

Software Testing

Verification and Validation

Nadjib Lazaar

Ing - Phd - Assistant Professor - University of Montpellier - COCONUT Team
<http://www.lirmm.fr/~lazaar/>

Bugs célèbres

Sonde Mariner 1, 1962

- Détruite 294,5 secondes après son lancement
- Coût : 18,5M dollars
- Trait d'union le plus cher de l'histoire



Bugs célèbres

Ariane V vol 501, 1996

- Détruite après 36,7 secondes après son lancement
- Coût : 370M dollars
- Arithmetic overflow



Bugs célèbres

Les plus coûteux de l'histoire

- **Système radar des missiles Patriot** : scud introuvable, 28 morts [02/1991]
- **Multidata Systems International** : double dose de rayons, 8 morts [11/2000]
- **Therac-25** : surdoses de rayonnement, 6 morts de 1985 à 1987
- **Système d'alerte Oko** : detection du lancement de 5 missiles côté US [1980]
- **Gestion pensions alimentaire britannique** : un milliard de dollars [2004]
- **La sonde Mars Climate Orbiter** : accident métrique, 125M dollars [09/1999]

Qualité du Logiciel

Critères de qualité

Qualité du Logiciel

Critères de qualité

- **Validité** : réponse aux attentes du client

Qualité du Logiciel

Critères de qualité

- **Validité** : réponse aux attentes du client
- **Utilisabilité** : prise en main, facilité d'utilisation et d'apprentissage

Qualité du Logiciel

Critères de qualité

- **Validité** : réponse aux attentes du client
- **Utilisabilité** : prise en main, facilité d'utilisation et d'apprentissage
- **Performance** : temps de réponse, complexité des algos, fluidité.

Qualité du Logiciel

Critères de qualité

- **Validité** : réponse aux attentes du client
- **Utilisabilité** : prise en main, facilité d'utilisation et d'apprentissage
- **Performance** : temps de réponse, complexité des algos, fluidité.
- **Fiabilité** : robustesse, sûreté et tolérance aux pannes

Qualité du Logiciel

Critères de qualité

- **Validité** : réponse aux attentes du client
- **Utilisabilité** : prise en main, facilité d'utilisation et d'apprentissage
- **Performance** : temps de réponse, complexité des algos, fluidité.
- **Fiabilité** : robustesse, sûreté et tolérance aux pannes
- **Sécurité** : intégrité des données et protection des accès

Qualité du Logiciel

Critères de qualité

- **Validité** : réponse aux attentes du client
- **Utilisabilité** : prise en main, facilité d'utilisation et d'apprentissage
- **Performance** : temps de réponse, complexité des algos, fluidité.
- **Fiabilité** : robustesse, sûreté et tolérance aux pannes
- **Sécurité** : intégrité des données et protection des accès
- **Maintenabilité** : facilité à corriger ou transformer le logiciel

Qualité du Logiciel

Critères de qualité

- **Validité** : réponse aux attentes du client
- **Utilisabilité** : prise en main, facilité d'utilisation et d'apprentissage
- **Performance** : temps de réponse, complexité des algos, fluidité.
- **Fiabilité** : robustesse, sûreté et tolérance aux pannes
- **Sécurité** : intégrité des données et protection des accès
- **Maintenabilité** : facilité à corriger ou transformer le logiciel
- **Portabilité** : changement d'environnement matériel ou logiciel

Qualité du Logiciel

Critères de qualité

- **Validité** : réponse aux attentes du client
- **Utilisabilité** : prise en main, facilité d'utilisation et d'apprentissage
- **Performance** : temps de réponse, complexité des algos, fluidité.
- **Fiabilité** : robustesse, sûreté et tolérance aux pannes
- **Sécurité** : intégrité des données et protection des accès
- **Maintenabilité** : facilité à corriger ou transformer le logiciel
- **Portabilité** : changement d'environnement matériel ou logiciel
- **Interopérabilité** : interagir en synergie avec d'autres logiciels

Cycle de vie

Processus de développement logiciel

	Effort	Erreurs	Coût de la maintenance
Spécification	15 %	55 %	80 %
Conception	15 %	25 %	15 %
Programmation	20 %	10 %	2 %
V&V	50 %	10 %	3 %

Cycle de vie

Processus de développement logiciel

	Effort	Erreurs	Coût de la maintenance
Spécification	15 %	55 %	80 %
Conception	15 %	25 %	15 %
Programmation	20 %	10 %	2 %
V&V	50 %	10 %	3 %

Plus de 90 % sur des logiciels critiques !

Cycle de vie

Processus de développement logiciel

	Effort	Erreurs	Coût de la maintenance
Spécification	15 %	55 %	80 %
Conception	15 %	25 %	15 %
Programmation	20 %	10 %	2 %
V&V	50 %	10 %	3 %

Plus de 90 % sur des logiciels critiques !

Développeur expérimenté ⇒ 163 fautes / 1000 instructions LOC

[B. Beizer Software Testing Techniques 1990]

Cycle de vie

Standish group, Chaos Report 2015

Softwares	Developers	Duration	Size (LOC)	Successful	Challenged	Failed
Grand	2K - 5K	5 - 10 yrs.	> 1M	6 %	51 %	43 %
Large	100 - 1K	4 - 5 yrs.	100K - 1M	11 %	59 %	30 %
Medium	5 - 20	2 - 3 yrs.	50K- 100K	12 %	62 %	26 %
Moderate	2 - 5	1 - 2 yrs.	3K - 20K	24 %	64 %	12 %
Small	1	1 - 6 mos.	1K - 2K	61 %	32 %	7 %

Successful: OnTime, OnBudget, OnTarget || 50,000 software projects

Cycle de vie

Standish group, Chaos Report 2015

Softwares	Developers	Duration	Size (LOC)	Successful	Challenged	Failed
Grand	2K - 5K	5 - 10 yrs.	> 1M	6 %	51 %	43 %
Large	100 - 1K	4 - 5 yrs.	100K - 1M	11 %	59 %	30 %
Medium	5 - 20	2 - 3 yrs.	50K- 100K	12 %	62 %	26 %
Moderate	2 - 5	1 - 2 yrs.	3K - 20K	24 %	64 %	12 %
Small	1	1 - 6 mos.	1K - 2K	61 %	32 %	7 %

Successful: OnTime, OnBudget, OnTarget || 50,000 software projects

Softwares	Successful	Challenged	Failed
Banking	30 %	55 %	15 %
Financial	29 %	56 %	15 %
Government	21 %	55 %	24 %
Healthcare	29 %	53 %	18 %
Manufacturing	28 %	53 %	19 %
Retail	35 %	49 %	16 %
Services	29 %	52 %	19 %
Telecom	24 %	53 %	23 %
Other	29 %	48 %	23 %

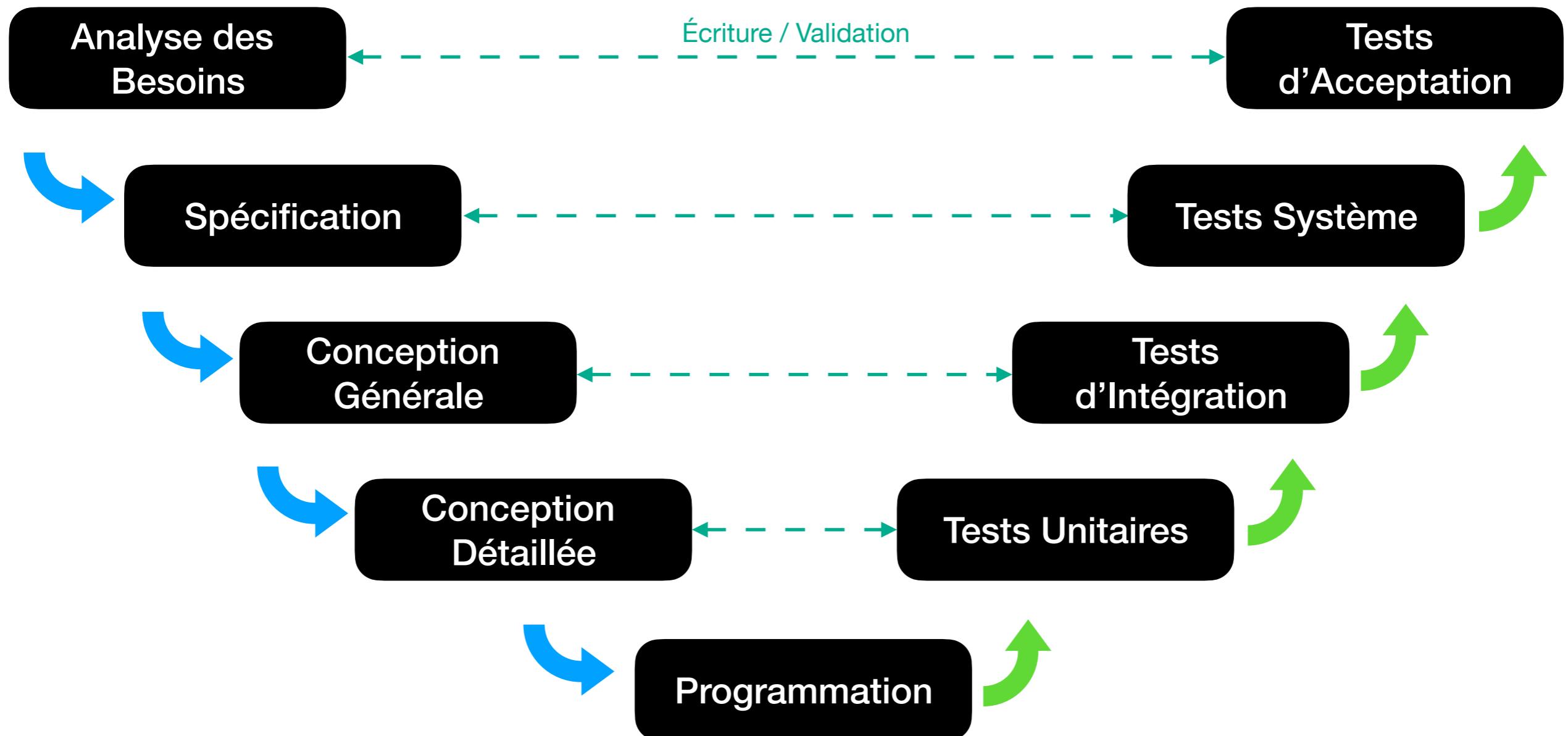
Modèles et méthodes

Standish group, Chaos Report 2016

SIZE	METHOD	SUCCESSFUL	CHALLENGED	FAILED
ALL SIZE PROJECTS	AGILE	40%	51%	9%
	WATERFALL	13%	60%	27%
	OTHER (AINO)	32%	52%	16%
LARGE SIZE PROJECTS	AGILE	21%	60%	19%
	WATERFALL	4%	56%	40%
MEDIUM SIZE PROJECTS	AGILE	30%	60%	10%
	WATERFALL	9%	69%	22%
SMALL SIZE PROJECTS	AGILE	55%	40%	5%
	WATERFALL	47%	40%	13%

