UNIVERSITÉ PARIS SUD

L3 informatique et MIAGE

Année 2016-2017

Génie logiciel avancé

Classification des types de test

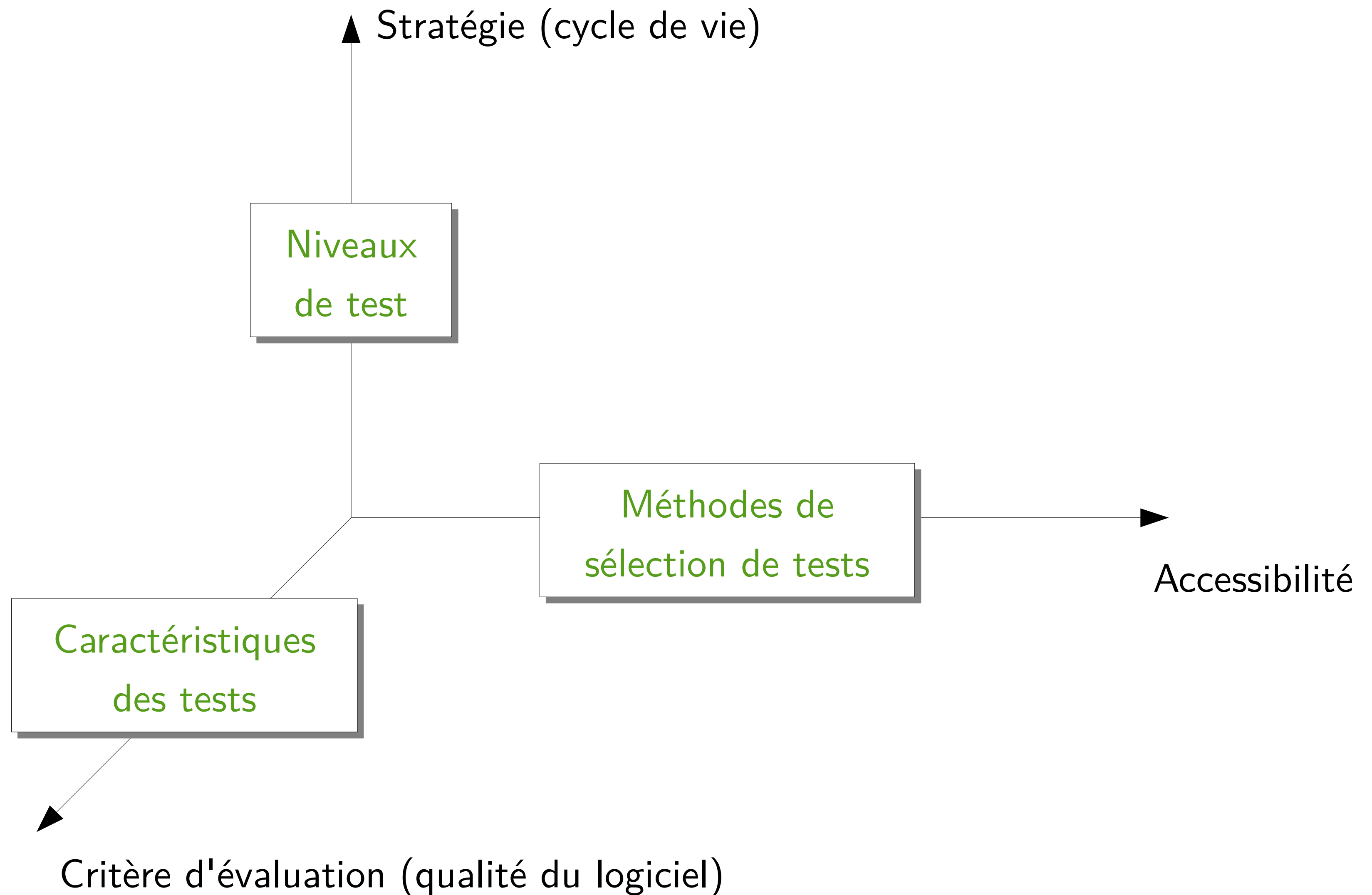
Test à partir de spécifications formelles

