Test boîte noire

Boîte noire ou test fonctionnel

Principes de base

- Basé sur la définition de ce qu'est la spécification d'un programme en opposition à sa structure:
  - On considère le logiciel comme une boite noire, soit du point de vue entrée/sortie, sans aucune considération de sa structure interne (i.e., comment il est implémenté).
  - Les cas de tests sont conçus à partir d'une spécification (i.e., sans égard au code, si celui-ci est disponible).
- Le niveau de rigueur (et de détail) dans les spécifications est particulièrement important et influe sur la facilité à catégoriser les données d'entrée et anticiper les sorties:
  → la génération des cas de tests et des oracles.

Les tests intuitifs ne sont pas fiables

- Tests intuitifs ou *ad-hoc*
  - *Exemple:* Un module accepte une donnée d'entrée dont le domaine de validité est les nombres entiers entre 1 et 100 inclusivement.
  - Le testeur choisit (intuitivement) les valeurs 55, 24 et 6 pour ses tests.
- Préoccupations
  - Ces trois valeurs sont-elles suffisantes pour démontrer que le module rencontre ses spécifications?
  - Devrait-on utiliser plus (ou moins) de valeurs pour faire une utilisation optimale des ressources ?
  - Est-ce qu'il y a d'autres valeurs plus susceptibles de révéler des défauts?
  - Devrait-on utiliser des valeurs de test à l'extérieur du domaine de validité (i.e., pas un entier, zéro, négatif, >100,...) ?
- Principal avantage: génération rapide des données d'entrée
- Principal inconvénient: peu susceptible de mener à des données de tests "efficaces" (i.e., à haute probabilité de révéler des défauts).

Plan

- Partitionnement en classes d'équivalence
- Analyse de la valeur limite
- Test catégorie-partition
- Tables de décision
- Graphes de cause-à-effet
- Fonctions logiques

Partitionnement par classe d'équivalence

Test par classes d'équivalence

- Motivation : nous aimerions obtenir un jeu de test **complet** et nous souhaiterions éviter la **redondance**.
- Classe d'équivalence : partage de l'ensemble des données.
- Ensemble des données entièrement couvert → complet.
- Classes séparées : éviter la redondance.
- Cas de tests : un élément de chacune des classes d'équivalence.
- Mais les classes d'équivalence doivent être choisies prudemment …
- Estimer le comportement probable du système sous-jacent …

Test par classes d'équivalence

- Hypothèse sur la faute
  - Test par classes d'équivalence faibles (WECT): une faute à la fois
  - Test par classes d'équivalence forts (SECT): plusieurs fautes possibles

- Valeurs invalides
  - Tests par classes d'équivalence faibles robustes (WRECT)
  - Tests par classes d'équivalence forts robustes (SRECT)

Test par classes d'équivalence faibles/forts

- Trois variables d'entrée de domaines A, B, C :
  - A = A1 u A2 u A3 où a_n in An
  - B = B1 u B2 u B3 u B4 où b_n in Bn
  - C = C1 u C2 où c_n in Cn

- Test par classes d'équivalence faibles (WECT) : choisir une forme de variable d'entrée pour chacune des classes d'équivalences :
  - **max(|A|, |B|, |C|)** cas de tests

- Test par classes d'équivalence forts (SECT) : basé sur le produit Cartésien des sous-ensembles des partitions (A x B x C), i.e., tester toutes les interactions de classes :
  - **|A| x |B| x |C|** cas de tests

Cas de test WECT

Cas de test	A	B	C
WE1	A1	B1	C1
WE2	A2	B2	C2
WE3	A3	B3	C1
WE4	A1	B4	C2

*Cas de test SECT ➔ 3*4*2*

Cas de Test	A	B	C
SE1	A1	B1	C1
SE2	A1	B1	C2
SE3	A1	B2	C1
SE4	A1	B2	C2
SE5	A1	B3	C1
SE6	A1	B3	C2
SE7	A1	B4	C1
SE8	A1	B4	C2
SE9	A2	B1	C1
SE10	A2	B1	C2
SE11	A2	B2	C1
SE12	A2	B2	C2
SE13	A2	B3	C1
SE14	A2	B3	C2
SE15	A2	B4	C1
SE16	A2	B4	C2
SE17	A3	B1	C1
SE18	A3	B1	C2
SE19	A3	B2	C1
SE20	A3	B2	C2
SE21	A3	B3	C1
SE22	A3	B3	C2
SE23	A3	B4	C1
SE24	A3	B4	C2

Test par classes d'équivalence faibles/forts robustes

- Test par classes d'équivalence faibles robustes (WRECT) :
  - choisir une forme de variable d'entrée pour chacune des classes d'équivalences valides.
  - Une classe d'équivalence invalide à la fois pour un cas de test, i.e., toutes les autres classes d'équivalence sont valides

- Test par classes d'équivalence forts robustes (SRECT) : basé sur le produit Cartésien de toutes les classes d'équivalence, i.e., valides et invalides
- Problème avec les tests par classes d'équivalence robustes
  - Souvent les spécifications ne sont pas explicites pour les données invalides

Exemple `NextDate`

- *`NextDate`* est une fonction avec trois variables : `MOIS`, `JOUR`, `ANNEE`. Elle retourne la date du jour après la date donnée. Années valides: 1812-2012.