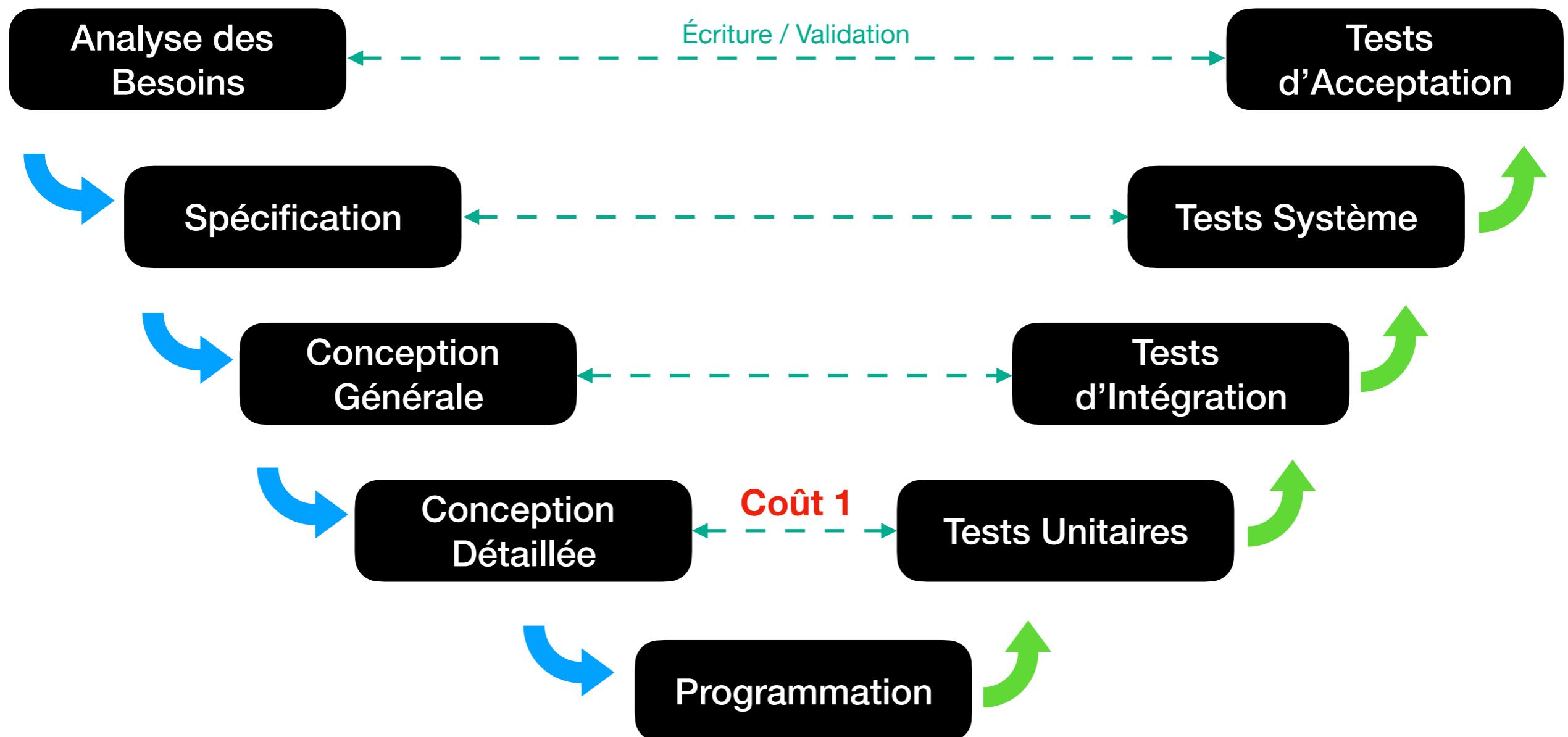
Coût de VW

Modèle en V (V-Shaped Model)



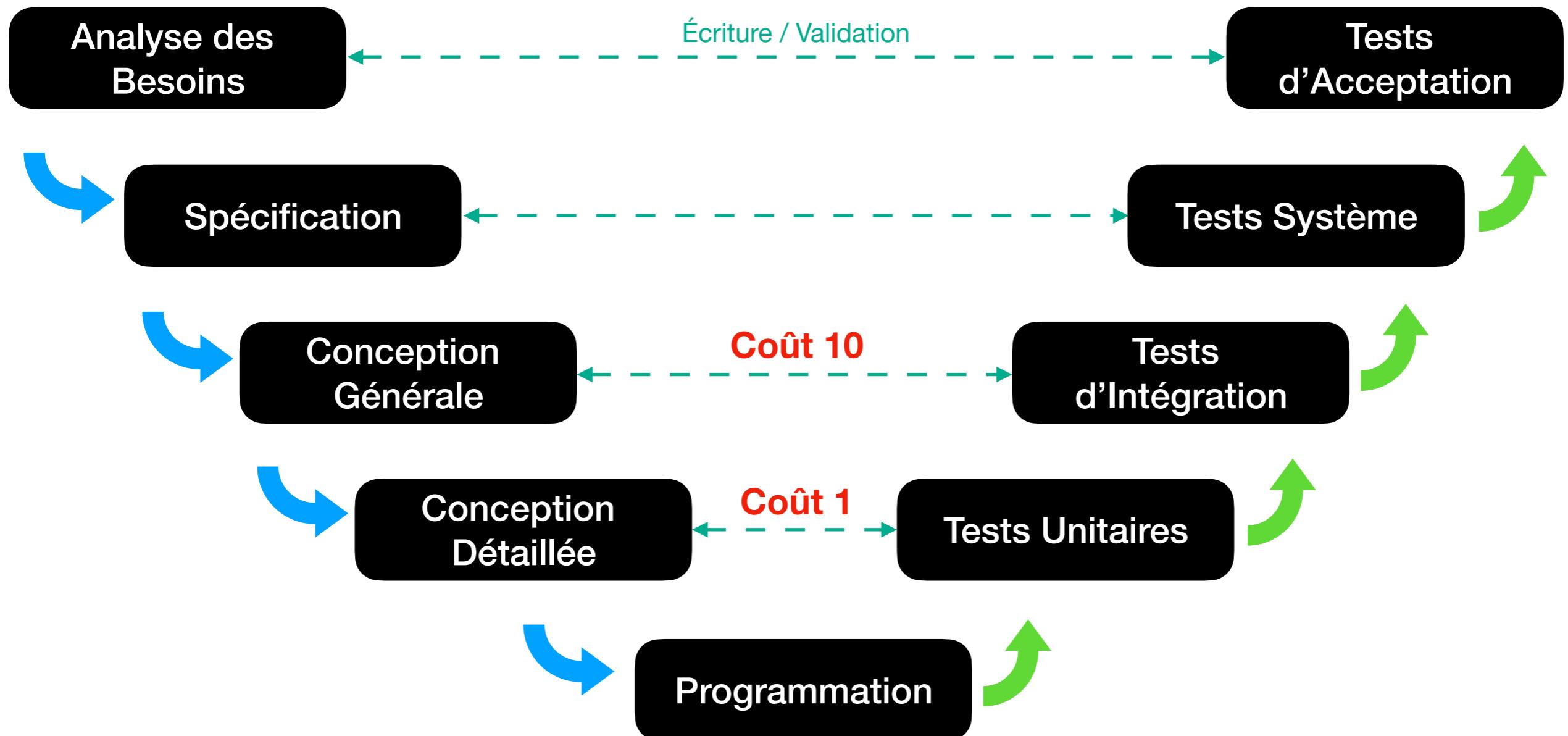
Coût de VW

Modèle en V (V-Shaped Model)