Delphine Longuet

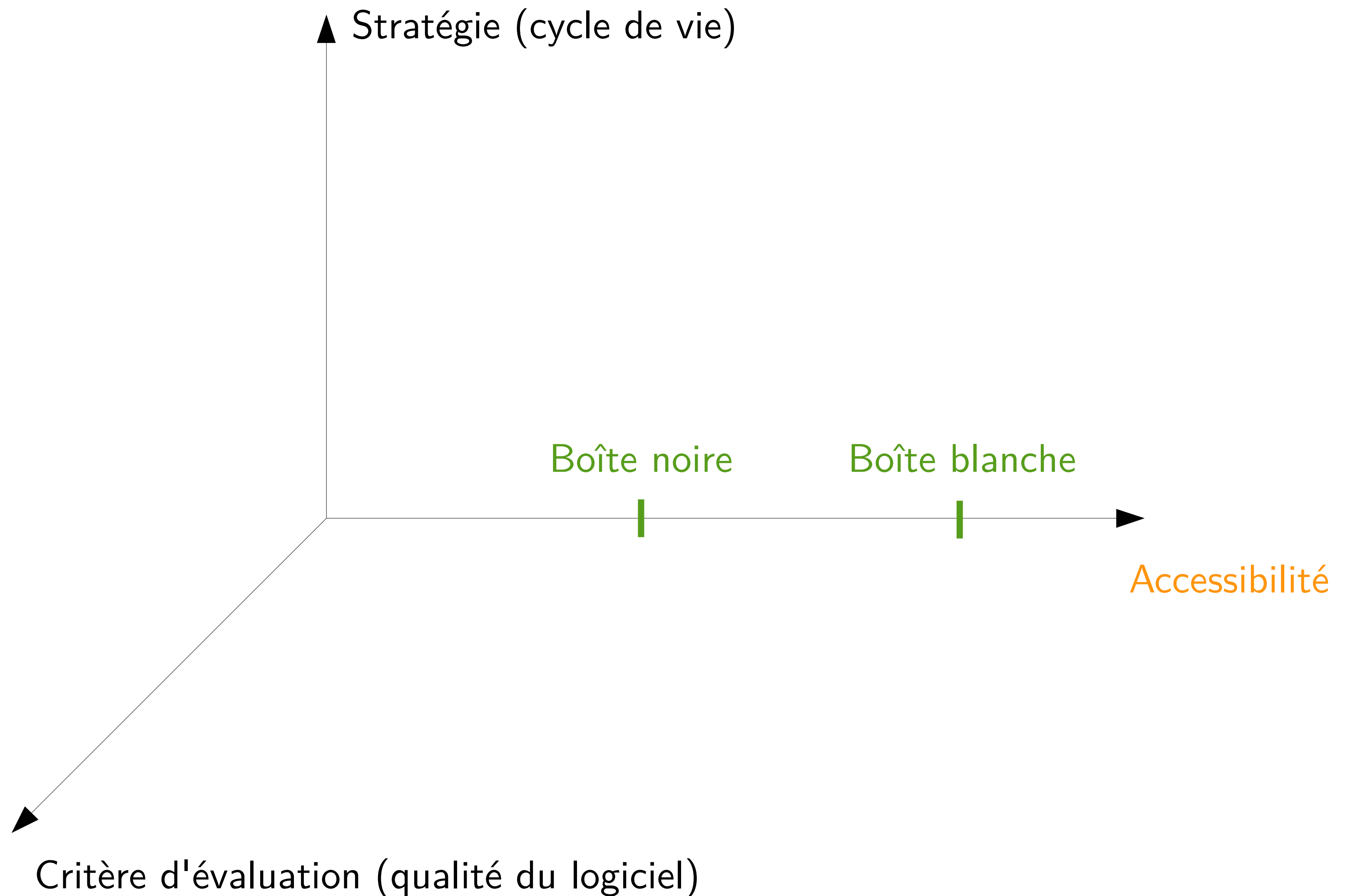
delphine.longuet@lri.fr

http://www.lri.fr/~longuet/Enseignements/16-17/L3-GLA

Types de tests

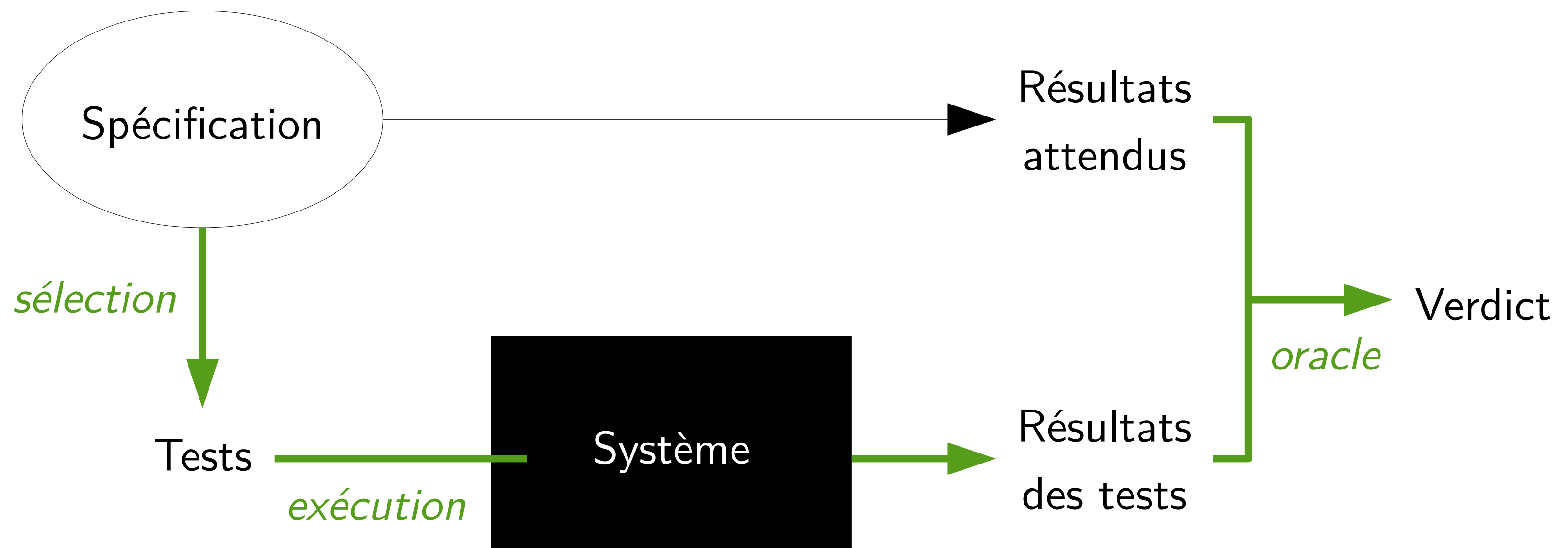


Types de tests



Test fonctionnel (ou test boîte noire)

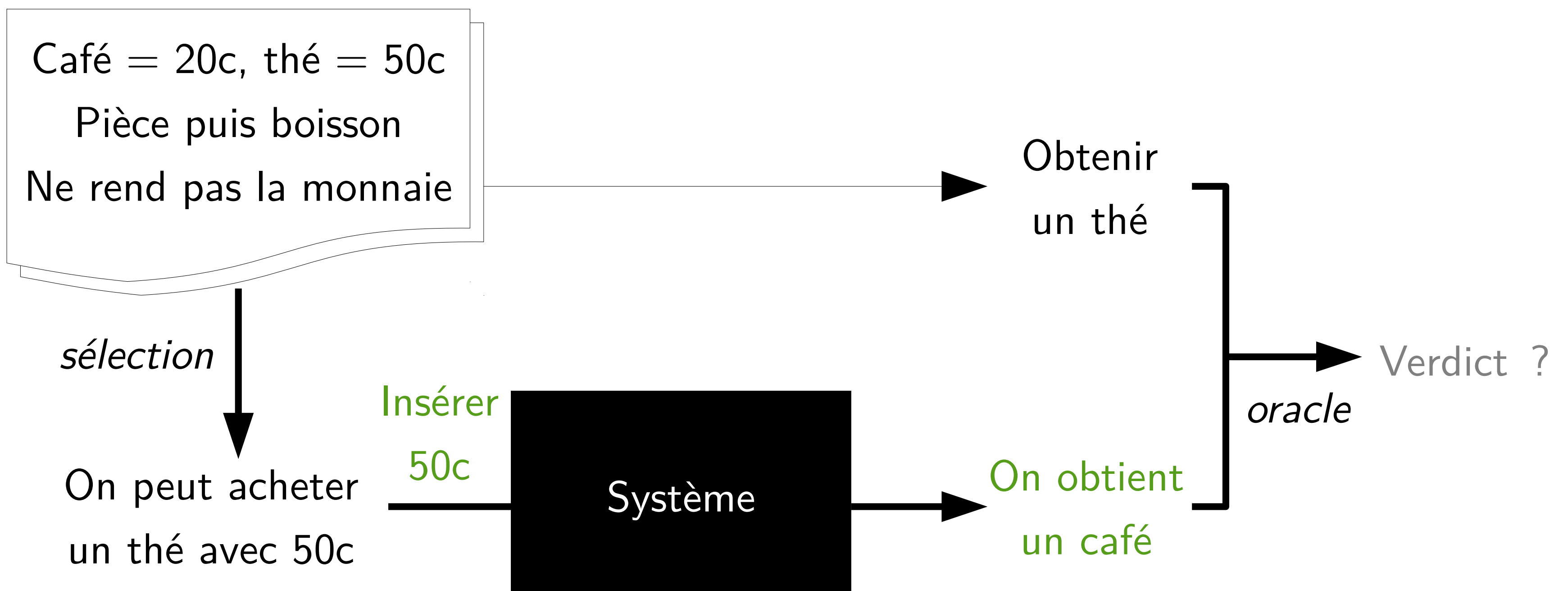
Sélection des tests à partir d'une spécification du système (formelle ou informelle), sans connaissance de l'implantation



Possibilité de construire les tests pendant la conception, avant la programmation