- Sommaire du traitement :
  - si ce n'est pas le dernier jour du mois, la dernière fonction date augmente simplement la valeur `JOUR`.
  - À la fin du mois, le jour suivant est 1 et la valeur `MOIS` est augmentée.
  - À la fin d'une année, les deux valeurs `JOUR` et `MOIS` sont remis à 1, et la valeur `ANNEE` est augmentée.
  - Finalement, le problème de l'année bissextile rend intéressante la détermination du dernier jour d'un mois.

Classes d'équivalence `NextDate`

- M1 = { MOIS: MOIS à 30 jours}
- M2 = { MOIS: MOIS à 31 jours}
- M3 = { MOIS: MOIS est Fevrier}
- D1 = {JOUR: 1<= JOUR <= 28}
- D2 = {JOUR: JOUR = 29}
- D3 = {JOUR: JOUR = 30}
- D4 = {JOUR: JOUR = 31}
- Y1 = {ANNEE: ANNEE = 1900}
- Y2 = {ANNEE: 1812 <= ANNEE <= 2012 et (ANNEE != 1900) et (ANNEE mod 4 = 0)}
- Y3 = {ANNEE: (1812 <= ANNEE <= 2012 et ANNEE mod 4 != 0)}

Cas de test `NextDate` *(1)*

- WECT: 4 cas de tests – partition maximum (D)

```
Cas ID: MOIS,  JOUR,   ANNEE, Sortie
WE1:      6,     14,   1900, 6/15/1900
WE2:      7,     29,   1912, 7/30/1912
WE3:      2,     30,   1913, Donnée invalide
WE4:      6,     31,   1900, Donnée invalide
```

- SECT: 36 cas de tests >> WECT
  - Meilleur mais plus "cher" : 36 cas contre 4 (pour WECT)
  - Potentiel problème: On pourrait ne pas tester le 28 Février.
    - ✓ Solution 5 classes pour JOUR: D1 [1-27], D5 [28] → 45 cas de tests

NextDate SECT: 36 cas de tests

Cas ID	Mois	Jour	An	Sortie anticipée
SE1	6	14	1900	6/15/1900
SE2	6	14	1912	6/15/1912
SE3	6	14	1913	6/15/1913
SE4	6	29	1900	6/30/1900
SE5	6	29	1912	6/30/1912
SE6	6	29	1913	6/30/1913
SE7	6	30	1900	7/1/1900
SE8	6	30	1912	7/1/1912
SE9	6	30	1913	7/1/1913
SE10	6	31	1900	ERREUR
SE11	6	31	1912	ERREUR
SE12	6	31	1913	ERREUR
SE13	7	14	1900	7/15/1900
SE14	7	14	1912	7/15/1912
SE15	7	14	1913	7/15/1913
SE16	7	29	1900	7/30/1900
SE17	7	29	1912	7/30/1912
SE18	7	29	1913	7/30/1913
SE19	7	30	1900	7/31/1900
SE20	7	30	1912	7/31/1912
SE21	7	30	1913	7/31/1913
SE22	7	31	1900	8/1/1900
SE23	7	31	1912	8/1/1912
SE24	7	31	1913	8/1/1913
SE25	2	14	1900	2/15/1900
SE26	2	14	1912	2/15/1912
SE27	2	14	1913	2/15/1913
SE28	2	29	1900	ERREUR
SE29	2	29	1912	3/1/1912
SE30	2	29	1913	ERREUR
SE31	2	30	1900	ERREUR
SE32	2	30	1912	ERREUR
SE33	2	30	1913	ERREUR
SE34	2	31	1900	ERREUR
SE35	2	31	1912	ERREUR
SE36	2	31	1913	ERREUR

Discussion

- Si des conditions d'erreurs sont une haute priorité, nous devons considérer les classes d'équivalence robustes (i.e., en incluant les classes invalides).
- L'ECT est approprié quand les données d'entrée sont définies en termes de domaines et d'ensembles de valeurs discrètes.
- Le SECT fait la supposition que les variables sont indépendantes – les dépendances génèreront des "erreurs" de cas de test.
- Possiblement trop … (Voir les sections suivantes : catégorie-partition et techniques de tables de décision …)

Analyse des valeurs limites

Motivations

- Nous avons partitionné les domaines de données en classes appropriées, avec l'hypothèse que le comportement du programme est "similaire".
- Quelques erreurs typiques de programmation se passent à la limite entre les différentes classes.
- C'est sur quoi le test de valeurs limites se concentre.
- Plus simple mais complémentaire aux techniques précédentes.

Analyses des valeurs limites

- Supposons une fonction F, avec deux variables x1 et x2.
- (Possiblement sans état) Limites :
  a <= x1 <= b, c <= x2 <= d.
- Dans certains langages de programmation, un fort typage permet la spécification de tels intervalles.
- Se concentre sur les bornes de l'espace d'entrée pour identifier les cas de test.
- Le raisonnement est que les erreurs tendent à se produire extrêmement près des valeurs des variables d'entrée – ceci est confirmé par quelques études.

