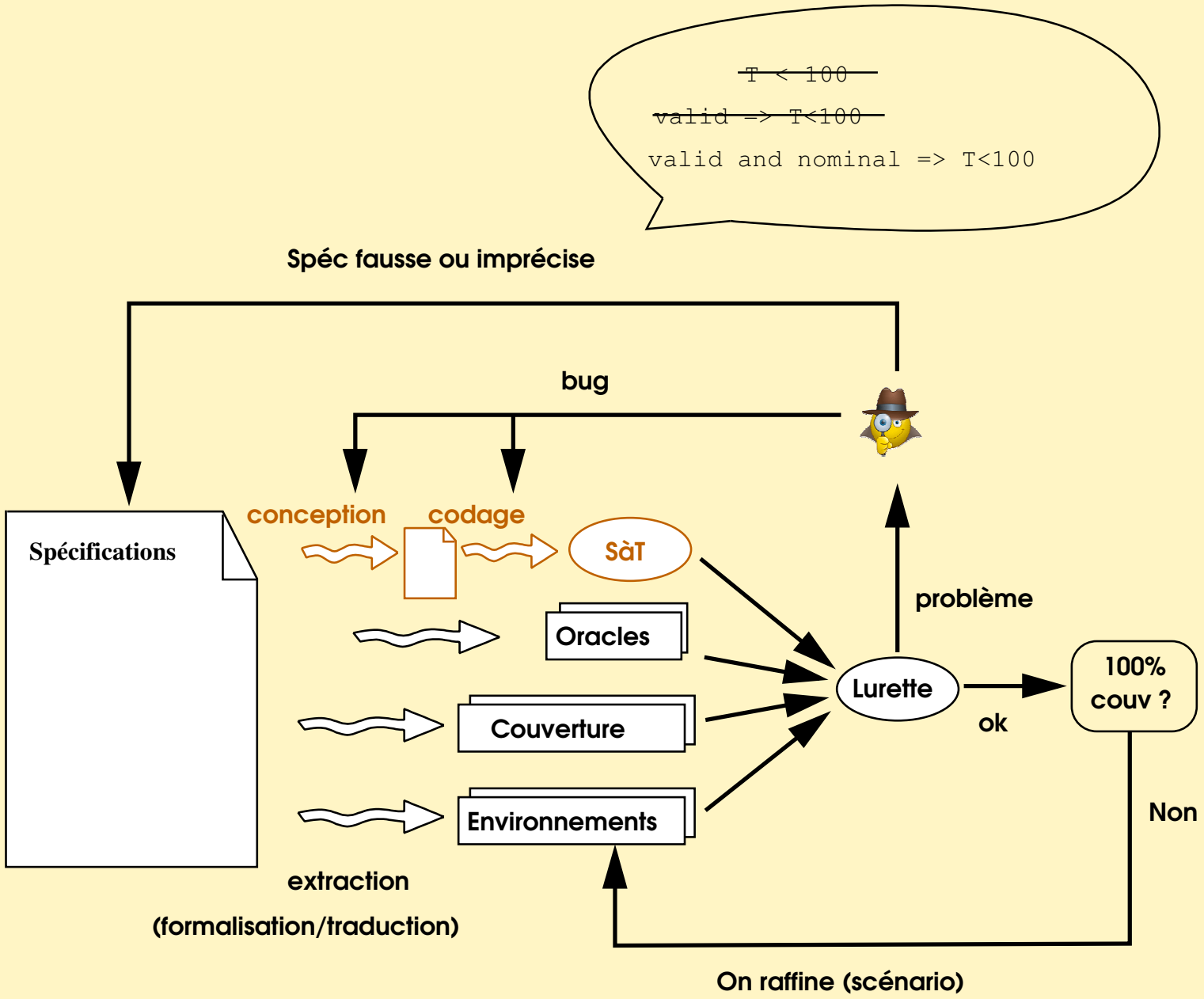
Processus de test Lurette



Processus de test Lurette



Processus de test Lurette



Processus de test Lurette

~~T < 100~~

~~valid => T<100~~

valid and nominal => T<100

Spéc fausse ou imprécise

bug

conception

codage

Spécifications

SàT

Oracles

Couverture

Environnements

problème

Lurette

ok

100% couv ?