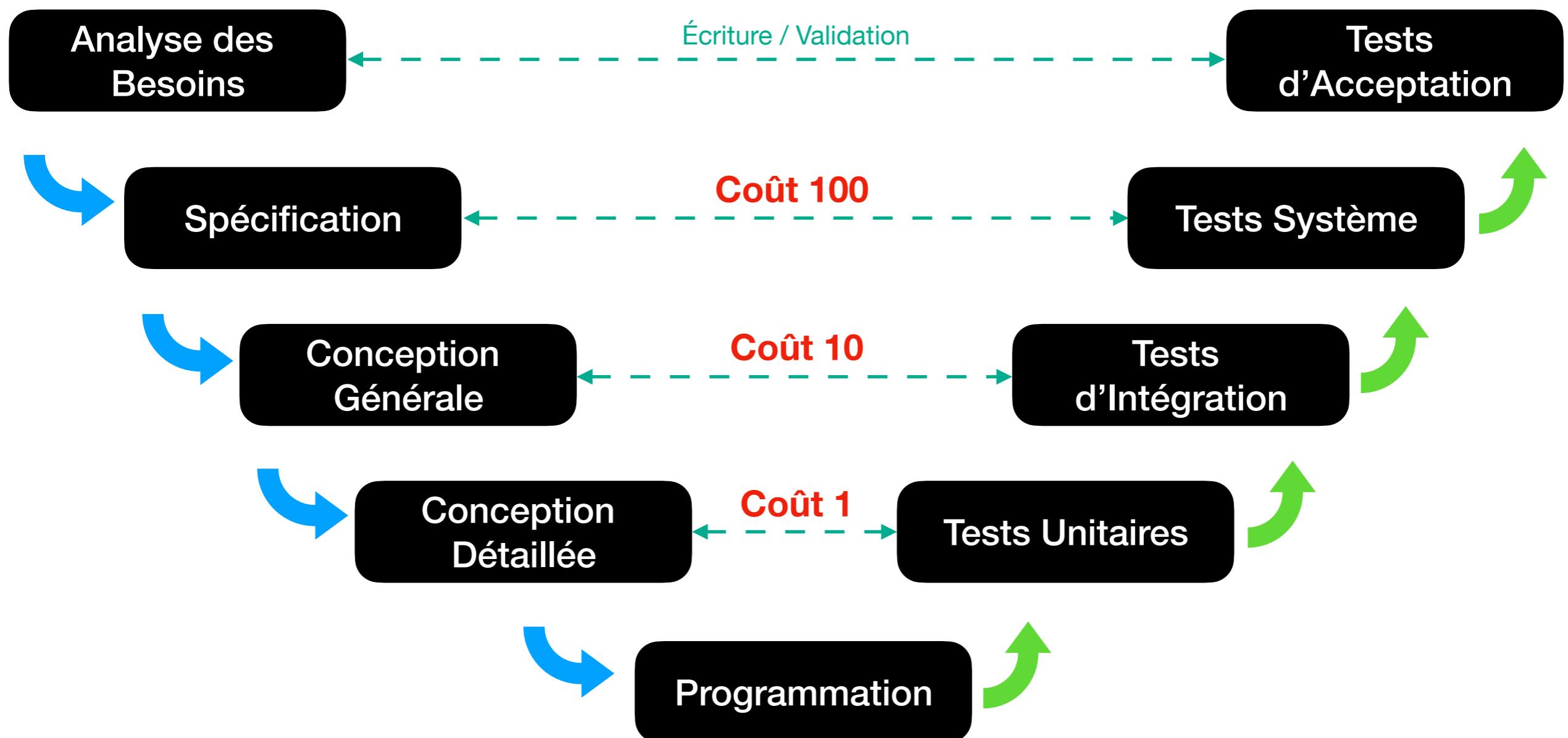
Coût de VW

Modèle en V (V-Shaped Model)



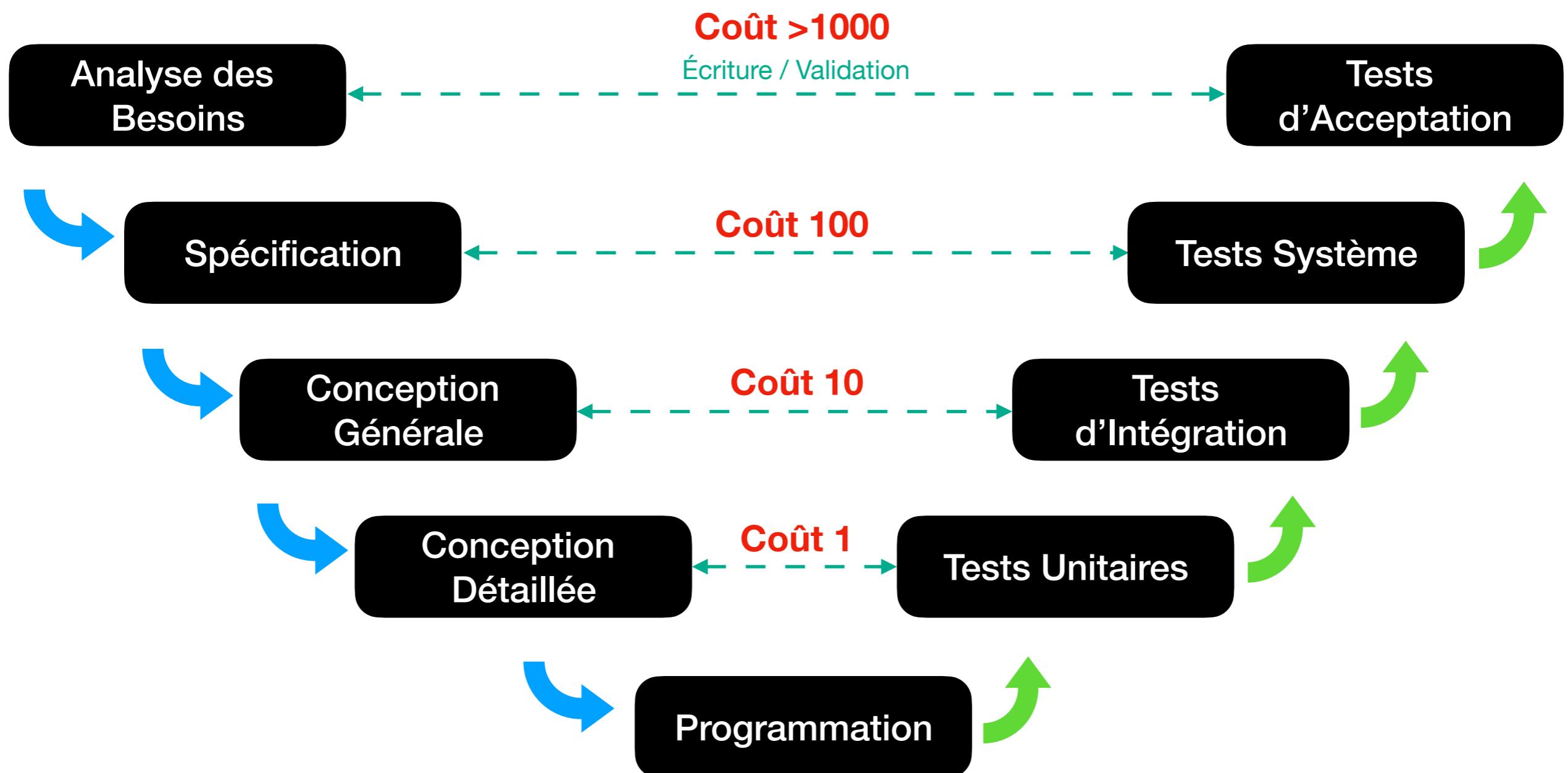
Coût de VW

Modèle en V (V-Shaped Model)



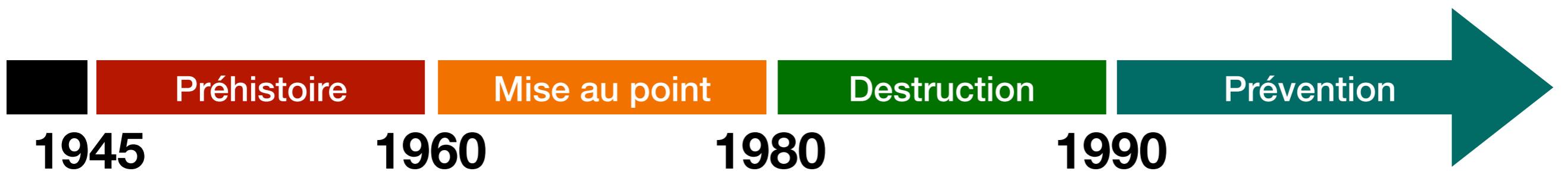
Coût de VW

Modèle en V (V-Shaped Model)



Problématique du test

Historique



Époques test

Mise au Point et Destruction

- 1960 - 1980 :
 - Chaine causale : ERREUR \Rightarrow FAUTE \Rightarrow DÉFAILLANCE
 - Deux activités de base :
 - Test : Détection des défaiillances
 - Mise au point : Localisation des fautes et correction des erreurs
- 1980 - 1990 :

Époques test

Mise au Point et Destruction

- 1960 - 1980 :
 - Chaine causale : ERREUR \Rightarrow FAUTE \Rightarrow DÉFAILLANCE
 - Deux activités de base :
 - Test : Détection des défaiillances
 - Mise au point : Localisation des fautes et correction des erreurs
- 1980 - 1990 :

« Testing is the process of executing a program with
the intent of finding errors »

[G. Myers The Art of Software Testing 1979]

Types de test

Test logiciel

Types de test

Test logiciel

- **Stratégie (dimension cycle de vie) :**

- Unitaire
- Intégration
- Système

Types de test

Test logiciel

- **Stratégie (dimension cycle de vie) :**
 - Unitaire
 - Intégration
 - Système
- **Critère d'évaluation (dimension qualité du logiciel) :**
 - Conformité
 - Robustesse
 - Performance
 - Sécurité

Types de test

Test logiciel

- **Stratégie (dimension cycle de vie) :**
 - Unitaire
 - Intégration
 - Système
- **Critère d'évaluation (dimension qualité du logiciel) :**
 - Conformité
 - Robustesse
 - Performance
 - Sécurité
- **Accessibilité :**
 - Boite noire
 - Boite blanche
 - Boite grise

Test logiciel

Définition

«Program Testing can be used to prove the presence
of bugs, but never their absence »

[Dijkstra 74]

Test logiciel

Définition

«Program Testing can be used to prove the presence
of bugs, but never their absence »

[Dijkstra 74]

- Tester : exécuter un programme P pour mettre en évidence la présence de fautes, par rapport à sa spécification F (**recherche de contre-exemple**)
- CT_i : Cas de test
- DT_i : Donnée de test
- ST_i : Sortie de test
- $CT_i = (DT_i, ST_i)$

Test logiciel

Définition

«Program Testing can be used to prove the presence of bugs, but never their absence »

[Dijkstra 74]

- Tester : exécuter un programme P pour mettre en évidence la présence de fautes, par rapport à sa spécification F (**recherche de contre-exemple**)
- CT_i : Cas de test
- DT_i : Donnée de test
- ST_i : Sortie de test
- $CT_i = (DT_i, ST_i)$

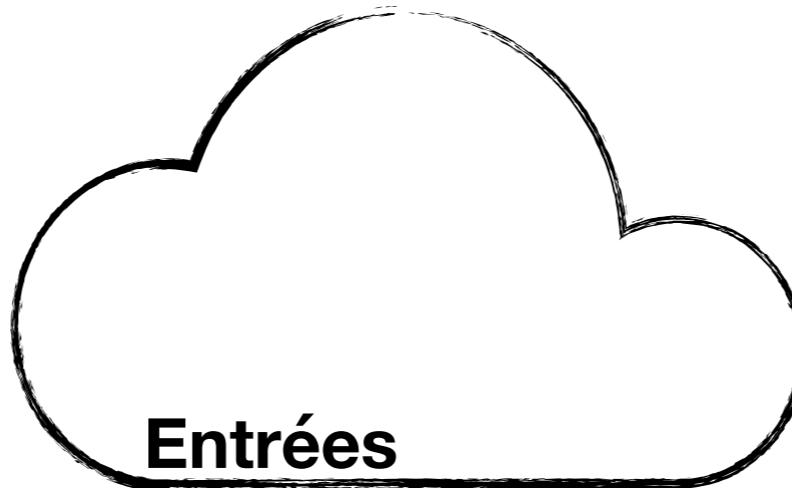
$$\exists CT_i : P(DT_i) = ST_i \wedge F(DT_i) \neq ST_i ?$$