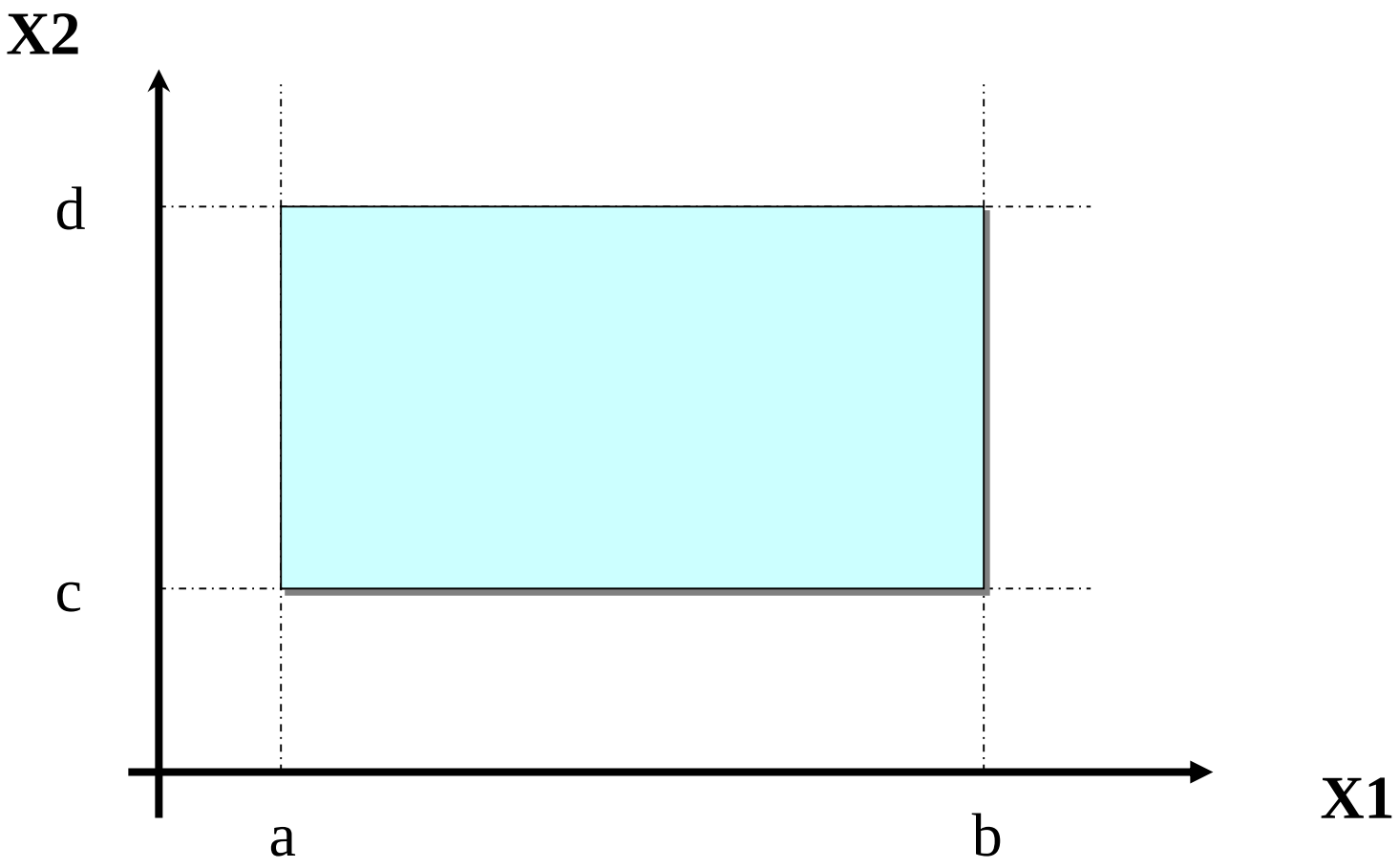
Idées fondamentales

- Les valeurs des variables d'entrée à leur minimum, juste au-dessus du minimum, à une valeur nominale, juste en dessous de leur maximum, et à leur maximum.