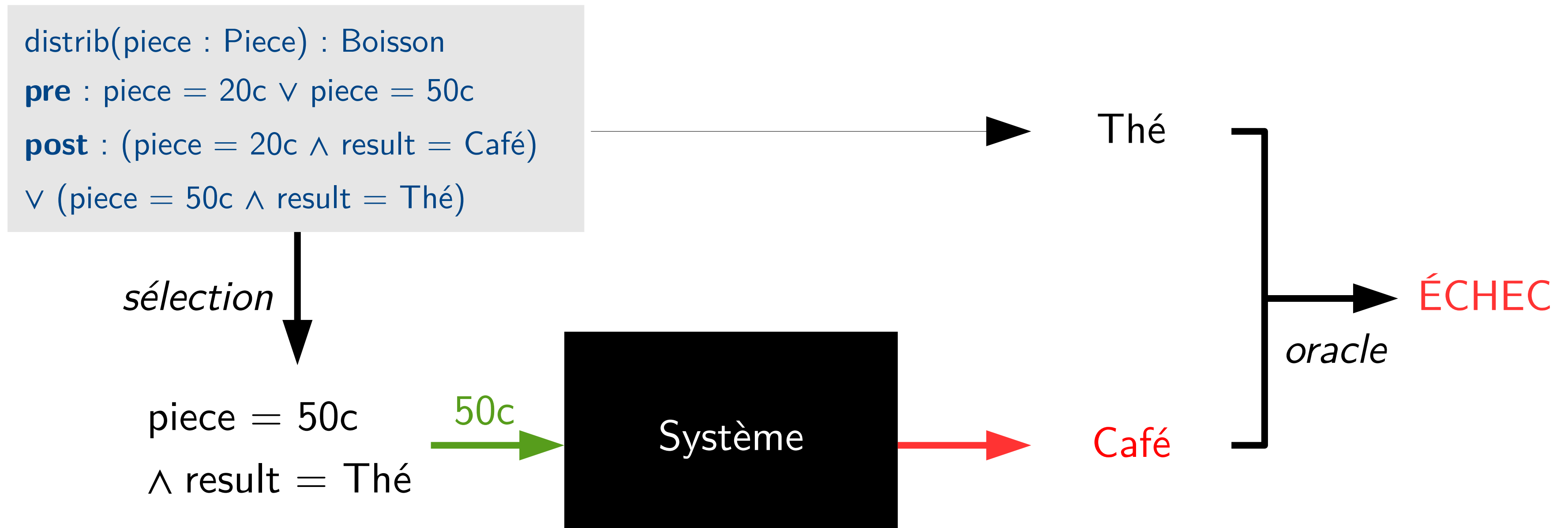
Test fonctionnel

Distributeur de café et thé



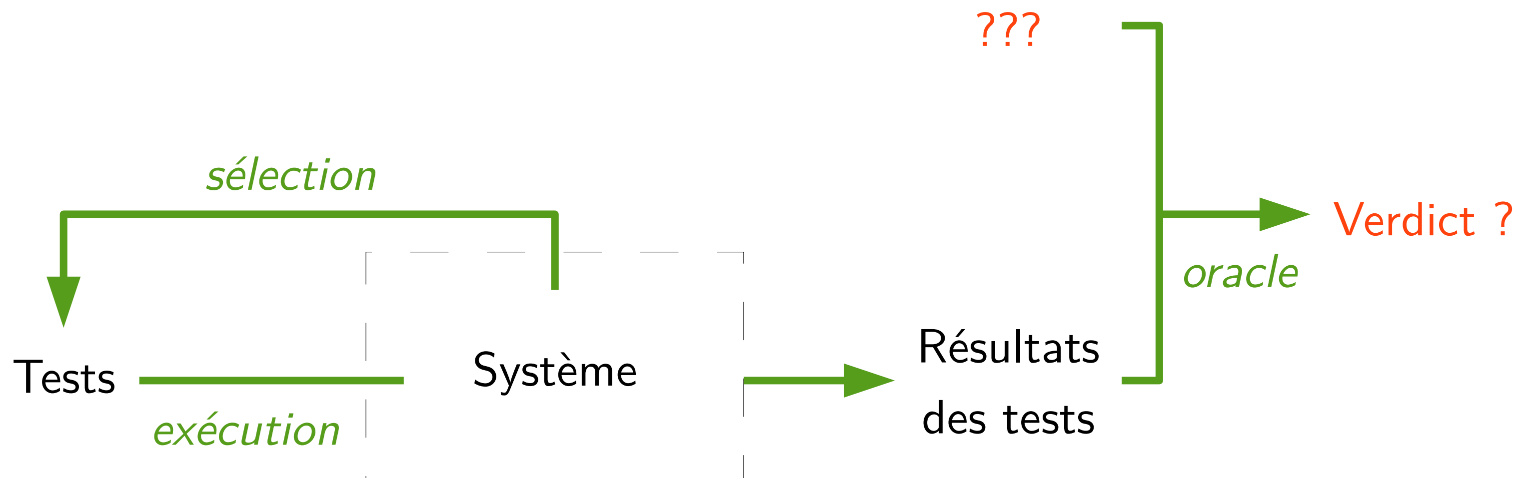
À partir d'une spécification formelle

Distributeur de café et thé



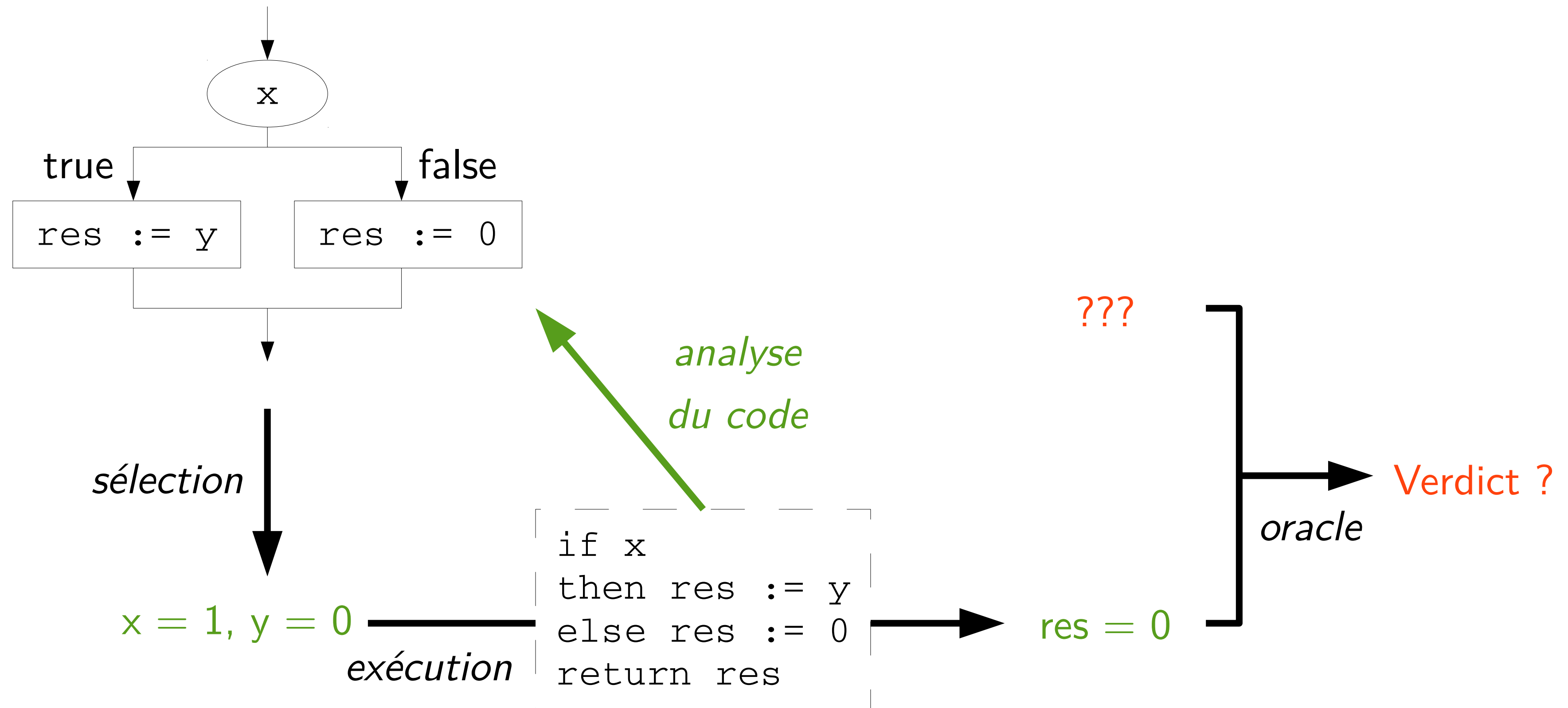
Test structurel (ou test boîte blanche)