Oui

Non

extraction

(formalisation/traduction)

On raffine (scénario)

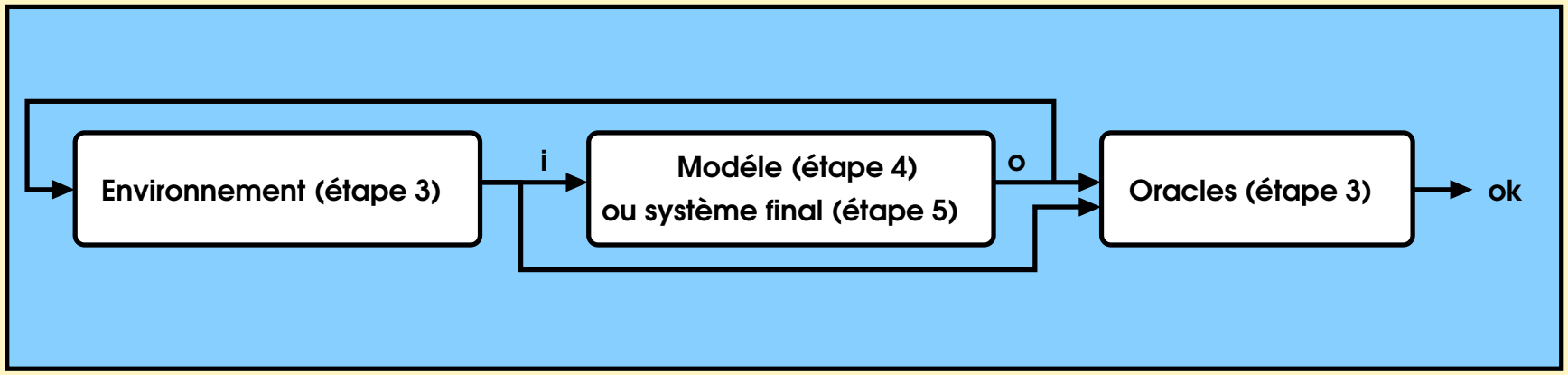
Plan

Test et Développement sans rupture dans COMON

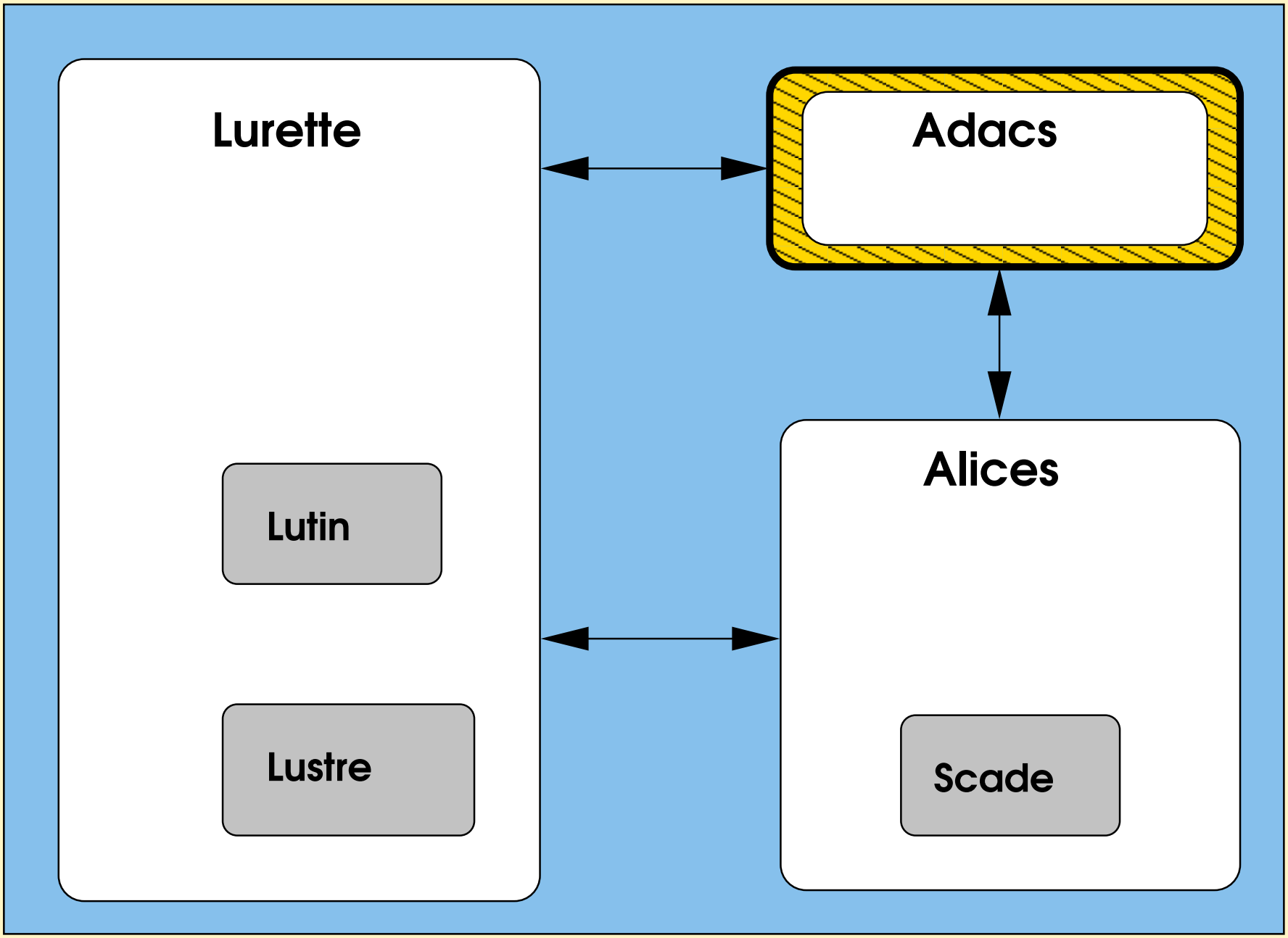
1. Analyse des besoins
2. Spécification des exigences fonctionnelles en langue naturelle et à l'aide de dessins informels
3. Formalisation de ces exigences fonctionnelles (Oracles)
4. Obtention au plus tôt d'un modèle abstrait et exécutable satisfaisant ces exigences (Alices)
5. Obtention d'une implémentation finale satisfaisant ces exigences

Test et Développement sans rupture dans COMON

1. Analyse des besoins
2. Spécification des exigences fonctionnelles en langue naturelle et à l'aide de dessins informels
3. Formalisation de ces exigences fonctionnelles (Oracles)
4. Obtention au plus tôt d'un modèle abstrait et exécutable satisfaisant ces exigences (Alices)
5. Obtention d'une implémentation finale satisfaisant ces exigences