- Convention : min, min+, nom, max-, max

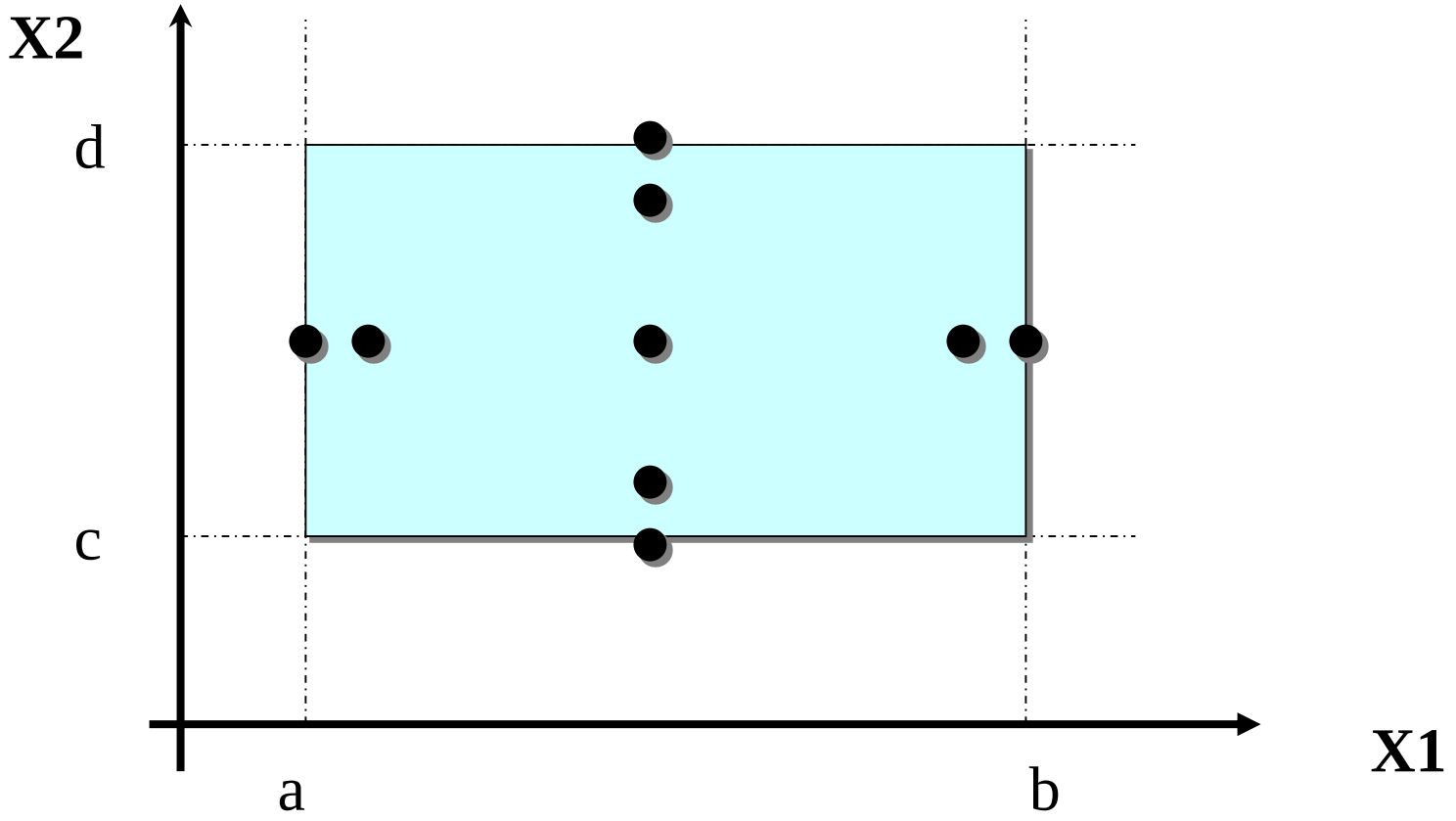
- Garder les valeurs de toutes les variables, sauf une, à leurs valeurs nominales; laissant une seule variable à sa valeur extrême.