Sélection des tests à partir de l'analyse du code source du système

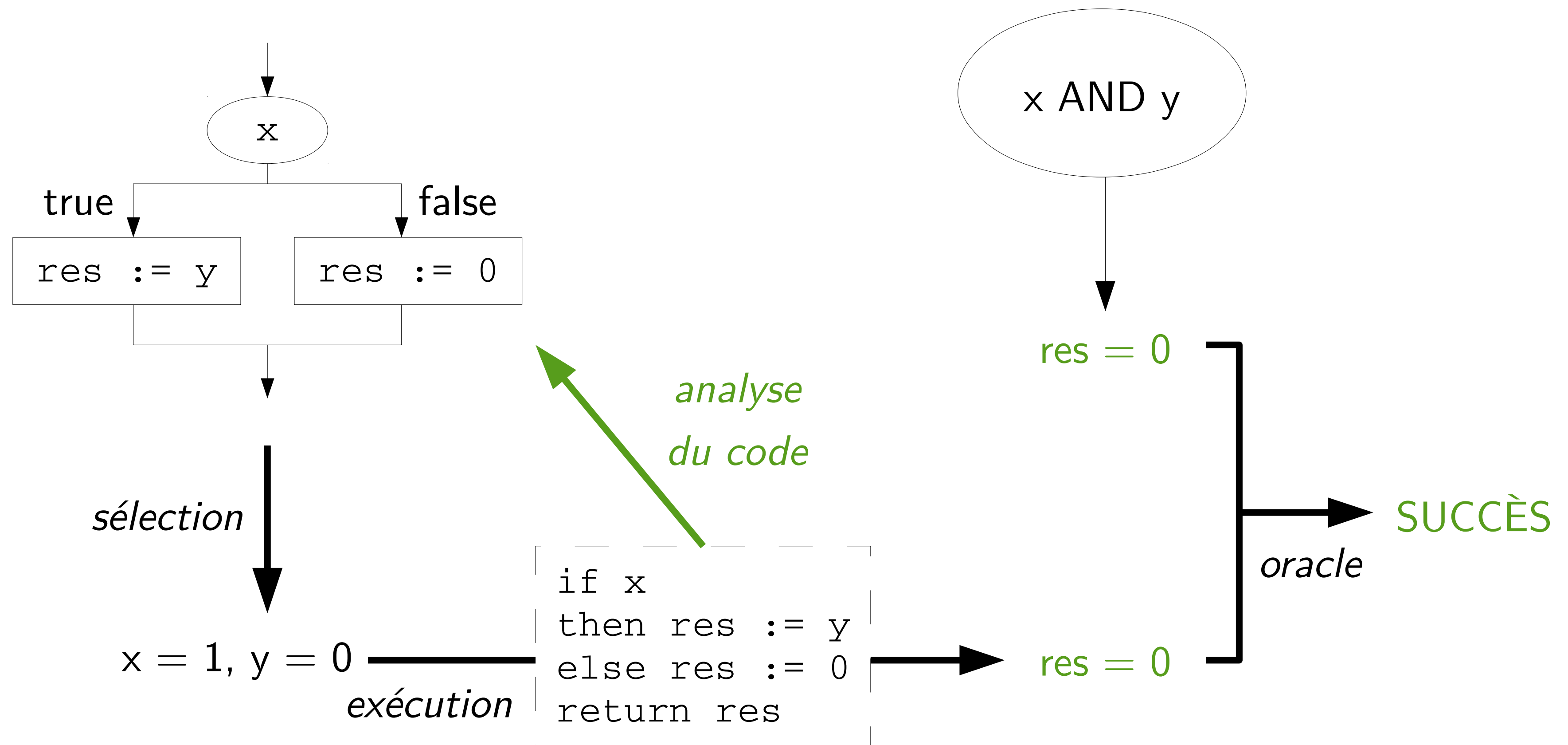


Construction des tests uniquement pour du code déjà écrit

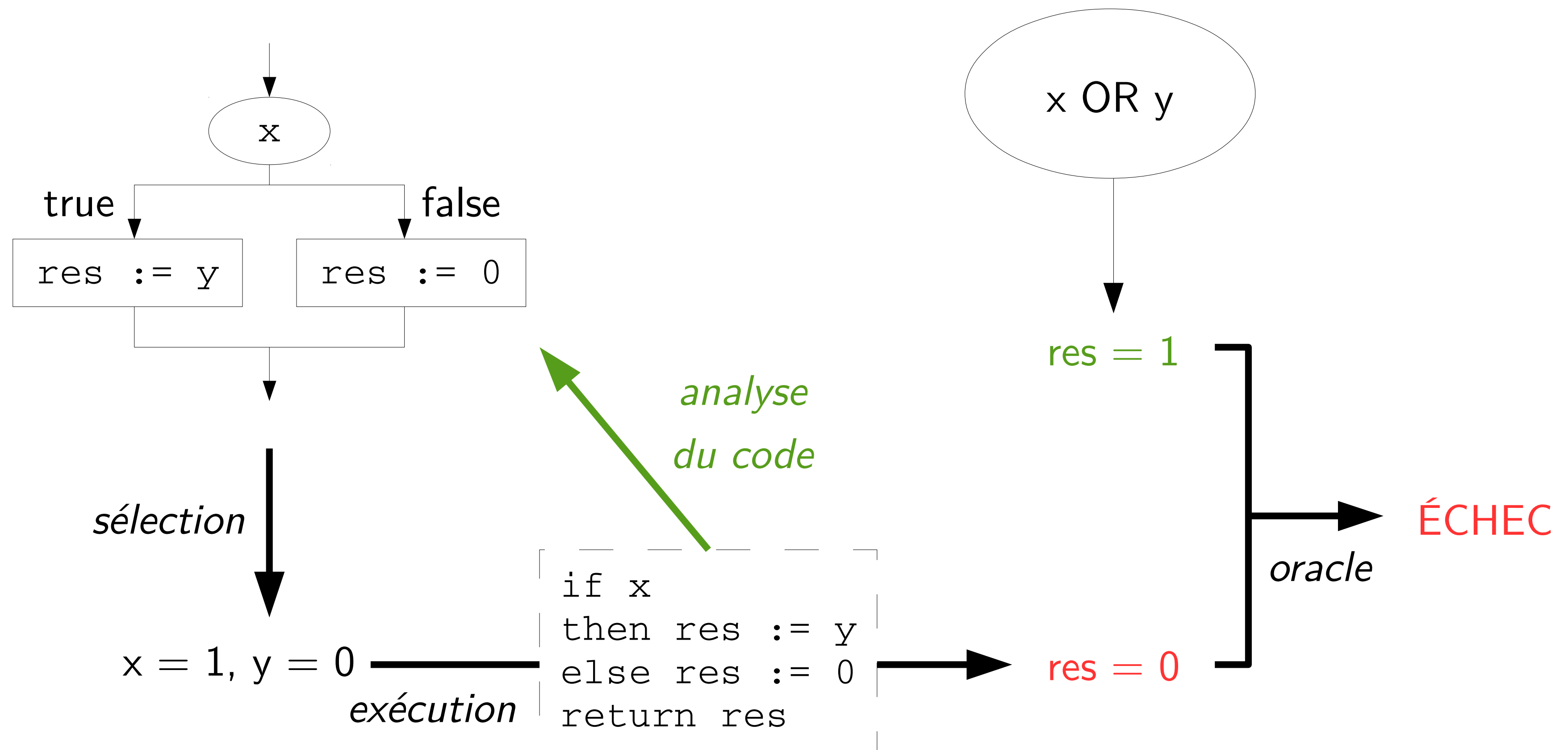
Test structurel



Test structurel

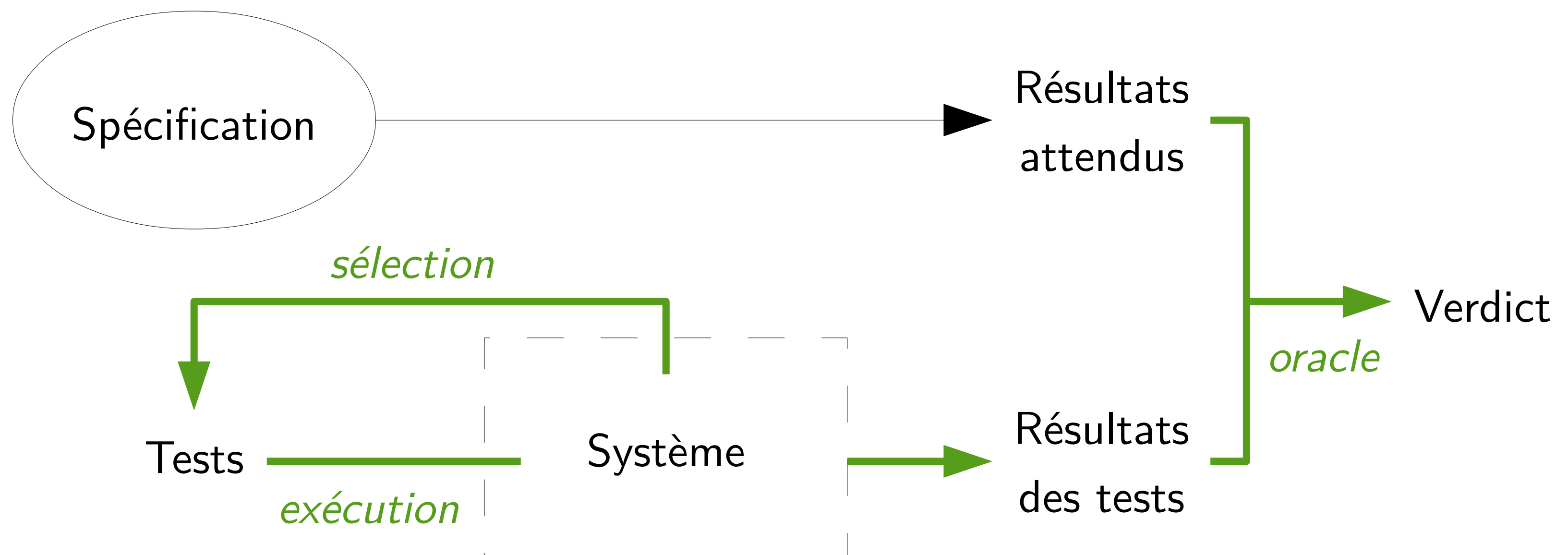


Test structurel



Test structurel

Sélection des tests à partir de l'analyse du code source du système



Construction des tests uniquement pour du code déjà écrit

Test fonctionnel vs. structurel

Complémentarité : détection de fautes différentes

- Test fonctionnel : détecte les oublis ou les erreurs par rapport à la spécification
- Test structurel : détecte les erreurs de programmation

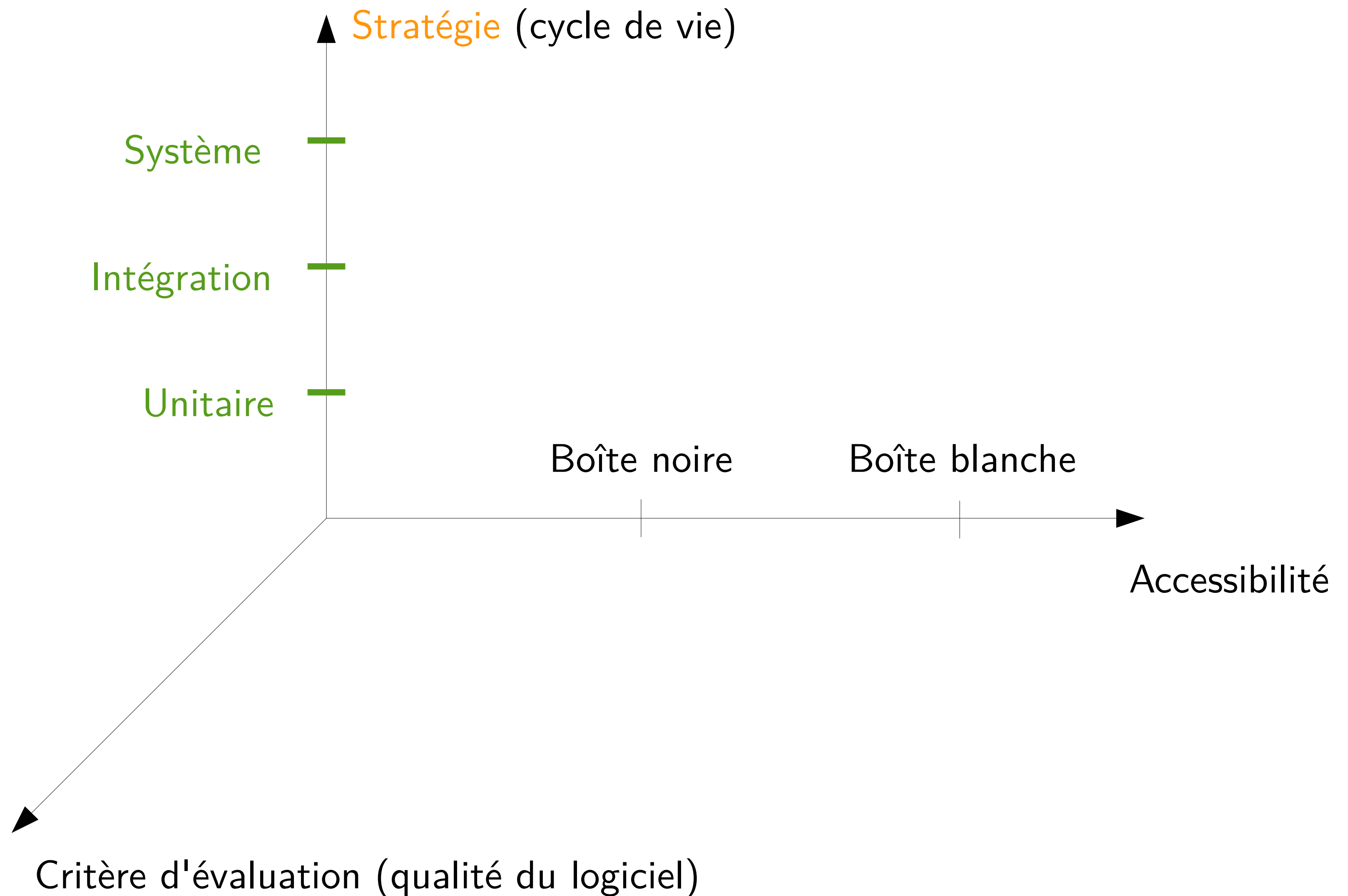
Addition d'entiers modulo 100 000

```
Function sum(x,y : integer) : integer
   if (x = 600 and y = 500)
   then sum := x - y
   else sum := x + y
```