Architecture logicielle du banc test

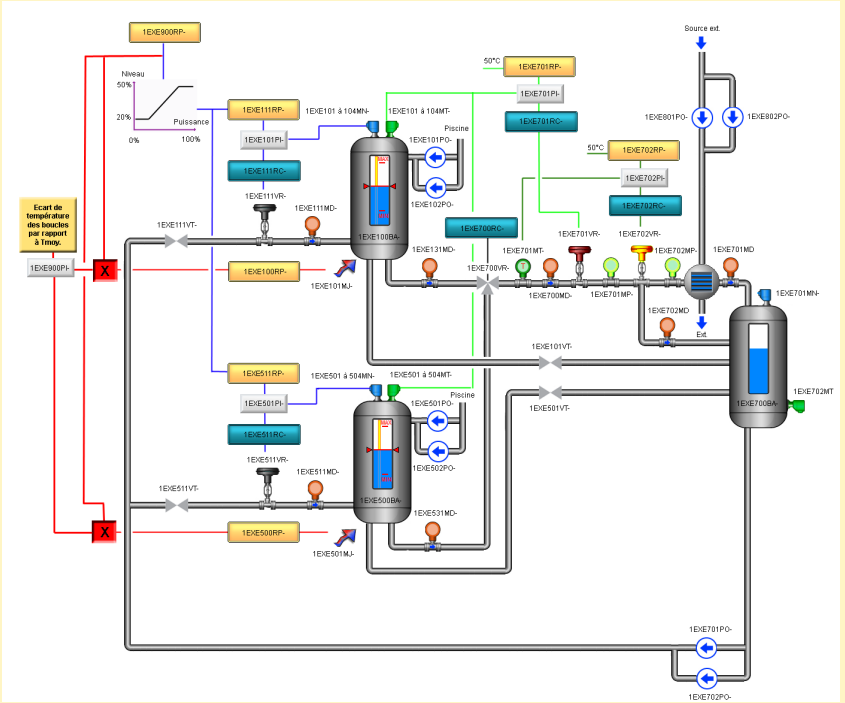


Expérimentations effectuées sur le cas d'étude

- Formalisation
  - Lustre pour les oracles
  - Lutin pour les environnements/stimulateurs
- Bibliothèque d'utilitaires génériques pour écrire les oracles et de stimulateurs
  - Vérifier qu'une valeur numérique est dans sa plage
  - Calculer les états du système
    - nominal, dégradé, 2/3-1/3, urgence
  - Détecter la stabilité d'une variable (`est_stable`)
  - Effectuer une action et attendre la stabilité
  - etc.

Le cas d'étude COMON

- Une étude de cas représentative
  - D'un circuit hydraulique nucléaire
    - Circuit physique
    - Capteurs et actionneurs
  - Et de son contrôle-commande
    - Redondance
    - Régulation
    - Sûreté
  - Son démonstrateur
    - Procédé simulé
    - N1 standard simulé
    - N1 classé émulé
    - N2 réel



Ce que l'on voit dans la Démo

- 4 ateliers hétérogènes communiquer
- Scénarios de stimulation (Environnement)
  - `n` (0, puis 1, puis 2) pannes au hasard, avec comme contrainte d'éviter (si possible) l'action de sûreté
  - Un opérateur virtuel bouclé
    - Une consigne cible est choisie au hasard
    - L'opérateur virtuel change la consigne d'au plus « `pas` » pour se rapprocher de la cible ; il attend la stabilité du sàt avant de la changer la consigne à nouveau (boucle 1)
    - quand la consigne atteint sa cible, on rechoisit une cible (boucle 2)
- Propriétés Invalidées (Oracles)
  - Transitions entre états du système (nominal <-> 2/3-1/3 <-> dégradé <-> urgence) : anomalie dans le calcul de la situation sur le N2
  - Seuil haut dépassé : anomalie dans le voteur N1 qui n'élimine pas le capteur en panne d'une fausse moyenne

Un oracle surveillant la stabilisation du système

En mode nominal, après tout changement de consigne, toutes les valeurs issues des capteurs doivent être stables au bout de 2 minutes