Test logiciel

Processus

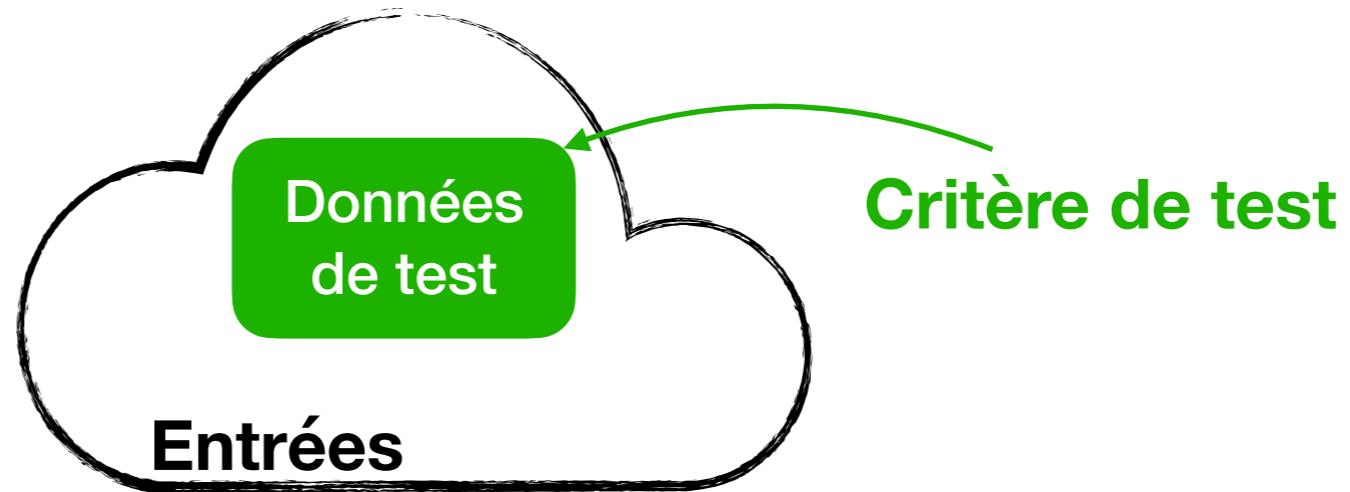
Test logiciel

Processus



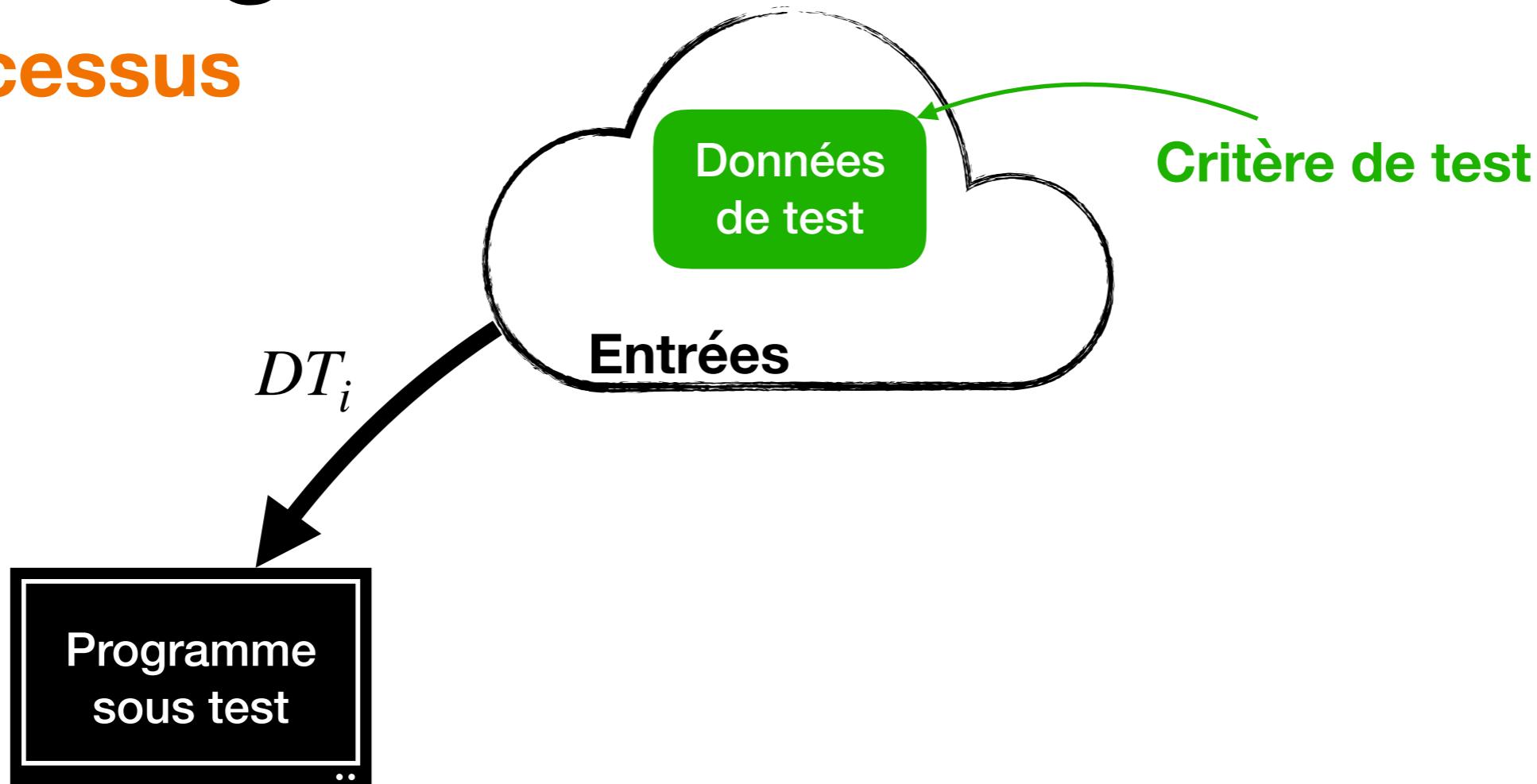
Test logiciel

Processus



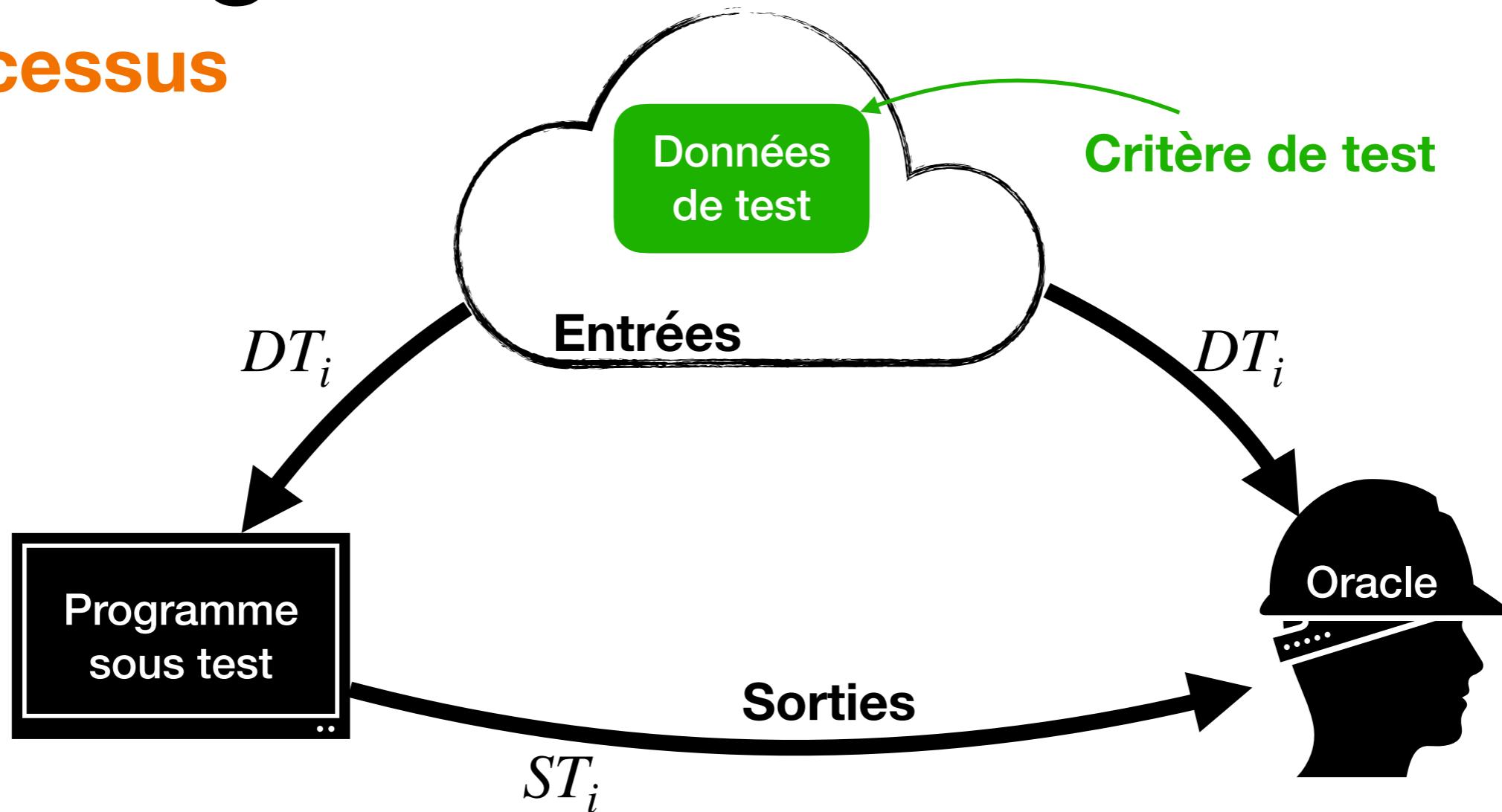
Test logiciel

Processus



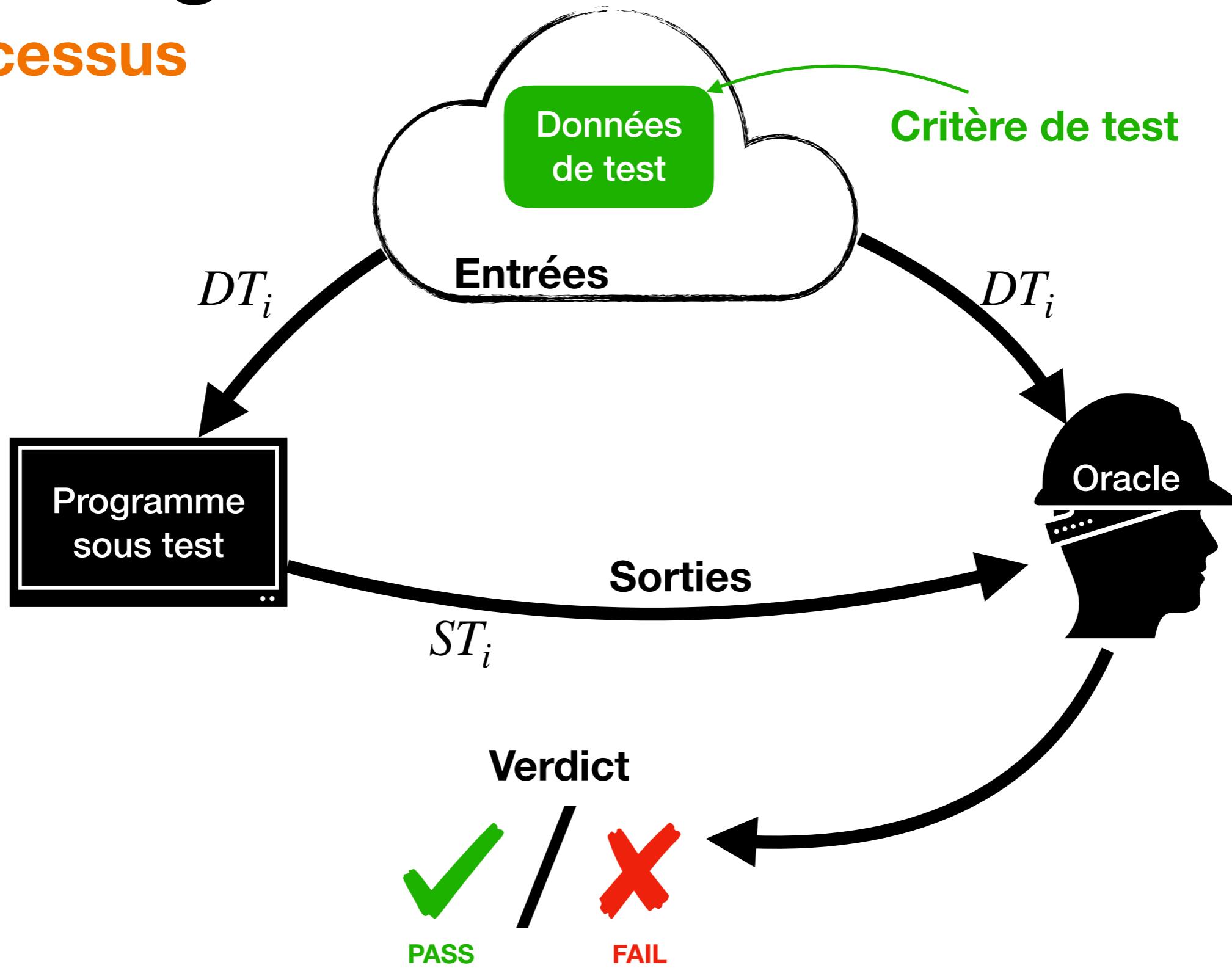
Test logiciel

Processus



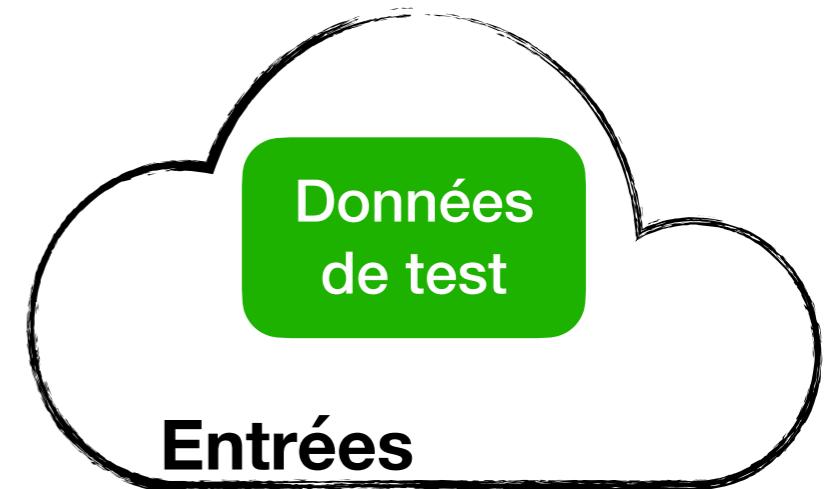
Test logiciel

Processus



Critères de test

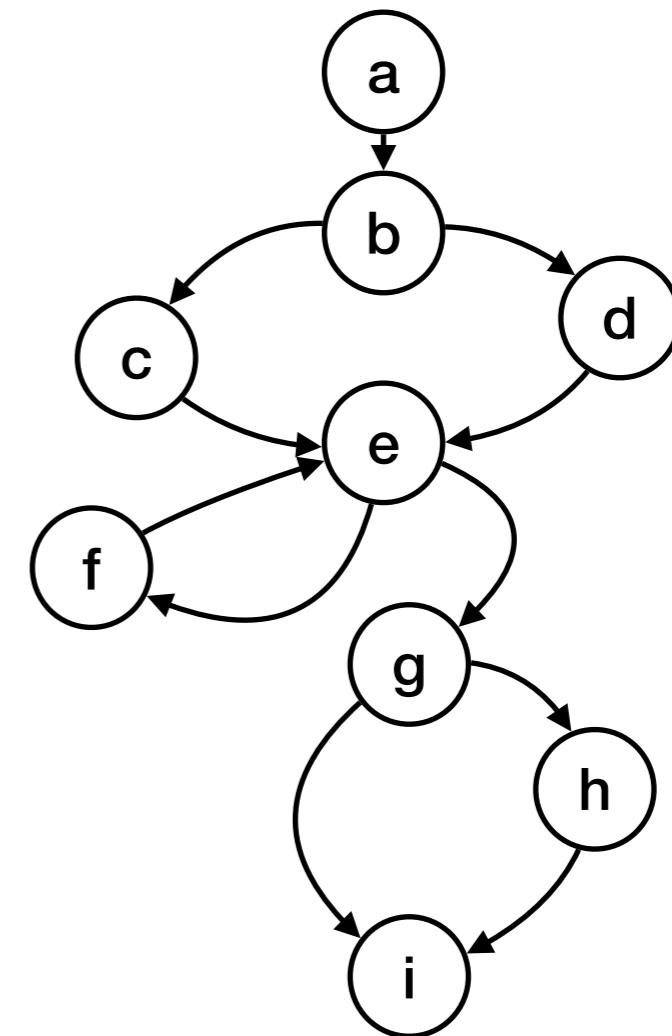
Sélection des tests



- Efficacité du test dépend fortement de *la qualité* des DT