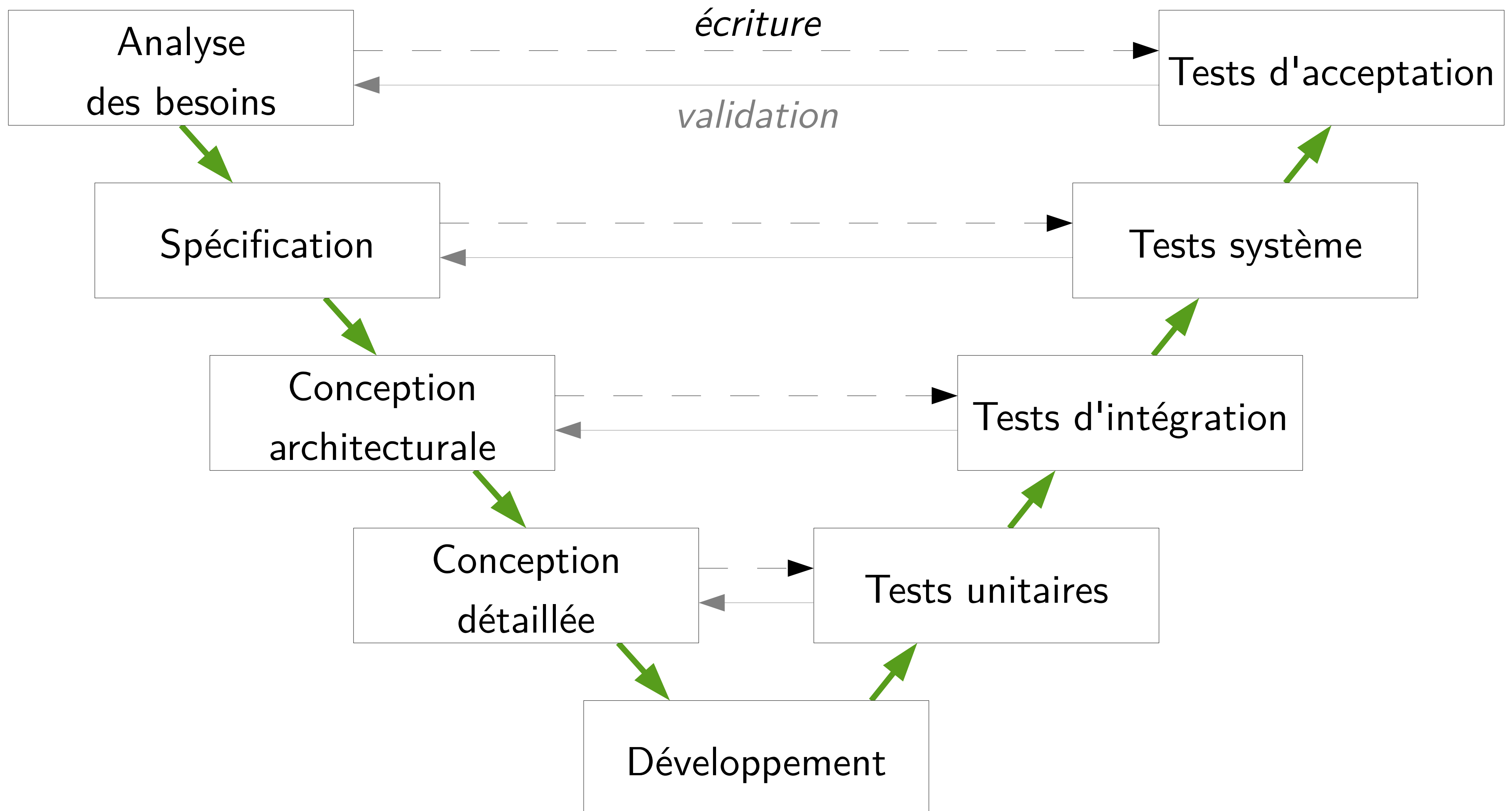
Test fonctionnel : détecte l'erreur par rapport à la spécification

Test structurel : détecte l'erreur pour les valeurs (600,500)

Types de tests



Cycle de vie du logiciel



Tests unitaires

Objectif : s'assurer que chaque unité de programme (méthode, classe, composant) remplit sa fonction, indépendamment du reste du système

- Tests effectués en isolant chaque partie du système
- Simulation nécessaire des parties dépendantes (bouchons)
- Tests ciblés sur les fonctionnalités associées à la méthode, à la classe, au composant
- (Possibilité d'utiliser des outils de mesure de la qualité des tests en terme de couverture de code)

Méthode

- Test incrémental : méthode, classe, package, composant

Tests d'intégration

Objectif : s'assurer que les composants interagissent correctement entre eux, par exemple :

- appels de méthodes extérieures (conditions sur les paramètres)
- accès à une base de données
- récupération des données saisies via l'interface utilisateur

Méthode

- Tests ciblés sur les interactions entre composants (à tous les niveaux)
- Tests construits à partir de :
  - l'architecture du système
  - la spécification des interfaces des composants

Tests système

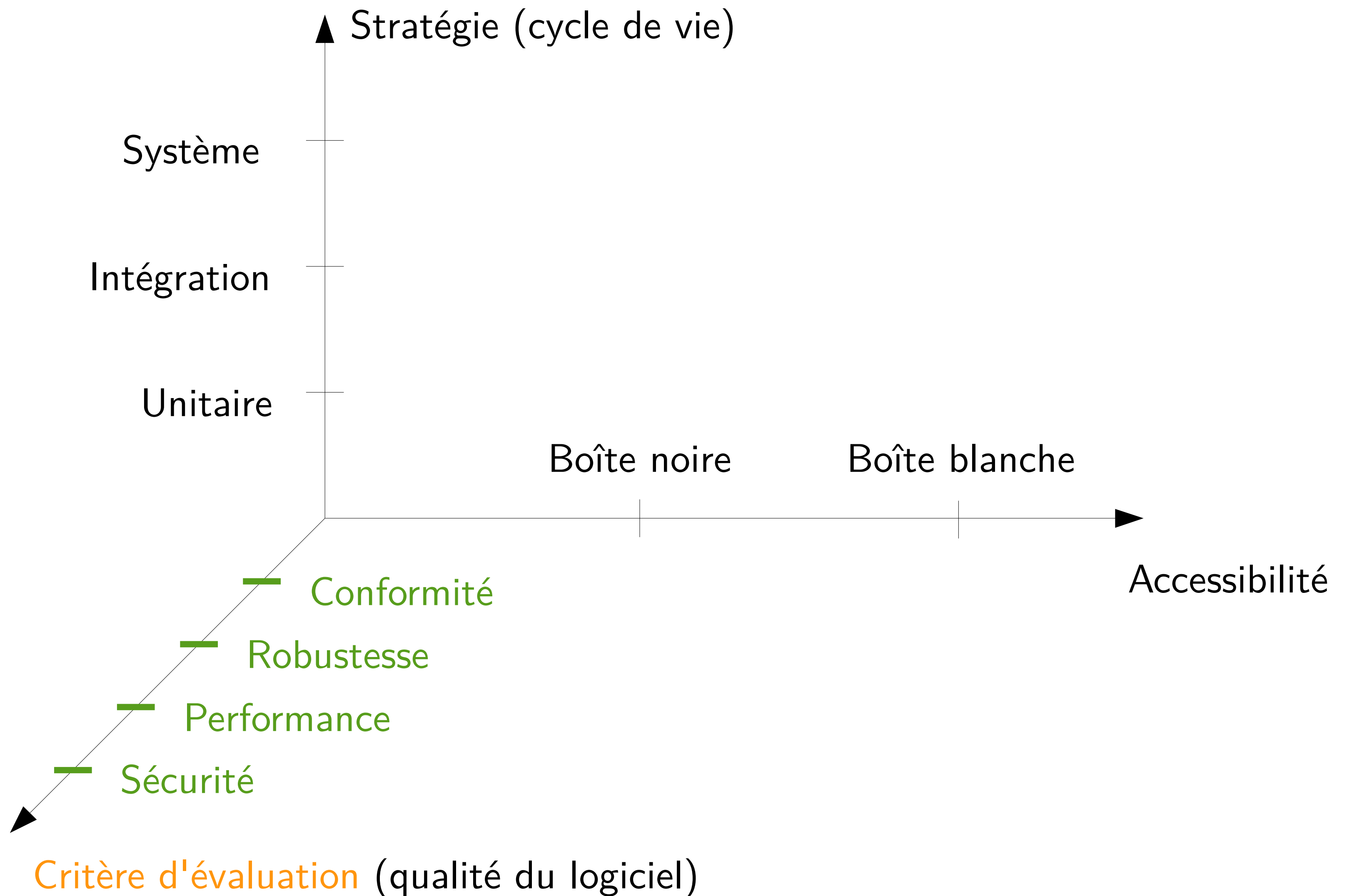
Objectif : s'assurer de la conformité du produit fini par rapport à son cahier des charges