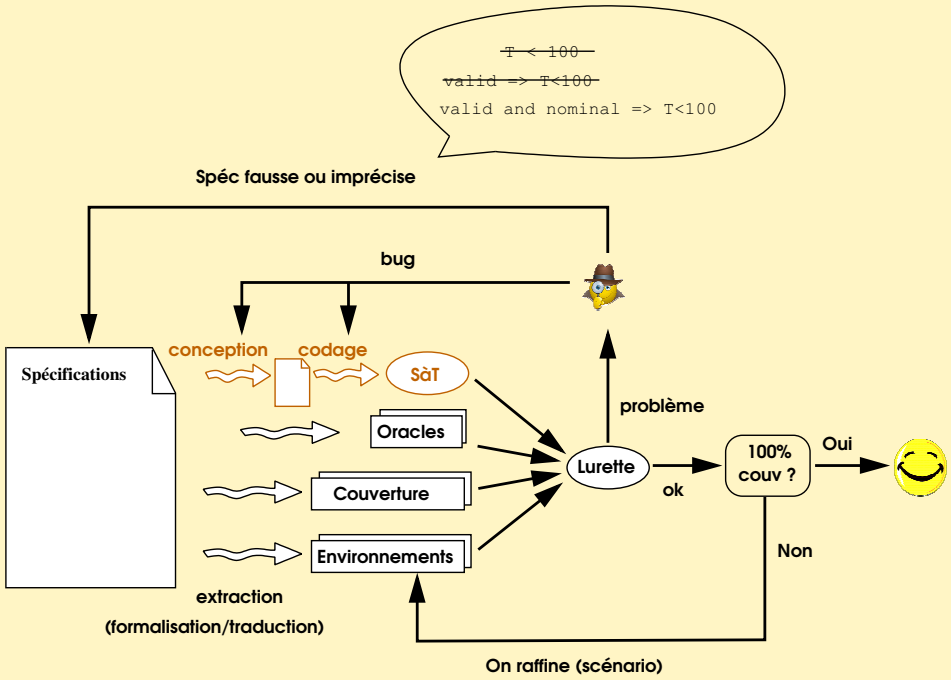
Un oracle surveillant la stabilisation du système

En mode nominal, après tout changement de consigne, toutes les valeurs issues des capteurs doivent être stables au bout de 2 minutes

```
C = vrai_depuis(aucun_chgt_consigne and nominal,120.0) ;
ok = (C => est_stable)
```

- Couvrir ce test, c'est générer une séquence où C est vraie
- D'où un besoin de scénarios où la consigne ne change pas toutes les secondes

Présentation des 2 stimulateurs de la démo



- Génération de pannes : contraintes qui génèrent plein de cas
- Modélisation d'un opérateur : illustre la capacité de Lutin à exprimer des scénarios assez évolués (couverture)

Un générateur de pannes (pseudo-)aléatoire

- Objectif : générer `n` pannes sans déclencher d'action classée
  - 2/2 ou 3/4 capteurs rédondés en défaut

Un générateur de pannes (pseudo-)aléatoire

- Objectif : générer `n` pannes sans déclencher d'action classée
  - 2/2 ou 3/4 capteurs rédondés en défaut
- On tire au hasard parmi les solutions des contraintes

Un générateur de pannes (pseudo-)aléatoire

- Objectif : générer `n` pannes sans déclencher d'action classée
  - 2/2 ou 3/4 capteurs rédondés en défaut
- On tire au hasard parmi les solutions des contraintes
- Un scénario minimal pour
  - ne pas changer de panne à chaque cycle
  - en changer tout de même au bout de X cycles (ou si reset)

Un générateur de pannes (pseudo-)aléatoire

- Objectif : générer `n` pannes sans déclencher d'action classée
  - 2/2 ou 3/4 capteurs rédondés en défaut
- On tire au hasard parmi les solutions des contraintes
- Un scénario minimal pour
  - ne pas changer de panne à chaque cycle
  - en changer tout de même au bout de X cycles (ou si reset)

```
node(n:int; reset:bool) returns(P1,P2,P3,P4,P5,P6) =
let SOS=deux_vrai(P1,P2) or trois_vrai(P3,P4,P5,P6) in
```

Un générateur de pannes (pseudo-)aléatoire

- Objectif : générer `n` pannes sans déclencher d'action classée
  - 2/2 ou 3/4 capteurs rédondés en défaut
- On tire au hasard parmi les solutions des contraintes
- Un scénario minimal pour
  - ne pas changer de panne à chaque cycle
  - en changer tout de même au bout de X cycles (ou si reset)

```
node(n:int; reset:bool) returns(P1,P2,P3,P4,P5,P6) =
let SOS=deux_vrai(P1,P2) or trois_vrai(P3,P4,P5,P6) in
let b2i(b:bool) = if b then 1 else 0 in
```