Domaine d'entrée de la fonction F



Analyse de la limite des cas de test

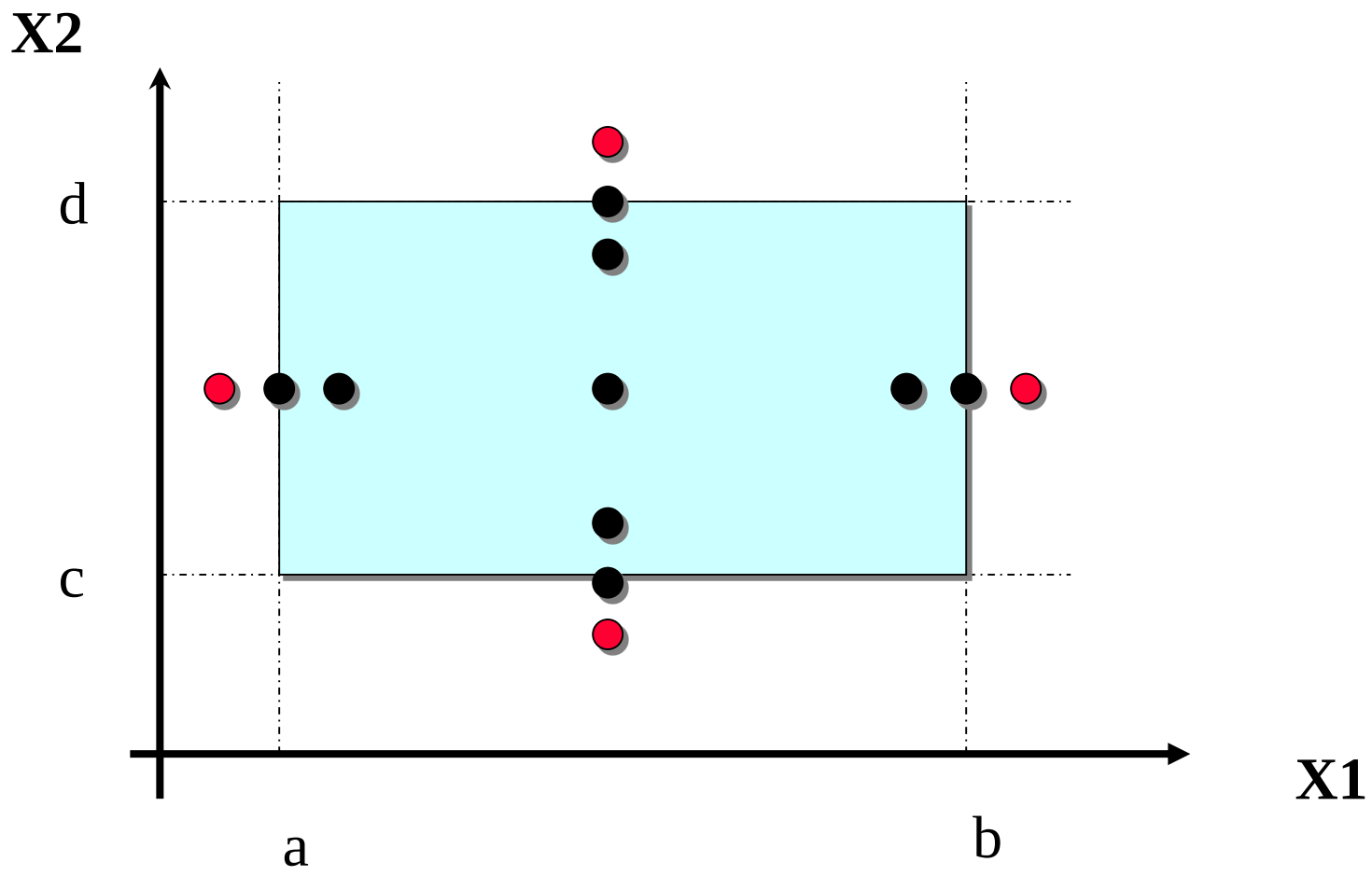
- Ensemble de tests ={<$x1_{nom}$, $x2_{min}$>, <$x1_{nom}$, $x2_{min+}$>, <$x1_{nom}$, $x2_{nom}$>, <$x1_{nom}$, $x2_{max-}$>, <$x1_{nom}$, $x2_{max}$>, <$x1_{min}$, $x2_{nom}$,>, <$x1_{min+}$, $x2_{nom}$,>, <$x1_{max-}$, $x2_{nom}$>, <$x1_{max}$, $x2_{nom}$>}



Cas général et limitations

- Une fonction avec n variables nécessitera **4n + 1** cas de tests.
- Fonctionne bien avec les variables qui représentent des quantités physiques limitées.
- Sans considération de la nature de la fonction ni du sens des variables.
- Une technique rudimentaire qui est adéquate au test de robustesse.

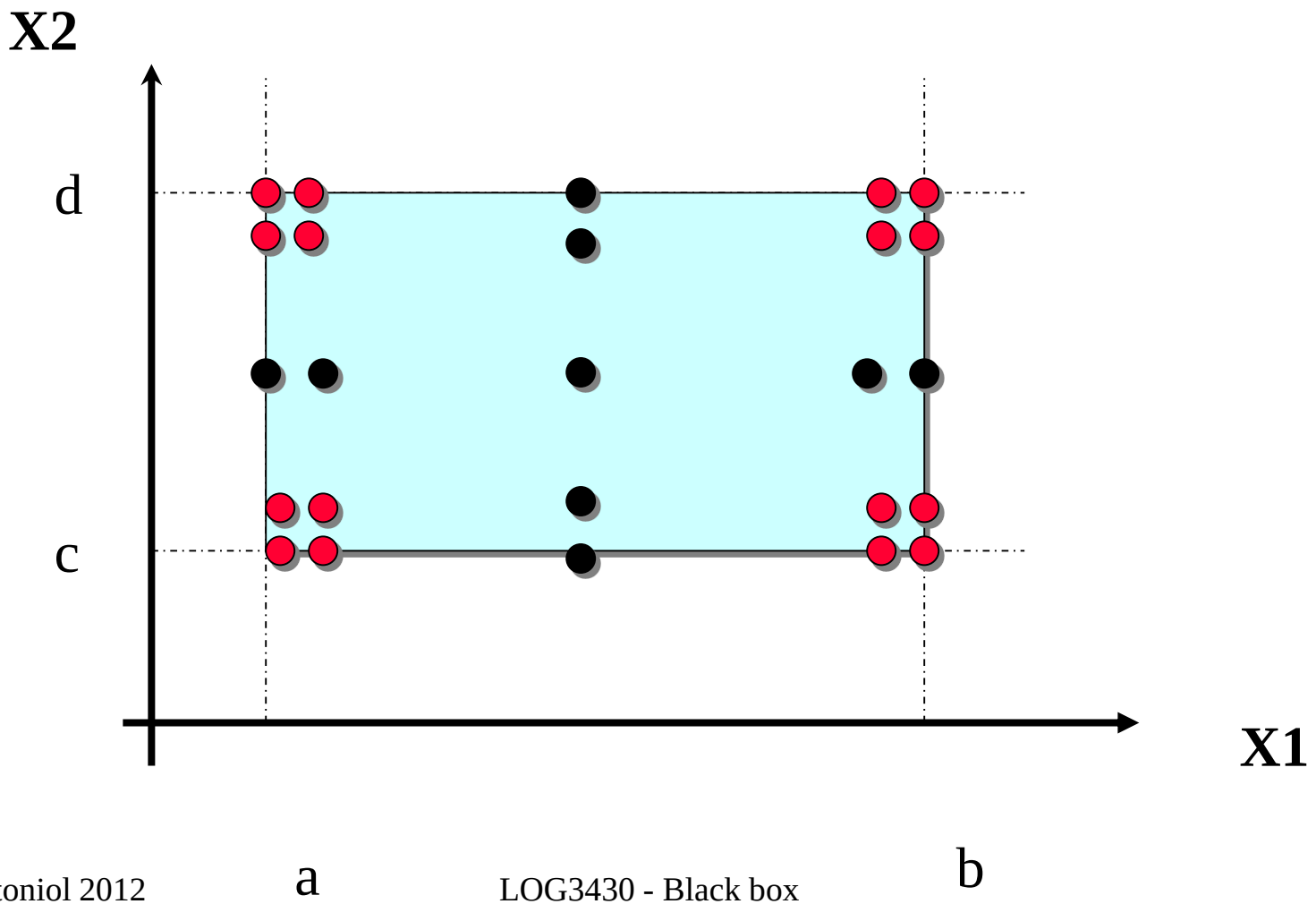
Test de robustesse (max+ and min-)