- **Objectif** : avoir un ensemble minimal de DT qui ne rate aucune **faute**
- DT \Rightarrow échantillon représentatif, pertinent et pas de redondance
- **Critère de test** : « qu'est ce qu'un bon jeu de test ? »
 - Guide pour choisir les DT pertinentes
 - Bonne corrélation au pouvoir de détection des fautes
 - Concis et automatisable

Graphe de Flot de Contrôle (GFC)

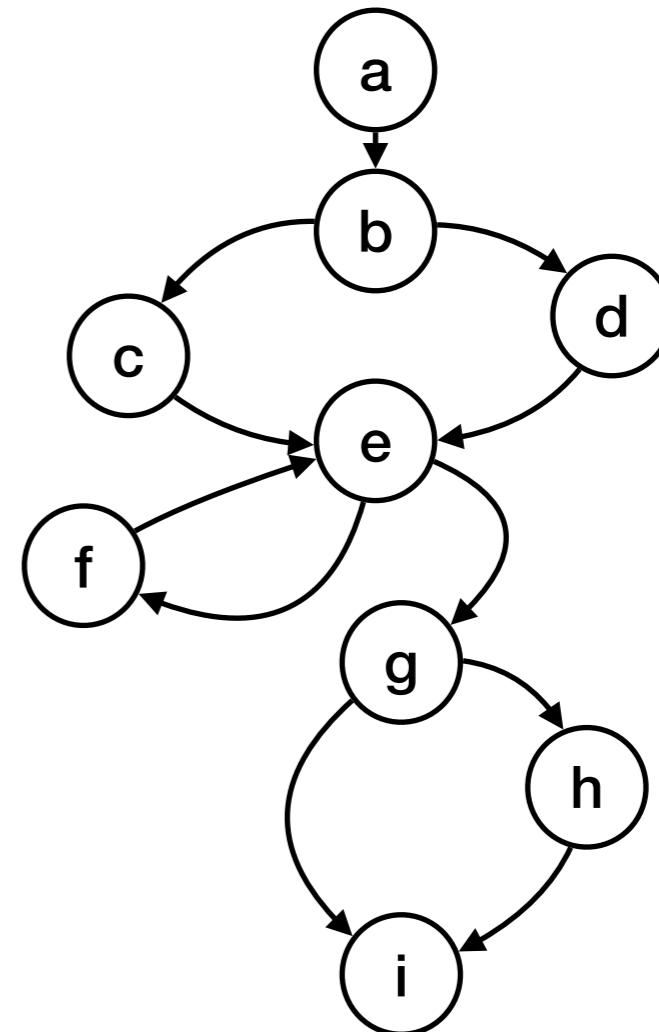
exemple



Graphe de Flot de Contrôle (GFC)