Un générateur de pannes (pseudo-)aléatoire

- Objectif : générer `n` pannes sans déclencher d'action classée
  - 2/2 ou 3/4 capteurs rédondés en défaut
- On tire au hasard parmi les solutions des contraintes
- Un scénario minimal pour
  - ne pas changer de panne à chaque cycle
  - en changer tout de même au bout de X cycles (ou si reset)

```
node(n:int; reset:bool) returns(P1,P2,P3,P4,P5,P6) =
let SOS=deux_vrai(P1,P2) or trois_vrai(P3,P4,P5,P6) in
let b2i(b:bool) = if b then 1 else 0 in
let nb_pannes = b2i(P1)+...+b2i(P6) in
```

Un générateur de pannes (pseudo-)aléatoire

- Objectif : générer `n` pannes sans déclencher d'action classée
  - 2/2 ou 3/4 capteurs rédondés en défaut
- On tire au hasard parmi les solutions des contraintes
- Un scénario minimal pour
  - ne pas changer de panne à chaque cycle
  - en changer tout de même au bout de X cycles (ou si reset)

```
node(n:int; reset:bool) returns(P1,P2,P3,P4,P5,P6) =
let SOS=deux_vrai(P1,P2) or trois_vrai(P3,P4,P5,P6) in
let b2i(b:bool) = if b then 1 else 0 in
let nb_pannes = b2i(P1)+...+b2i(P6) in
loop {
```

Un générateur de pannes (pseudo-)aléatoire

- Objectif : générer `n` pannes sans déclencher d'action classée
  - 2/2 ou 3/4 capteurs rédondés en défaut
- On tire au hasard parmi les solutions des contraintes
- Un scénario minimal pour
  - ne pas changer de panne à chaque cycle
  - en changer tout de même au bout de X cycles (ou si reset)

```
node(n:int; reset:bool) returns(P1,P2,P3,P4,P5,P6) =
let SOS=deux_vrai(P1,P2) or trois_vrai(P3,P4,P5,P6) in
let b2i(b:bool) = if b then 1 else 0 in
let nb_pannes = b2i(P1)+...+b2i(P6) in
loop {
 { (not SOS and nb_pannes = n) |> nb_pannes = n }
```

Un générateur de pannes (pseudo-)aléatoire

- Objectif : générer `n` pannes sans déclencher d'action classée
  - 2/2 ou 3/4 capteurs rédondés en défaut
- On tire au hasard parmi les solutions des contraintes
- Un scénario minimal pour
  - ne pas changer de panne à chaque cycle
  - en changer tout de même au bout de X cycles (ou si reset)

```
node(n:int; reset:bool) returns(P1,P2,P3,P4,P5,P6) =
let SOS=deux_vrai(P1,P2) or trois_vrai(P3,P4,P5,P6) in
let b2i(b:bool) = if b then 1 else 0 in
let nb_pannes = b2i(P1)+...+b2i(P6) in
loop {
 { (not SOS and nb_pannes = n) |> nb_pannes = n }
 fby loop X {not reset and P1=pre P1 and ... P6=pre P6}
}
```

Un générateur de pannes (pseudo-)aléatoire

- Objectif : générer `n` pannes sans déclencher d'action classée
  - 2/2 ou 3/4 capteurs rédondés en défaut
- On tire au hasard parmi les solutions des contraintes
- Un scénario minimal pour
  - ne pas changer de panne à chaque cycle
  - en changer tout de même au bout de X cycles (ou si reset)

```
node(n:int; reset:bool) returns(P1,P2,P3,P4,P5,P6) =
let SOS=deux_vrai(P1,P2) or trois_vrai(P3,P4,P5,P6) in
let b2i(b:bool) = if b then 1 else 0 in
let nb_pannes = b2i(P1)+...+b2i(P6) in
loop {
 { (not SOS and nb_pannes = n) |> nb_pannes = n }
 fby loop X {not reset and P1=pre P1 and ... P6=pre P6}
}
```