- Tests effectués en boîte noire au travers de l'interface
- Tests ciblés sur les exigences décrites dans la spécification
- Construction possible des tests à partir des cas et des scénarios d'utilisation

Intérêt de la planification des tests système en phase d'analyse

- Validation du produit fini par rapport aux spécifications initiales
- Permet de s'assurer qu'on n'a pas dévié des spécifications

Types de tests



Test de conformité

But : Assurer que le système présente les fonctionnalités attendues par l'utilisateur

Méthode : Sélection des tests à partir de la spécification, de façon à contrôler que toutes les fonctionnalités spécifiées sont implantées selon leurs spécifications

Service de paiement en ligne :

- Scénarios avec transaction acceptée/refusée, couverture des différents cas et cas d'erreur prévus

Test de robustesse

But : Assurer que le système supporte les utilisations imprévues

Méthode : Sélection des tests en dehors des comportements spécifiés (entrées hors domaine, utilisation incorrecte de l'interface, environnement dégradé...)

Service de paiement en ligne :

- Login dépassant la taille du buffer
- Coupure réseau pendant la transaction

Test de sécurité

But : Assurer que le système ne possède pas de vulnérabilités permettant une attaque de l'extérieur

Méthode : Simulation d'attaques pour découvrir les faiblesses du système qui permettraient de porter atteinte à son intégrité

Service de paiement en ligne :

- Essayer d'utiliser les données d'un autre utilisateur
- Faire passer la transaction pour terminée sans avoir payé

Test de performance

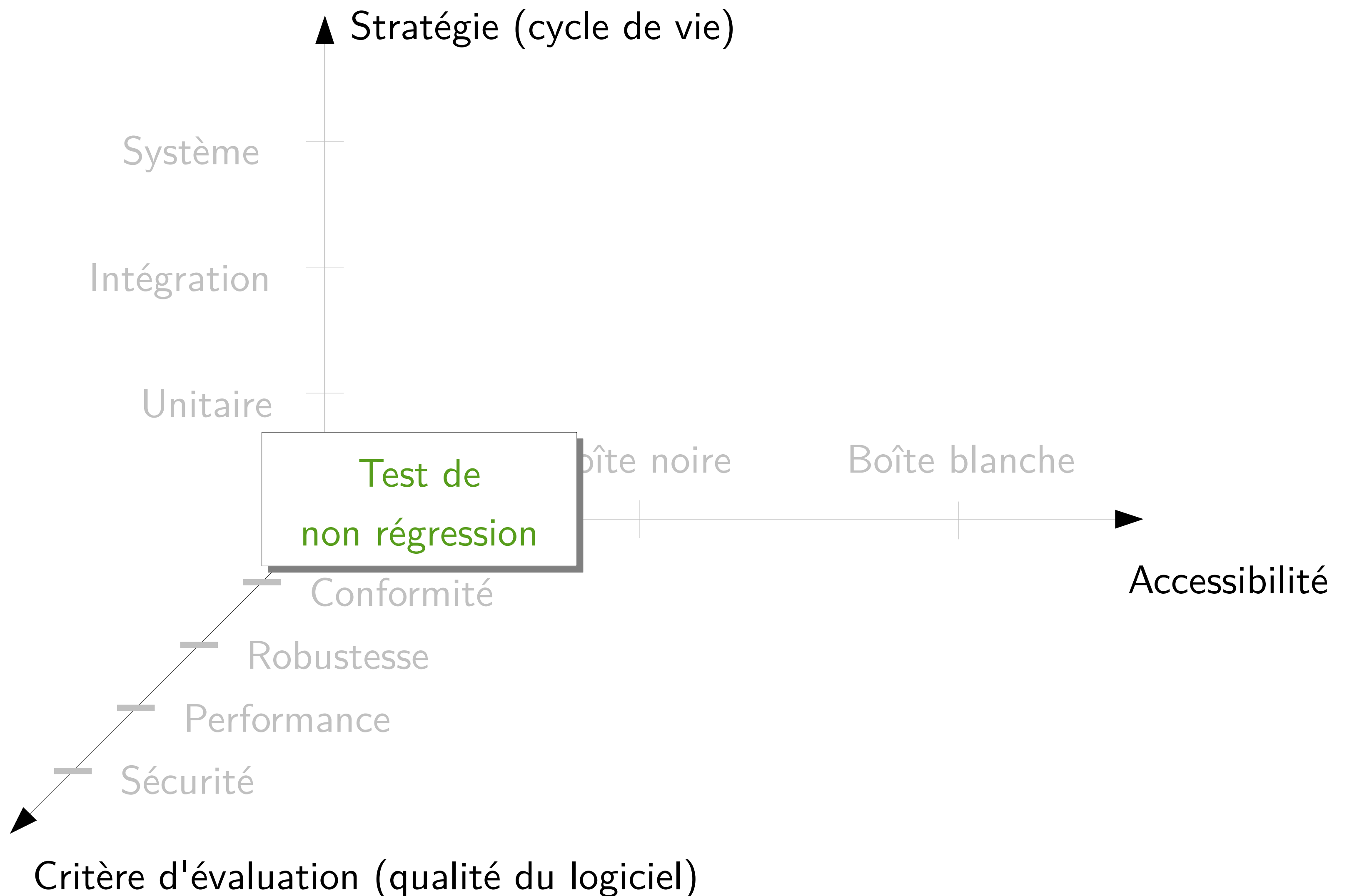
But : Assurer que le système garde des temps de réponse satisfaisants à différents niveaux de charge

Méthode : Simulation à différents niveaux de charge d'utilisateurs pour mesurer les temps de réponse du système, l'utilisation des ressources...

Service de paiement en ligne :

- Lancer plusieurs centaines puis milliers de transactions en même temps

Un type de test transversal



Test de non régression

But : Assurer que les corrections et les évolutions du code n'ont pas introduit de nouveaux défauts

Méthode : À chaque ajout ou modification de fonctionnalité, rejouer les tests pour cette fonctionnalité, puis pour celles qui en dépendent, puis les tests des niveaux supérieurs

↳ Lourd mais indispensable
Automatisable en grande partie

Test à partir de spécifications formelles

Rappels sur les méthodes informelles

Objectif général : couvrir l'ensemble des comportements ou des fautes possibles :

- analyse partitionnelle : couverture des comportements (un comportement par classe d'équivalence)
- test aux limites : couverture des bornes (fautes aux limites)
- pairwise : couverture des couples (fautes par couple de variables)
- graphe cause-effet : couverture des liens cause-effet (un comportement par lien)

Problème de ces méthodes

Problème : évaluation de la qualité des tests construits

- Méthodes subjectives fondées sur l'intuition et l'expérience (par ex : qualité de la partition ?)
- Modèles informels : couverture mal définie, pas quantifiable et pas automatisable

Solution : utiliser des modèles formels

- Vérification possible des modèles (model-checking, preuve)
- Automatisation possible de la génération de tests par couverture du modèle

Test à partir de spécifications formelles

Écrire une spécification pour l'opération qui calcule le maximum de trois entiers.

max(x : int, y : int, z : int) : int
pre : ...
post : ...

Problème : comment dériver automatiquement les cas à tester pour couvrir la spécification ?

Test à partir de spécifications formelles

Écrire une spécification pour l'opération qui calcule le maximum de trois entiers.

max(x : int, y : int, z : int) : int

pre : true

post : $result \geq x \wedge result \geq y \wedge result \geq z$

$\wedge\ (result = x \vee result = y \vee result = z)$

Problème : comment dériver automatiquement les cas à tester pour couvrir la spécification ?