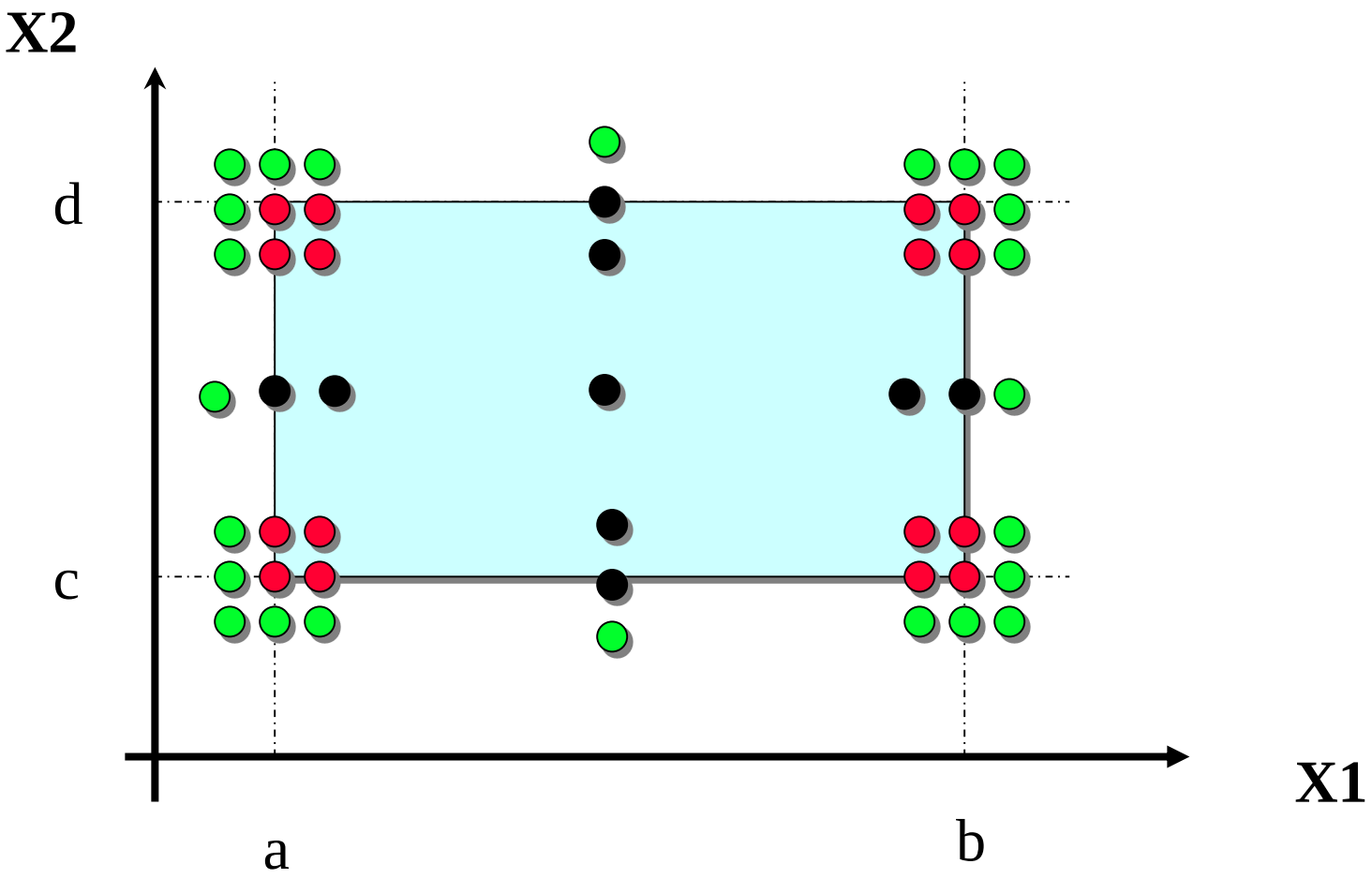
Test des pires cas (WCT)