exemple

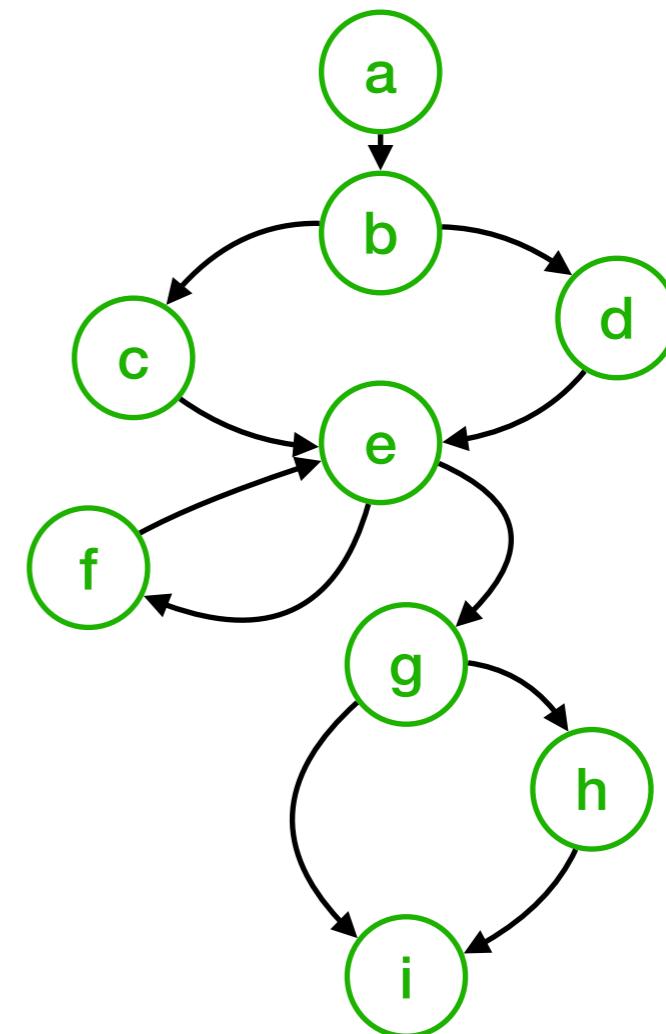
```
double power(int x, int y) {  
  
    int i, p;  
    i = 0;  
    double z = 1;  
    if (x < 0)  
        p = -x;  
    else  
        p = x;  
    while (i < p) {  
        z = z * x;  
        i = i + 1;  
    }  
    if (y < 0)  
        x = 1 / x;  
    return x;  
}
```



Critères boite blanche

Critères de couverture

Tous les noeuds

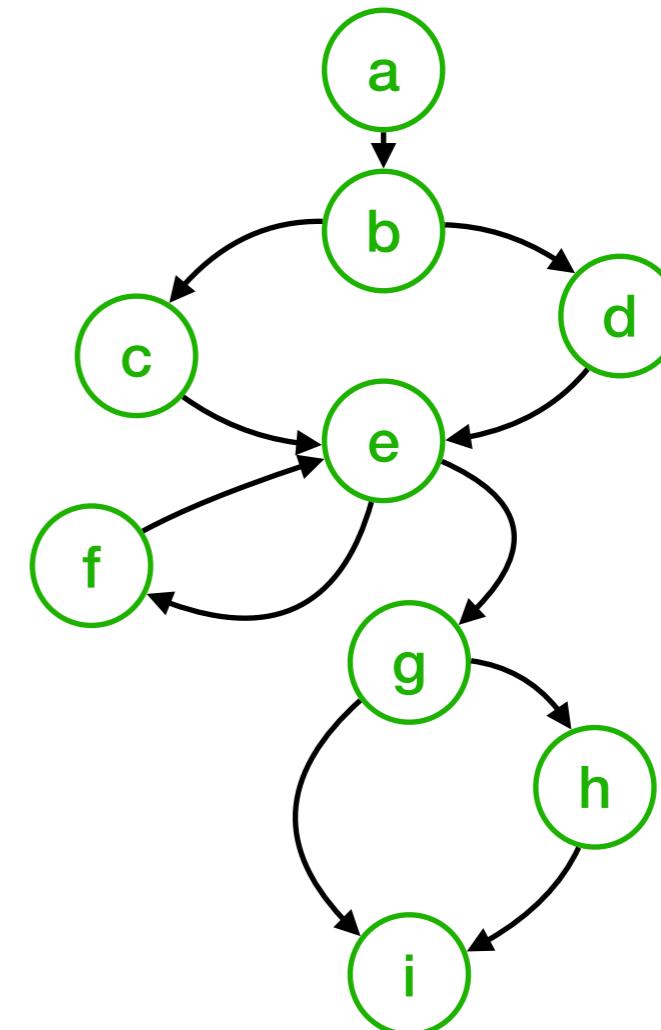


Critères boîte blanche

Critères de couverture

Tous les noeuds

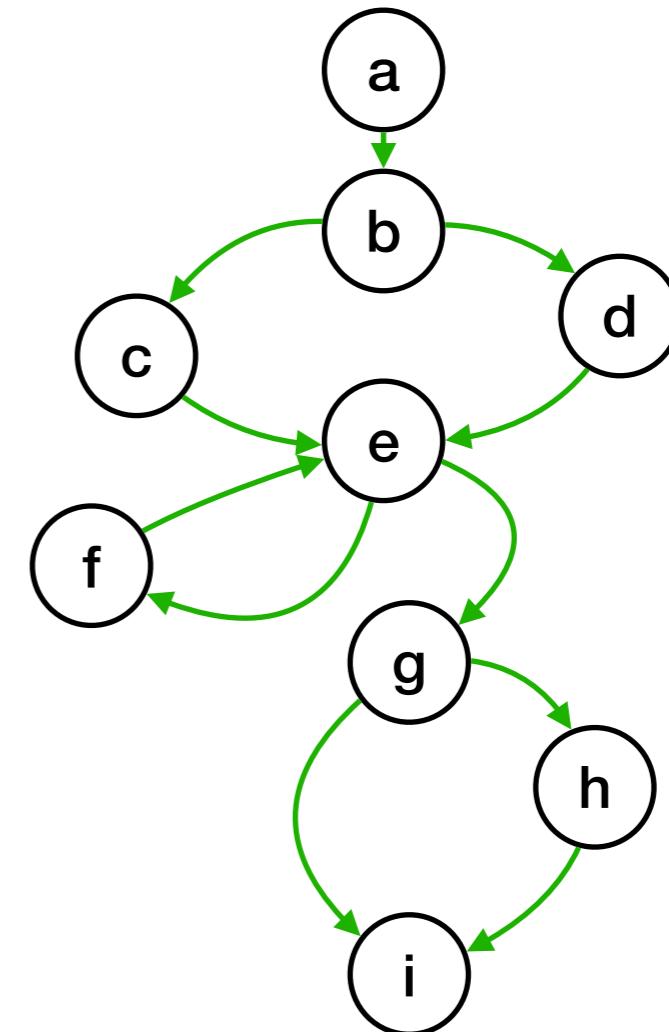
```
double power(int x, int y) {  
  
    int i, p;  
    i = 0;  
    double z = 1;  
    if (x < 0)  
        p = -x;  
    else  
        p = x;  
    while (i < p) {  
        z = z * x;  
        i = i + 1;  
    }  
    if (y < 0)  
        x = 1 / x;  
    return x;  
}
```



Critères boite blanche

Critères de couverture

Tous les arcs / décisions

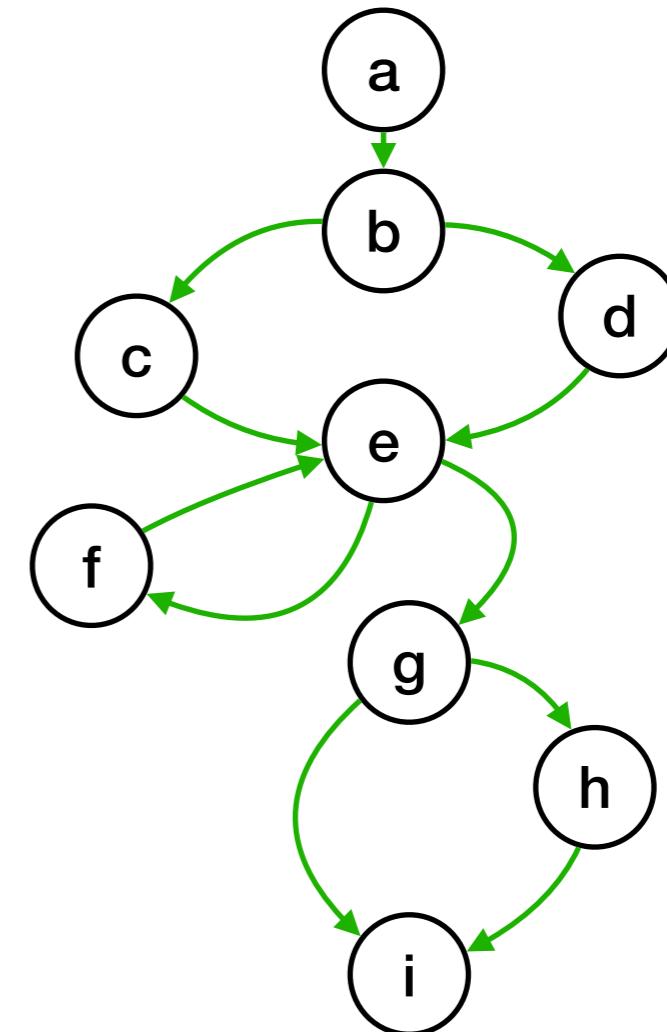


Critères boîte blanche

Critères de couverture

Tous les arcs / décisions

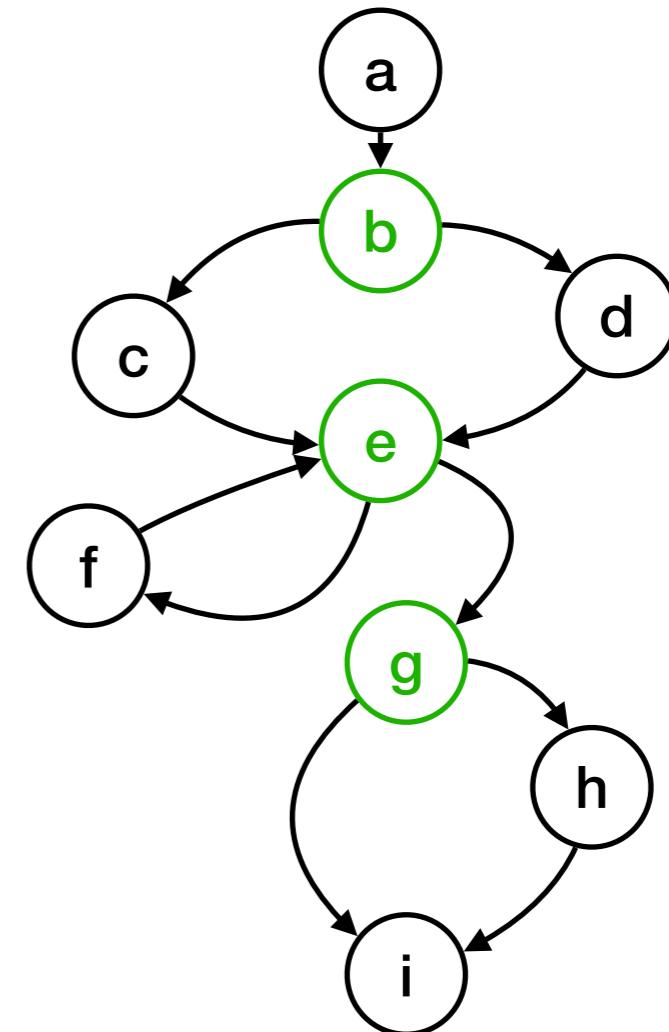
```
double power(int x, int y) {  
  
    int i, p;  
    i = 0;  
    double z = 1;  
    if (x < 0)  
        p = -x;  
    else  
        p = x;  
    while (i < p) {  
        z = z * x;  
        i = i + 1;  
    }  
    if (y < 0)  
        x = 1 / x;  
    return x;  
}
```