Test à partir de spécifications formelles

Cas simple : spécification réduite à une formule (post-condition)

$$\text{result} \geq x \wedge \text{result} \geq y \wedge \text{result} \geq z$$
$$\wedge\ (\text{result} = x \vee \text{result} = y \vee \text{result} = z)$$

Méthode : calculer la forme normale disjonctive de la formule :

$$C_1 \vee C_2 \vee C_3 \vee \ldots \vee C_n$$

où C_1, C_2, ..., C_n sont des conjonctions

Couverture de la spécification : un test par cas C_i

Forme normale disjonctive (DNF)

Post-condition

$$\text{result} \geq x \wedge \text{result} \geq y \wedge \text{result} \geq z \wedge (\text{result} = x \vee \text{result} = y \vee \text{result} = z)$$

Application de la distributivité :

$$(\text{result} \geq x \wedge \text{result} \geq y \wedge \text{result} \geq z \wedge \text{result} = x) \quad C_1$$
$$\vee (\text{result} \geq x \wedge \text{result} \geq y \wedge \text{result} \geq z \wedge \text{result} = y) \quad C_2$$
$$\vee (\text{result} \geq x \wedge \text{result} \geq y \wedge \text{result} \geq z \wedge \text{result} = z) \quad C_3$$

Jeu de tests obtenu :

Objectif de test	Données d'entrée			Résultat attendu
	x	y	z	
C_1	10	5	-2	10
C_2	4	59	9	59
C_3	-7	-18	-1	-1

Test à partir de spécifications formelles

Invariant de la classe Compte

$\forall$ c $\in$ Compte, c.solde $\leq$ c.plafond $\wedge$ c.plafond $>$ 0 $\wedge$ c.frais $\geq$ 0

Compte
solde : real plafond : real frais : real
déposer(montant:real):real

Spécification de l'opération déposer

Compte :: déposer(montant : real) : real

pre : montant $>$ 0 $\wedge$ this.solde $+$ montant $\leq$ this.plafond

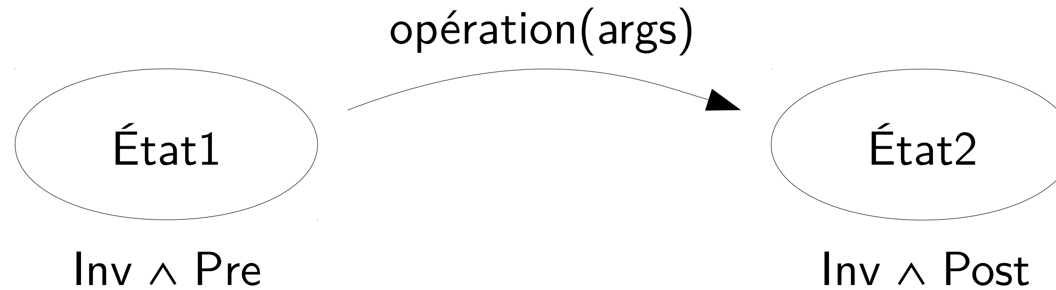
post : (old(this.solde) $+$ montant $\geq$ 0 $\Rightarrow$ this.solde $=$ old(this.solde) $+$ montant)

$\wedge$ (old(this.solde) $+$ montant $<$ 0 $\Rightarrow$ this.solde $=$ old(this.solde) $+$ montant $-$ this.frais)

$\wedge$ result $=$ this.solde

Dérivation des objectifs de test : de quelle formule faut-il calculer la DNF ?

Sémantique d'un contrat d'opération



Inv : invariants	$\forall$ a $\in$ Classe, Inv(a)
Pre : pré-condition de l'opération	Pre(this,arg_1,…,arg_n)
Post : post-condition de l'opération	Post(this,arg_1,…,arg_n,result)

Sémantique de la spécification de l'opération, exprimée dans État2 :

$$\forall a \in Classe,\ old(Inv(a)) \land old(Pre(a,arg_1,\ldots,arg_n)) \land Inv(a) \land Post(a,arg_1,\ldots,arg_n,result)$$

Test d'une opération à partir de son contrat

But : dériver un ensemble d'objectifs de tests à partir du contrat de l'opération et des invariants de la classe

Analyse partitionnelle : objectif de test = sous-ensemble des instances valides du système et des valeurs valides des arguments et du résultat, correspondant à un comportement particulier de l'opération

Invariants : instances valides du système
Pré-condition : instances valides avant l'opération et valeurs valides des arguments
Post-condition : instances valides après l'opération et valeurs valides du résultat

Test d'une opération à partir de son contrat

Invariant de la classe Compte

$\forall$ c $\in$ Compte, c.solde $\leq$ c.plafond $\wedge$ c.plafond > 0 $\wedge$ c.frais ≥ 0

Compte
solde : real plafond : real frais : real
déposer(montant:real):real

Spécification de l'opération déposer

Compte :: déposer(montant : real) : real

pre : montant > 0 $\wedge$ this.solde $+$ montant $\leq$ this.plafond

post : (old(this.solde) $+$ montant $\geq 0 \Rightarrow$ this.solde $=$ old(this.solde) $+$ montant)

$\wedge$ (old(this.solde) $+$ montant $< 0 \Rightarrow$ this.solde $=$ old(this.solde) $+$ montant $-$ this.frais)

$\wedge$ result $=$ this.solde

Dérivation des objectifs de test : calcul de la DNF de la formule

$$\text{old}(\text{Inv}(c)) \wedge \text{old}(\text{Pre}(c, \text{montant})) \\ \wedge \text{Inv}(c) \wedge \text{Post}(c, \text{montant}, \text{result})$$

Appliquer old à une formule

old(P) : formule P dans laquelle on remplace **a.expression** par **old(a.expression)**, la valeur de l'expression dans l'état précédent de l'objet **a**

$\text{old}(\text{Pre}(\text{c, montant})) \Leftrightarrow \text{montant} > 0 \wedge \text{old}(\text{c.solde}) + \text{montant} \leq \text{old}(\text{c.plafond})$

$\text{old}(\text{Inv}(\text{c})) \Leftrightarrow \text{old}(\text{c.solde}) \leq \text{old}(\text{c.plafond}) \wedge \text{old}(\text{c.plafond}) > 0 \wedge \text{old}(\text{c.frais}) \geq 0$

Calcul de la DNF

Formule de départ :

old(Inv)	$\text{old}(c.solde) \leq \text{old}(c.plafond) \land \text{old}(c.plafond) > 0 \land \text{old}(c.frais) \geq 0$
$\land$ old(Pre)	$\land\ montant > 0 \land \text{old}(c.solde) + montant \leq \text{old}(c.plafond)$
$\land$ Inv	$\land\ c.solde \leq c.plafond \land c.plafond > 0 \land c.frais \geq 0$
$\land$ Post	$\land\ ((\text{old}(c.solde) + montant \geq 0 \Rightarrow c.solde = \text{old}(c.solde) + montant)$ $\land\ (\text{old}(c.solde) + montant < 0 \Rightarrow c.solde = \text{old}(c.solde) + montant - c.frais)$ $\land\ result = c.solde)$

DNF de cette formule ?

- old(Inv), old(Pre) et Inv : conjonctions
- Post : éliminer les implications et calculer sa DNF

Calcul de la DNF de Post

Notations pour simplifier les calculs :

$$\begin{aligned}
\text{Post} \Leftrightarrow\ & (\underbrace{\text{old(c.solde)} + \text{montant} \geq 0}_{A} \Rightarrow \underbrace{\text{c.solde} = \text{old(c.solde)} + \text{montant}}_{B}) \\
& \wedge (\text{old(c.solde)} + \text{montant} < 0 \Rightarrow \underbrace{\text{c.solde} = \text{old(c.solde)} + \text{montant} - \text{c.frais}}_{C}) \\
& \wedge \underbrace{\text{result} = \text{c.solde}}_{D}
\end{aligned}$$

Calcul de la DNF :

$$\begin{aligned}
\text{Post} & \Leftrightarrow (A \Rightarrow B) \wedge (\neg A \Rightarrow C) \wedge D \\
& \Leftrightarrow (\neg A \vee B) \wedge (A \vee C) \wedge D \\
& \Leftrightarrow (\neg A \wedge C \wedge D) \vee (A \wedge B \wedge D) \qquad \text{(B et C incompatibles)} \\
& \Leftrightarrow \text{Post1} \vee \text{Post2}
\end{aligned}$$

Finalement :

$$\begin{aligned}
\text{Post1} \Leftrightarrow\ & \text{old(c.solde)} + \text{montant} \geq 0 \\
& \wedge \text{c.solde} = \text{old(c.solde)} + \text{montant} \\
& \wedge \text{result} = \text{c.solde}
\end{aligned}$$

$$\begin{aligned}
\text{Post2} \Leftrightarrow\ & \text{old(c.solde)} + \text{montant} < 0 \\
& \wedge \text{c.solde} = \text{old(c.solde)} + \text{montant} - \text{c.frais} \\
& \wedge \text{result} = \text{c.solde}
\end{aligned}$$

Objectifs de test obtenus

Cas 1 : le dépôt rend le solde positif, alors le solde a été augmenté du montant du dépôt et le résultat renvoyé est la nouvelle valeur du solde