- La valeur limite fait la supposition commune que les défaillances, la plupart du temps, proviennent d'une faute.
- Qu'arrive-t-il quand plus d'une variable ont une valeur extrême ?
- Une idée provient de l'électronique pour l'analyse de circuits.
- Produit Cartésien de {min, min+, nom, max-, max}.
- Clairement plus minutieuse que l'analyse de la valeur de limite mais beaucoup plus d'effort : **5^n** cas de test
- Une bonne stratégie quand des variables physiques ont de nombreuses interactions et où la défaillance est coûteuse.
- Encore plus loin : tests robustes des pires cas (RWCT).

WCT pour 2 variables



RWCT pour 2 variables



Exercice

Vous développez un logiciel pour gérer un entrepôt de boîtes commerciales. Le poids des boîtes va de 1 à 100 kg. Différentes couleurs doivent être assignées aux boîtes selon leur poids : ROUGE (>50 kg), JAUNE ([30, 50] kg), BLANC (< 30 Kg). L'entreposage se fait sur une étagère ayant trois (3) niveaux d'élévation : Bas, Moyen et Haut. Trois (3) variables booléennes B (pour bas), M (pour moyen) et H (pour haut) permettent de savoir si les niveaux sont disponibles (variable à Vrai) ou non (variable à Faux).

- Les boites Blanches sont de préférence placées au niveau Haut mais s'il est occupé, elles peuvent être posées au niveau Moyen mais, par souci d'efficacité, jamais au niveau Bas.
- Les boites Jaunes sont de préférence placées au niveau Moyen mais s'il est occupé, elles peuvent être entreposées soit au niveau Haut ou au niveau Bas.
- Les boites Rouges sont de préférence placées au niveau Bas mais s'il est occupé, elles peuvent être posées au niveau moyen mais, par souci de sécurité, jamais au niveau Haut.

La fonction *place(couleur, poids, B, M, H)* retourne le niveau où la boîte sera stockée. En outre, elle retourne NOSPACE si la boîte ne peut pas être entreposée. Si les paramètres ne sont pas valides, la fonction retourne ERREUR.

Vous devez couvrir trois critères de test ; classes d'équivalence faible, classe d'équivalence forte et conditions limites. Donnez 1 jeu de test par critère.

Test catégorie-partition

Test catégorie-partition: Étapes

- Le test par catégorie-partition intègre les classes d'équivalence, les valeurs limites et rajoute
  - Des dépendances explicites entre classes d'équivalence (ici des "choix")
  - Des caractéristiques externes (*environnement* sous lequel la fonction opère etc.)
- Le système est divisé en "fonctions" individuelles qui peuvent être testées individuellement.
- La méthode identifie les *paramètres* de chaque "fonction" et, pour chaque paramètre, identifie des *catégories* distinctes.
- Les *catégories* sont des propriétés majeures ou caractéristiques.
- De plus, les *catégories* sont subdivisées en *choix* de la même manière que le partitionnement par classe d'équivalence qui est appliqué ("valeurs" possibles).

Test catégorie-partition: Étapes II

- Les *contraintes* opérant entre les choix sont alors identifiées, i.e., comment l'occurrence d'un choix peut affecter l'existence d'un autre.
  - E.g., dans un exemple de tri de tableau, si Len = 0, alors le reste n'a pas d'importance.
- Les *trames de test* "*test frames"* sont générées, ce qui consiste en combinaisons admissibles de choix dans les catégories (test de spécifications).
- Les trames de test sont alors converties en données de test.

Contraintes

- Les *propriétés, les sélecteurs* associés avec les choix.

Catégorie A :

Choix A1 [propriété X, Y, Z],

Choix A2.

A2 et B2 ne vont pas être combinés

Catégorie B :

Choix B1,

Choix B2 [si X et Z].

Annotation spéciale : [Erreur], [Simple]

Pour des choix qui peuvent être testés avec un seul cas de test, c.-à-d. qui n'ont pas à être combinés avec d'autres choix.

Petit exemple

Un système électronique contrôlant l'accès à un coffre-fort est relié à trois panneaux mécaniques. Le premier panneau (étiqueté S) affiche une des 10 premières lettres de l'alphabet, soit de A à J; les 2ème (étiqueté A) et 3ème (étiqueté N) panneaux affichent chacun un entier entre 0 et 9. Sur la base de ces trois panneaux, un code d'accès numérique est généré. Ce code est le produit des deux entiers et de la position (dans l'ordre alphabétique) de la lettre du panneau S.