Critères boîte blanche

Critères de couverture

Toutes les conditions

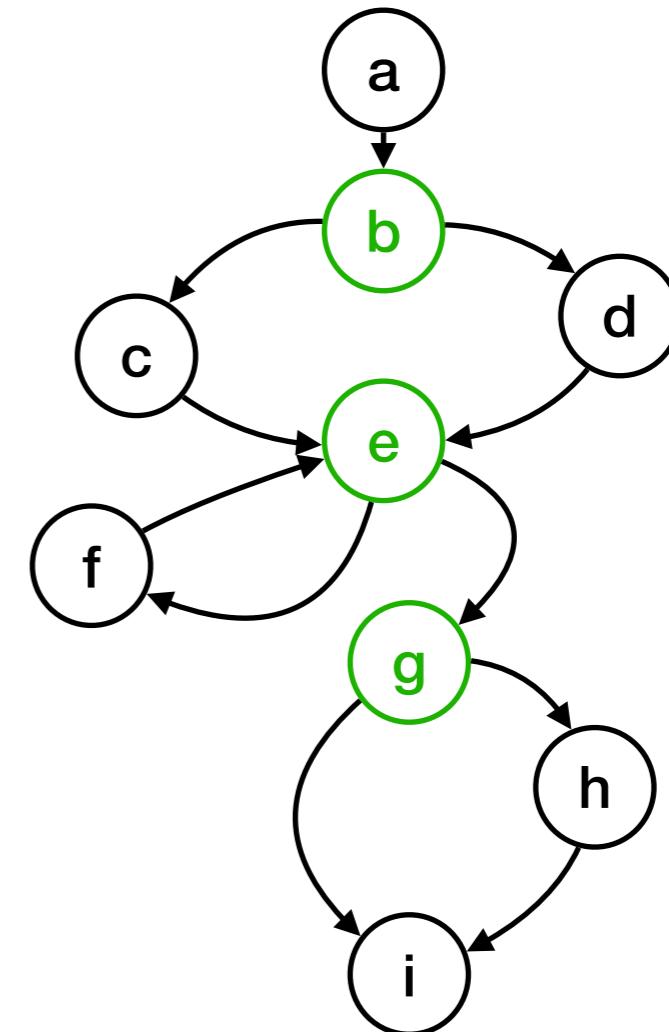


Critères boîte blanche

Critères de couverture

Toutes les conditions

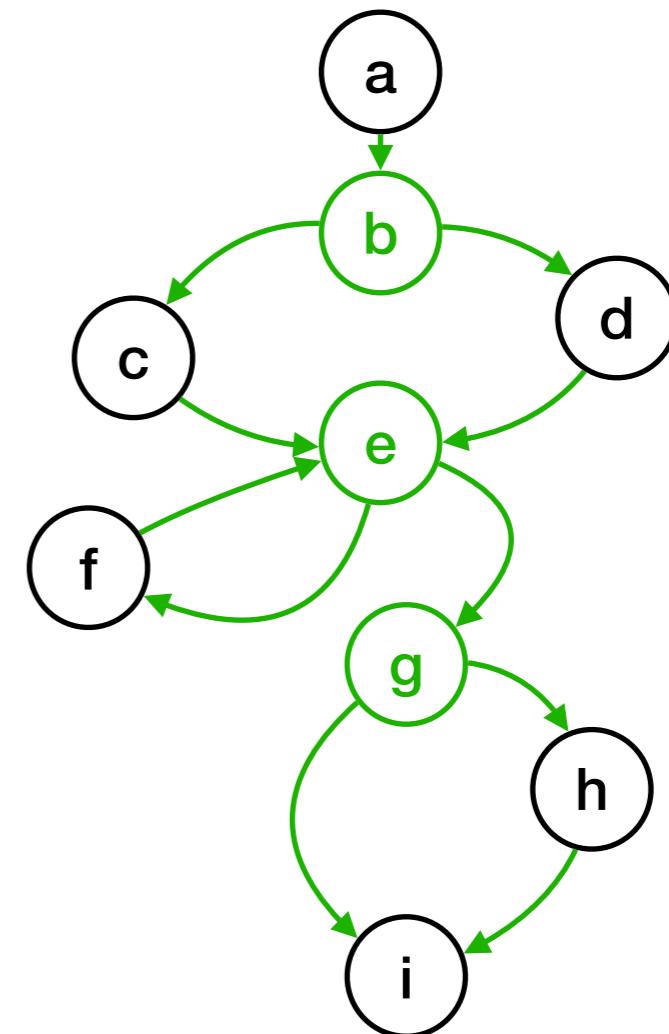
```
double power(int x, int y) {  
    int i, p;  
    i = 0;  
    double z = 1;  
    if (x < 0)  
        p = -x;  
    else  
        p = x;  
    while (i < p) {  
        z = z * x;  
        i = i + 1;  
    }  
    if (y < 0)  
        x = 1 / x;  
    return x;  
}
```



Critères boîte blanche

Critères de couverture

Toutes les conditions / décisions

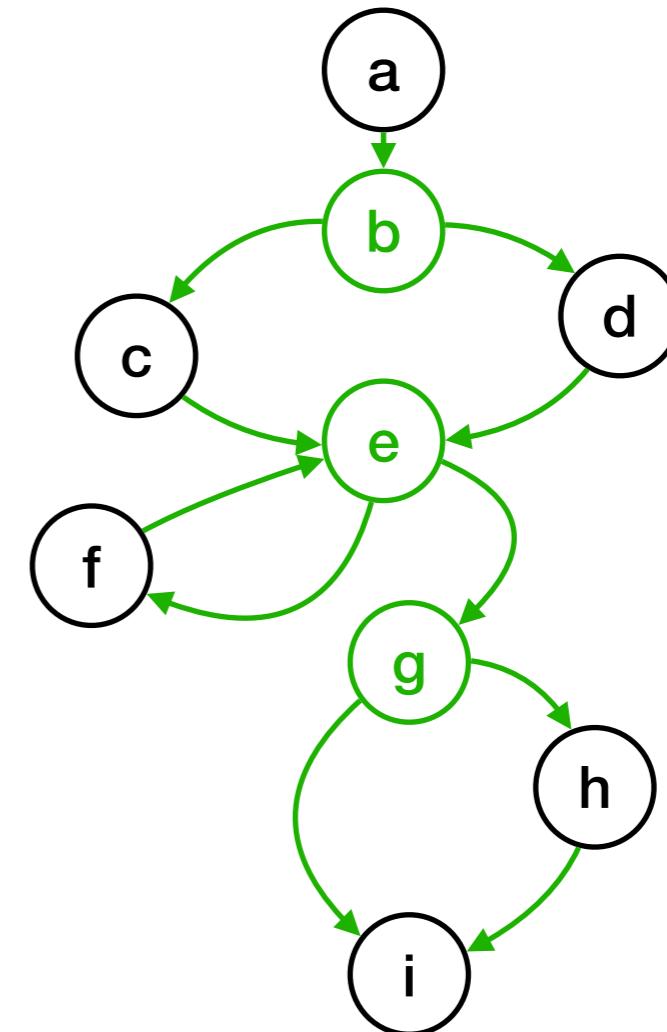


Critères boîte blanche

Critères de couverture

Toutes les conditions / décisions

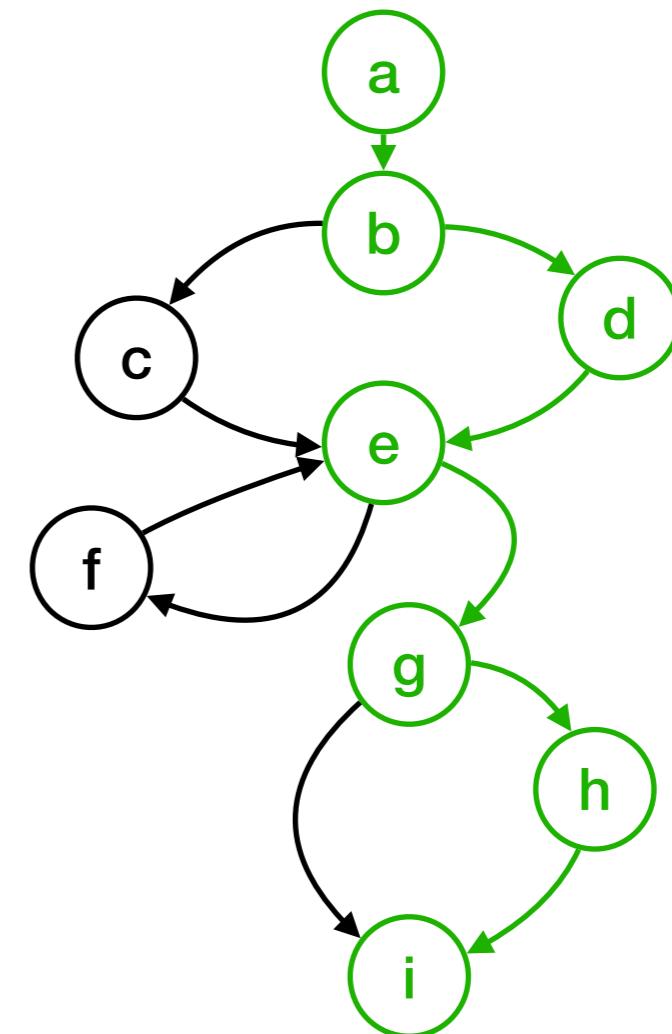
```
double power(int x, int y) {  
  
    int i, p;  
    i = 0;  
    double z = 1;  
    if (x < 0)  
        p = -x;  
    else  
        p = x;  
    while (i < p) {  
        z = z * x;  
        i = i + 1;  
    }  
    if (y < 0)  
        x = 1 / x;  
    return x;  
}
```