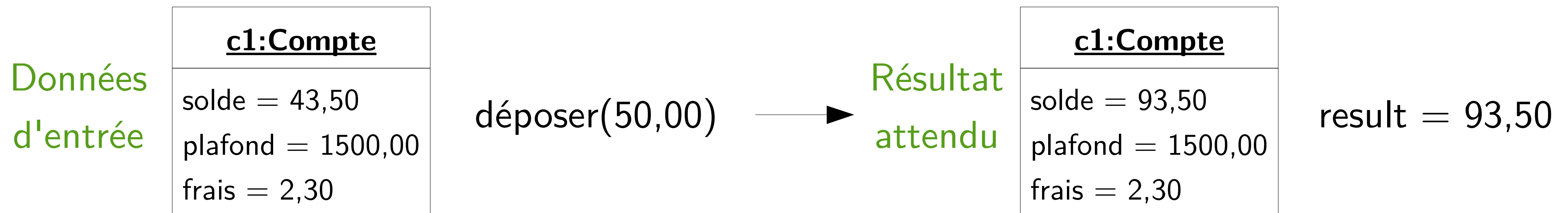
old(Inv)	$old(c.solde) \leq old(c.plafond) \land old(c.plafond) > 0 \land old(c.frais) \geq 0$
$\land$ old(Pre)	$\land\ montant > 0 \land old(c.solde) + montant \leq old(c.plafond)$
$\land$ Inv	$\land$ ~~$c.solde \leq c.plafond$~~ $\land$ ~~$c.plafond > 0$~~ $\land$ ~~$c.frais \geq 0$~~
$\land$ Post1	$\land\ old(c.solde) + montant \geq 0 \land c.solde = old(c.solde) + montant \land result = c.solde$

Cas 2 : le dépôt laisse le solde négatif, alors le solde a été augmenté du montant du dépôt mais des frais ont été prélevés, et le résultat renvoyé est la nouvelle valeur du solde

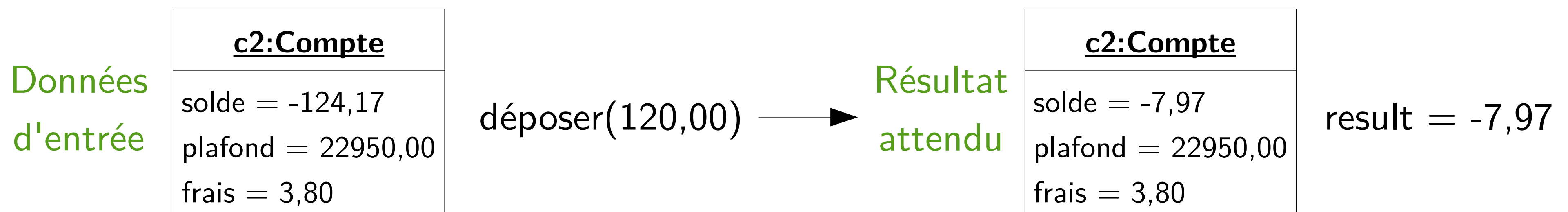
old(Inv)	$old(c.solde) \leq old(c.plafond) \land old(c.plafond) > 0 \land old(c.frais) \geq 0$
$\land$ old(Pre)	$\land\ montant > 0 \land old(c.solde) + montant \leq old(c.plafond)$
$\land$ Inv	$\land$ ~~$c.solde \leq c.plafond$~~ $\land$ ~~$c.plafond > 0$~~ $\land$ ~~$c.frais \geq 0$~~
$\land$ Post2	$\land\ old(c.solde) + montant < 0 \land c.solde = old(c.solde) + montant - c.frais \land result = c.solde$

Cas de test obtenus

Cas 1 : le dépôt rend le solde positif, alors le solde a été augmenté du montant du dépôt et le résultat renvoyé est la nouvelle valeur du solde



Cas 2 : le dépôt laisse le solde négatif, alors le solde a été augmenté du montant du dépôt mais des frais ont été prélevés, et le résultat renvoyé est la nouvelle valeur du solde



Test et spécifications formelles

Trois exemples d'utilisation des spécifications formelles pour le test

- Automatisation de l'oracle pour des tests écrits à la main

  Outil : OpenJML
- Génération aléatoire guidée et automatisation de l'oracle

  Outil : QuickCheck
- Génération automatique de tests

  Outil : HOL-TestGen

Vérification à l'exécution

Spécifications insérées comme annotations dans le code

exemple :

Java+JML

```
public class CompteBancaire {
    private /*@ spec_public @*/ int solde;
    //@ invariant solde >= 0;

    /*@ requires montant > 0 && solde >= montant;
      @ ensures solde == \old(solde) - montant;
      @*/
    public void debiter(int montant) { ... }
}
```

Spécifications exécutables, vérifiées pendant l'exécution du programme

`C.debiter(m)` →

1. Vérification de l'invariant de `Compte` sur `C`
2. Vérification de la pré-condition de `debiter` sur `C` et `m`
3. Exécution de `debiter` sur `C` et `m`
4. Vérification de l'invariant de `Compte` sur `C`
5. Vérification de la post-condition de `debiter` sur `C`, `m` et le résultat renvoyé

Arrêt de l'exécution si une des conditions est violée

Vérification à l'exécution

Intérêt : factoriser et automatiser l'oracle (décision de l'adéquation entre résultat attendu et résultat obtenu)

↳ Tests réduits à la construction de l'état et aux appels des opérations à tester

```
public class Compte {
    private /*@ spec_public @*/ double
solde;
    //@ invariant solde >= 0;

    /*@ requires s >= 0;
      @ ensures solde == s;
      @*/
    public Compte(double s) { }

    /*@ requires m > 0 && m <= solde;
      @ ensures solde == \old(solde) - m;
      @*/
    public void debiter(double m) { }

    //@ ensures \result == solde;
    public double solde() { }
}
```

```
public class TestCompte {

    public static void testDebiter() {
        Compte c = new Compte(200);
        c.debiter(50);
        assert(c.solde() == 150);
        c.debiter(150);
        assert(c.solde() == 0);
    }
}
```

OpenJML *runtime assertion checker*

OpenJML : outil de vérification statique et dynamique pour Java

Runtime assertion checker : vérification des spécifications à l'exécution

```
public class Compte {
    private /*@ spec_public @*/ double
solde;
    //@ invariant solde >= 0;

    /*@ requires s >= 0;
      @ ensures solde == s;
      @*/
    public Compte(double s) { }

    /*@ requires m > 0 && m <= solde;
      @ ensures solde == \old(solde) - m;
      @*/
    public void debiter(double m) { }

    //@ ensures \result == solde;
    public double solde() { }
}
```

```
public class TestCompte {

    public static void testDebiter() {
        Compte c = new Compte(200);
        c.debiter(50);
        c.debiter(150);
        c.debiter(40);
    }
}
```

```
>jmlc Compte.java
>jmlrac Compte
org.jmlspecs.jmlrac.runtime.JMLInterna
lPreconditionError: by method
Compte.debiter
    at Compte.main<Compte.java:1487>
```

Génération aléatoire guidée

Spécifications dans un fichier accompagnant le code

```
prop_time_to_seconds() ->
    ?FORALL({H,M,S},{nat(),nat(),nat()},
    ?IMPLIES(H < 24,
    ?IMPLIES(M < 60,
    ?IMPLIES(S < 60,
             clock:hms2s({H,M,S}) == (H*60*60)+(M*60)+S))))).
```

Spécifications exécutables, utilisées pour :

- filtrer les entrées générées aléatoirement, grâce à la pré-condition
- produire le verdict, grâce à la post-condition

QuickCheck pour Erlang

Librairie de test pour la génération aléatoire de données d'entrée et la vérification à l'exécution des spécifications

```
prop_subtract_seconds_from_time() ->
    ?FORALL({H,M,S,S2},{nat(),nat(),nat(),nat()},
    ?IMPLIES(M < 60,
    ?IMPLIES(S < 60,
    ?IMPLIES(H < 24,
    ?IMPLIES(S2 =< ((H*60*60)+(M*60)+S),
    begin
        T = {H,M,S},
        clock:sub_sec(T,S2) == clock:s2hms((clock:hms2s(T) - S2))
    end))))).
```

```
1>prop_subtract_seconds_from_time: ............................
x.......................................xxx..xx..........
.....x.x..xx...
OK, passed 100 tests
1% {22,23,28,21}
1% {21,6,24,1}
1% {20,15,13,11}
(etc)
```

HOL-TestGen

Outil de test construit au-dessus de l'assistant de preuve Isabelle

Génération automatique de tests à partir de spécifications

```
theory List_test
imports Main begin

consts is_sorted:: "('a::ord) list ⇒ bool"
primrec
"is_sorted [] = True"
"is_sorted (x#xs) = case xs of
                     [] ⇒ True
                    | y#ys ⇒ ((x < y)∨(x =
y))
                              ∧is_sorted xs"

test_spec "is_sorted (prog (l::('a list)))"
apply(gen_test_cases prog)
store_test_thm "test_sorting"
gen_test_data "test_sorting"
gen_test_script "test_lists.sml" list" prog
end
```

```
Test Results:
Test 0 - *** FAILURE: post-condition
false, result: [1, 0, 10]
Test 1 - SUCCESS, result: [6, 8]
Test 2 - SUCCESS, result: [3]
Test 3 - SUCCESS, result: []
Summary:
Number successful tests cases: 3 of 4
(ca. 75%)
Number of warnings: 0 of 4 (ca. 0%)
Number of errors: 0 of 4 (ca. 0%)
Number of failures: 1 of 4 (ca. 25%)
Number of fatal errors: 0 of 4 (ca. 0%)
Overall result: failed
```