[Pannes-i.lut ; Démo]

Un opérateur (pseudo-)aléatoire scénarisé

Change la consigne en surveillant la valeur de capteurs de niveaux

1. Choisit une consigne Cible (dans [0 ; 100])

Un opérateur (pseudo-)aléatoire scénarisé

Change la consigne en surveillant la valeur de capteurs de niveaux

1. Choisit une consigne Cible (dans [0 ; 100])
2. Utilise un nœud qui va amener la consigne à la cible pas à pas

Un opérateur (pseudo-)aléatoire scénarisé

Change la consigne en surveillant la valeur de capteurs de niveaux

1. Choisit une consigne Cible (dans [0 ; 100])
2. Utilise un nœud qui va amener la consigne à la cible pas à pas
3. Quand la cible est atteinte, Recommence en 1.

Un opérateur (pseudo-)aléatoire scénarisé

Change la consigne en surveillant la valeur de capteurs de niveaux

1. Choisit une consigne Cible (dans [0 ; 100])
2. Utilise un nœud qui va amener la consigne à la cible pas à pas
3. Quand la cible est atteinte, Recommence en 1.

Le nœud change_consigne_pas_a_pas est du même style

1. Maintient la consigne tant qu'on n'a pas la stabilité

Un opérateur (pseudo-)aléatoire scénarisé

Change la consigne en surveillant la valeur de capteurs de niveaux

1. Choisit une consigne Cible (dans [0 ; 100])
2. Utilise un nœud qui va amener la consigne à la cible pas à pas
3. Quand la cible est atteinte, Recommence en 1.

Le nœud change_consigne_pas_a_pas est du même style

1. Maintient la consigne tant qu'on n'a pas la stabilité
2. Se rapproche de la consigne cible d'au plus `pas`
3. Si la consigne cible n'est pas atteinte, Recommence en 1

Cette modélisation illustre la capacité de Lutin à exprimer des scénarios assez évolués (couverture)
[operateur.lut ; Démo]

Le noeud `operateur` en Lutin

```
node operateur(est_stable:bool)
returns(C,Cible:real;phase:int)=
```

Le noeud operateur en Lutin

```
node operateur(est_stable:bool)
returns(C,Cible:real;phase:int)=
   phase=0 and between(Cible,0.0,100.0) and C=Cible
```

Le noeud operateur en Lutin

```
node operateur(est_stable:bool)
returns(C,Cible:real;phase:int)=
   phase=0 and between(Cible,0.0,100.0) and C=Cible
 fby
 loop {
```

Le noeud operateur en Lutin

```
node operateur(est_stable:bool)
returns(C,Cible:real;phase:int)=
   phase=0 and between(Cible,0.0,100.0) and C=Cible
 fby
 loop {
   loop { phase=1 and -- Attend la stabilite du systeme
     not est_stable and
     maintient(C) and maintient(Cible)
   }
```

Le noeud operateur en Lutin

```
node operateur(est_stable:bool)
returns(C,Cible:real;phase:int)=
   phase=0 and between(Cible,0.0,100.0) and C=Cible
 fby
 loop {
   loop { phase=1 and -- Attend la stabilite du systeme
     not est_stable and
     maintient(C) and maintient(Cible)
   }
   fby phase=2 and -- Choisi la prochaine consigne
     between(Cible, 0.0, 100.0) and
     maintient(C)
```

Le noeud operateur en Lutin

```
node operateur(est_stable:bool)
returns(C,Cible:real;phase:int)=
   phase=0 and between(Cible,0.0,100.0) and C=Cible
 fby
 loop {
   loop { phase=1 and -- Attend la stabilite du systeme
     not est_stable and
     maintient(C) and maintient(Cible)
   }
   fby phase=2 and -- Choisi la prochaine consigne
     between(Cible, 0.0, 100.0) and
     maintient(C)
   fby -- Vise la cible pas a pas
     assert phase=3 and maintient(Cible) in
     run C := change_consigne_pas_a_pas(
                        est_stable,pre C,pre Cible,2.0)
     in
       while (C <> Cible)
  }
```