Critères boîte blanche

Critères de couverture

Tous les chemins

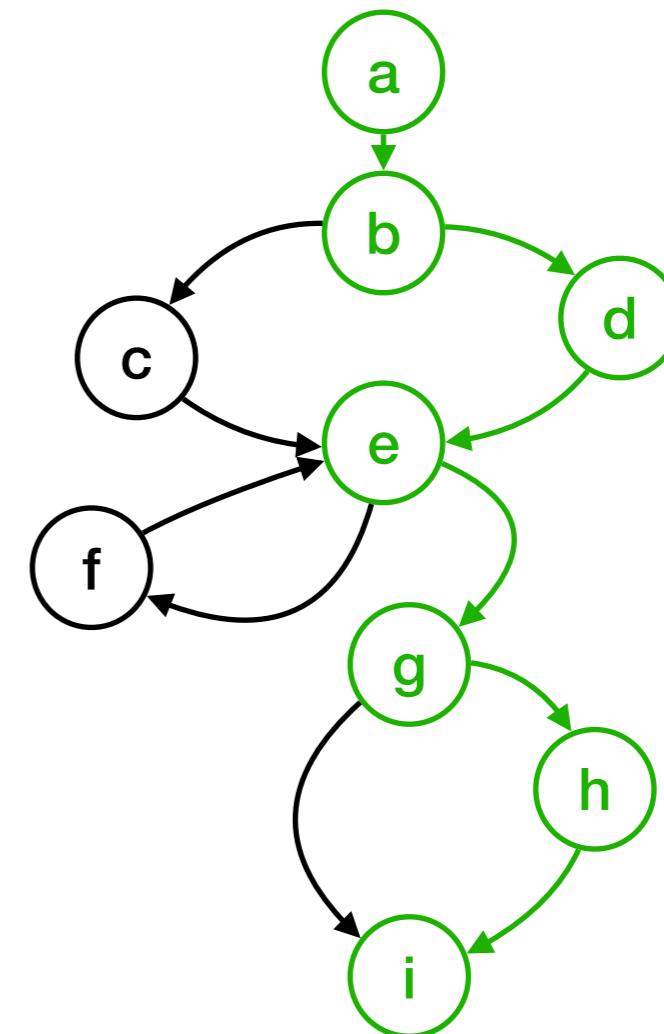


Critères boîte blanche

Critères de couverture

Tous les chemins

```
double power(int x, int y) {  
    int i, p;  
    i = 0;  
    double z = 1;  
    if (x < 0)  
        p = -x;  
    else  
        p = x;  
    while (i < p) {  
        z = z * x;  
        i = i + 1;  
    }  
    if (y < 0)  
        x = 1 / x;  
    return x;  
}
```



Méthode à contraintes

Génération de réseau de contraintes

1. Programme impératif $P \Rightarrow$ Réseau de contraintes $N = (X, D, C)$
2. Elément sélectionné $e \Rightarrow$ contraintes à satisfaire c_e
3. Résoudre $N' = (X, D, C \wedge c_e)$

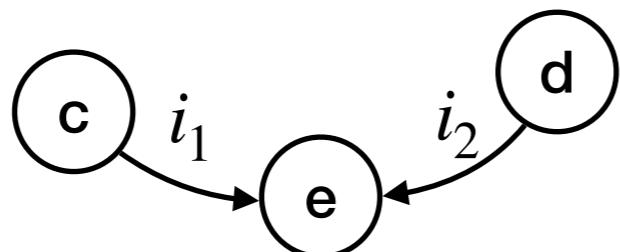
Génération de réseau de contraintes

Programme P => Programme PSSA

- Traduire chaque instruction en une contraintes en utilisant le renommage des variables (forme SSA for static single assignment) :

- $i = i + 1 \Rightarrow i_2 = i_1 + 1$

- Jointure :



$$i_3 = \phi(i_1, i_2)$$

Génération de réseau de contraintes

Programme $P \Rightarrow$ Programme P_{SSA}

- Traduire chaque instruction en une contraintes en utilisant le renommage des variables (forme SSA for static single assignment) :

P

P_{SSA}

Génération de réseau de contraintes

Programme P => Programme P_{SSA}

- Traduire chaque instruction en une contraintes en utilisant le renommage des variables (forme SSA for static single assignment) :

P

P_{SSA}

```
double power(int x_1, int y_1) {  
    int i_1, p_1;  
    i_1 = 0;  
    double z_1 = 1;  
    if (x_1 < 0)  
        p_1 = -x_1;  
    else  
        p_2 = x_1;  
    p_3= phi(p_1,p_2);  
    while (i_1 < p_3) {  
        z_2 = z_1 * x_1;  
        i_2 = i_1 + 1;  
    }  
    z_3= phi(z_1,z_2);  
    i_3=phi(i_1,i_2);  
    if (y_1 < 0)  
        x_2 = 1 / x_1;  
    x_3 = phi(x_1,x_2);  
    return x_3;  
}
```

Génération de réseau de contraintes

Programme P => Programme P_{SSA}

- Traduire chaque instruction en une contraintes en utilisant le renommage des variables (forme SSA for static single assignment) :

P

```
double power(int x, int y) {  
  
    int i, p;  
    i = 0;  
    double z = 1;  
    if (x < 0)  
        p = -x;  
    else  
        p = x;  
    while (i < p) {  
        z = z * x;  
        i = i + 1;  
    }  
    if (y < 0)  
        x = 1 / x;  
    return x;  
}
```

P_{SSA}

```
double power(int x_1, int y_1) {  
  
    int i_1, p_1;  
    i_1 = 0;  
    double z_1 = 1;  
    if (x_1 < 0)  
        p_1 = -x_1;  
    else  
        p_2 = x_1;  
    p_3 = phi(p_1, p_2);  
    while (i_1 < p_3) {  
        z_2 = z_1 * x_1;  
        i_2 = i_1 + 1;  
    }  
    z_3 = phi(z_1, z_2);  
    i_3 = phi(i_1, i_2);  
    if (y_1 < 0)  
        x_2 = 1 / x_1;  
    x_3 = phi(x_1, x_2);  
    return x_3;  
}
```

Génération de réseau de contraintes

Programme P => réseau de contraintes N

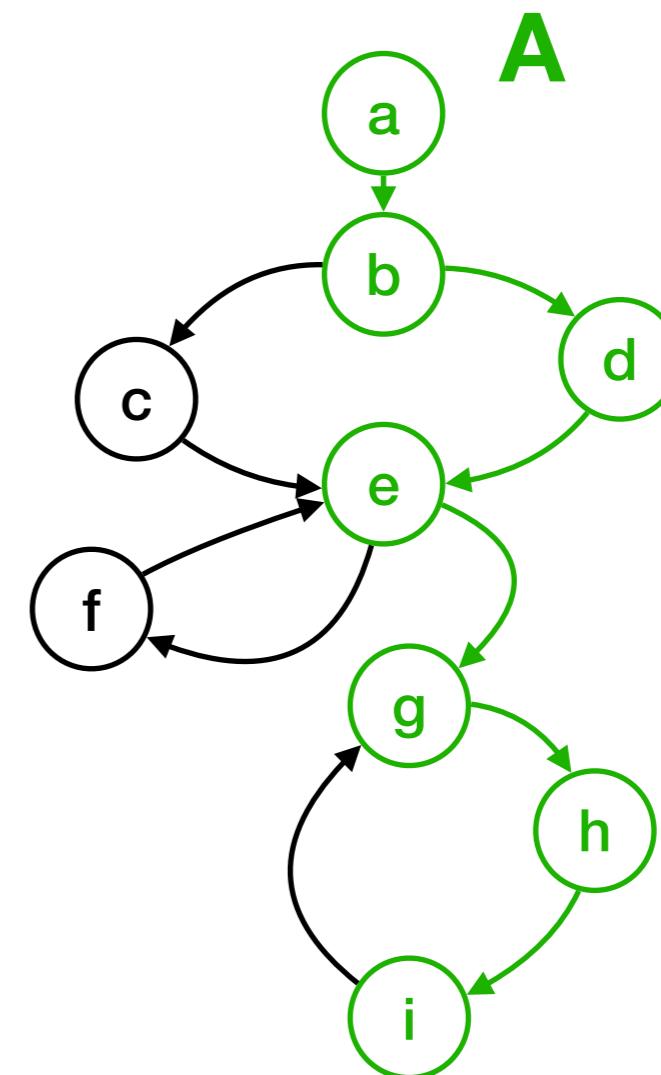
- Déclaration de variable :
 - $int\ i \Rightarrow i \in \{0, 2^{32} - 1\}$
- Affectation : $i_2 = i_1 + 1 \Rightarrow$ contrainte : $i_2 = i_1 + 1$
- Conditionnelle (SSA) : contrainte $ite(C_{cond}, V_1, V_2, V_3, C_{then}, C_{else})$
- Itération (SSA) : contrainte $w(C_{cond}, V_1, V_2, V_3, C_{body})$

Génération de réseau de contraintes

Donnez le réseau de contraintes du chemin A

P_{SSA}

```
double power(int x_1, int y_1) {  
    int i_1, p_1;  
    i_1 = 0;  
    double z_1 = 1;  
    if (x_1 < 0)  
        p_1 = -x_1;  
    else  
        p_2 = x_1;  
    p_3 = phi(p_1, p_2);  
    while (i_1 < p_3) {  
        z_2 = z_1 * x_1;  
        i_2 = i_1 + 1;  
    }  
    z_3 = phi(z_1, z_2);  
    i_3 = phi(i_1, i_2);  
    if (y_1 < 0)  
        x_2 = 1 / x_1;  
    x_3 = phi(x_1, x_2);  
    return x_3;  
}
```



Many

Thanks to

- Sébastien Bardin, CEA, Paris-Saclay
- Arnaud Gotlieb, SIMULA Research Lab., Oslo, Norway
- Yahia Lebbah, Oran 1, Algeria
- Delphine Longuet, LRI, Paris-Sud
- Christine Solnon, CITI, INSA Lyon