Le noeud change_consigne_pas_a_pas en Lutin

```
node change_consigne_pas_a_pas(
  est_stable : bool; -- calcule par ailleurs
  C, Cible, pas : real)
returns (newC : real) =
 loop {
```

Le noeud change_consigne_pas_a_pas en Lutin

```
node change_consigne_pas_a_pas(
  est_stable : bool; -- calcule par ailleurs
  C, Cible, pas : real)
returns (newC : real) =
 loop {
   loop { not est_stable and newC = C }
```

Le noeud change_consigne_pas_a_pas en Lutin

```
node change_consigne_pas_a_pas(
  est_stable : bool; -- calcule par ailleurs
  C, Cible, pas : real)
returns (newC : real) =
 loop {
   loop { not est_stable and newC = C }
   fby
     if Abs(Cible - C) < pas then
       newC = Cible
     else if Cible < C then
       C - pas < newC and newC < C
     else
       C < newC and newC < C + pas
```

Le noeud change_consigne_pas_a_pas en Lutin

```
node change_consigne_pas_a_pas(
  est_stable : bool; -- calcule par ailleurs
  C, Cible, pas : real)
returns (newC : real) =
 loop {
   loop { not est_stable and newC = C }
   fby
     if Abs(Cible - C) < pas then
       newC = Cible
     else if Cible < C then
       C - pas < newC and newC < C
     else
       C < newC and newC < C + pas
   fby
   -- on attend que le chgt de Consigne prenne effet
   loop [10] { newC = C }
 }
```

Plan

Conclusion

- Une approche basée sur des langages synchrones (versatilité)
  - Tests unitaires, intégration, etc.
  - Démarche itérative (raffinements successifs)

Conclusion

- Une approche basée sur des langages synchrones (versatilité)
  - Tests unitaires, intégration, etc.
  - Démarche itérative (raffinements successifs)
- Des langages pour des spécifications formelles lisibles

Conclusion

- Une approche basée sur des langages synchrones (versatilité)
  - Tests unitaires, intégration, etc.
  - Démarche itérative (raffinements successifs)
- Des langages pour des spécifications formelles lisibles
- Une mise en œuvre de l'approche « orientée par les modèles » pragmatique où l'on ne jette rien
  - le modèle n'est pas là pour palier à l'explosion de l'analyse du vrai système mais pour faire de la simulation et de la validation précoce (à toutes les étapes)

Conclusion

- Une approche basée sur des langages synchrones (versatilité)
  - Tests unitaires, intégration, etc.
  - Démarche itérative (raffinements successifs)
- Des langages pour des spécifications formelles lisibles
- Une mise en œuvre de l'approche « orientée par les modèles » pragmatique où l'on ne jette rien
  - le modèle n'est pas là pour palier à l'explosion de l'analyse du vrai système mais pour faire de la simulation et de la validation précoce (à toutes les étapes)
- Une façon d'introduire (et de tirer partie) des méthodes formelles dans l'industrie sans se heurter au mur de l'explosion combinatoire

Conclusion

- Une approche basée sur des langages synchrones (versatilité)
  - Tests unitaires, intégration, etc.
  - Démarche itérative (raffinements successifs)
- Des langages pour des spécifications formelles lisibles
- Une mise en œuvre de l'approche « orientée par les modèles » pragmatique où l'on ne jette rien
  - le modèle n'est pas là pour palier à l'explosion de l'analyse du vrai système mais pour faire de la simulation et de la validation précoce (à toutes les étapes)
- Une façon d'introduire (et de tirer partie) des méthodes formelles dans l'industrie sans se heurter au mur de l'explosion combinatoire
- La société Argosim va être créée (juin 2013) dans le but d'industrialiser ces idées/outils