Par exemple, pour S = B, A=5, N=4, le code est 2*5*4=40.

Les panneaux sont soumis à un ensemble de contraintes mécaniques : la lettre doit être une *consonne suivie par une consonne (dans l'ordre alphabétique)* et on doit pouvoir retrouver dans l'ensemble des deux chiffres *un nombre premier* et un *chiffre pair*. Toute autre combinaison est mécaniquement impossible.

La fonction *generer_code(S, A, N)* génère un code d'accès si les requis ci-dessus sont respectés, sinon elle génère une ERREUR.

Donnez des jeux de tests pour *generer_code* avec la méthode de catégorie-partition pour le critère **AC (c.-à-d. toutes les combinaisons).**

- Rappel : Un **nombre premier** est un entier naturel qui admet *exactement deux diviseurs distincts entiers et positifs* (qui sont alors 1 et lui-même)

Solution (simplifiée)

- S

  S1 {a,d,e,h,i} : [erreur]

  S2 {b,c,f,g,j} : [si pair et premier]
- A

  A1: {0,4,6,8} [p pair]

  A2: {2} [p pair] [p premier]

  A3: {3,5,7} [p premier]

  A4: {1,9}
- N

  N1: {0,4,6,8} [p pair]

  N2: {2} [p pair] [p premier]

  N3: {3,5,7} [p premier]

  N4: {1,9}

AC:

S2 A1 {N2, N3}

S2 A2 {N1, N2, N3, N4}

S2 A3 {N1, N2}

S2 A4 N2

Exemple complet

- Spécification :

  - Le programme demande à l'usager de taper un nombre entier positif entre 1 et 20 ainsi qu'une chaîne de caractères de la même longueur.

  - Le programme demande ensuite de taper un caractère et retourne la (première) position du caractère dans la chaîne ou un message indiquant l'absence du caractère dans la chaîne.

  - L'usager a l'option de chercher d'autres caractères.

Paramètres et catégories

- Trois paramètres : nombre entier x (longueur), la chaîne a et le caractère c.
- Pour x, les catégories sont « dans l'intervalle » (1-20) ou « hors de l'intervalle »
- Les catégories pour la longueur de a : minimale, maximale, intermédiaire.
- Les catégories pour c : le caractère apparaît au commencement, au milieu, à la fin de la chaîne, ou ne se produit pas dans la chaîne.

Choix

- Nombre entier x, hors intervalle : 0, 21.
- Nombre entier x, dans l'intervalle : 1, 2-19, 20.
- Longueur de la chaîne a : 1, 2-19, 20.
- Caractère c : premier, milieu, dernier, ne se produit pas.
- Remarque : parfois, il n'y a qu'un choix dans la catégorie. La solution ci-dessus combine analyse des limites, robustesse et classes d'équivalence

Spécifications du test formel

x:

x1)	0	[erreur]
x2)	1	[propriété chaineok*, longueur1]
x3)	2-19	[propriété chaineok, midlongueur]
x4)	20	[propriété chaineok, longueur20]
x5)	21	[erreur]

a:

a1)	longueur 1	[si chaineok et longueur1]
a2)	longueur 2-19	[si chaineok et midlongueur]
a3)	longueur 20	[si chaineok et longueur20]

c:

c1)	à la première position dans la	chaîne [si chaineok]
c2)	à la dernière position dans la	chaîne [si chaineok et not longueur1]
c3)	au milieu de la chaîne	[si chaineok et not longueur1]
c4)	pas dans la chaîne	[si chaineok]

Trames de test et de cas de test

x 1	x = 0
x 2a1c1	x = 1, a = ‘A’ , c = ‘A’
x 2a1c4	x = 1, a = ‘A’ , c = ‘B’
x 3a2c1	x = 7, a = ‘ABCDEFG’ , c = ‘A’
x 3a2c2	x = 7, a = ‘ABCDEFG’ , c = ‘G’
x 3a2c3	x = 7, a = ‘ABCDEFG’ , c = ‘D’
x 3a2c4	x = 7, a = ‘ABCDEFG’ , c = ‘X’
x 4a3c1	x = 20, a = ‘ABCDEFGHIJKLMNOPQRST’ , c = ‘A’
x 4a3c2	x = 20, a = ‘ABCDEFGHIJKLMNOPQRST’ , c = ‘T’
x 4a3c3	x = 20, a = ‘ABCDEFGHIJKLMNOPQRST’ , c = ‘J’
x 4a3c4	x = 20, a = ‘ABCDEFGHIJKLMNOPQRST’ , c = ‘X’
x 5	x = 21

12 cas de tests contre 60 (5*3*4) pour SECT

Critères utilisant les choix

- Toutes les combinaisons (**AC**) :Une valeur pour chaque choix de chaque paramètre doit être utilisée avec une valeur de chaque choix (possible) de chaque autre catégorie. **[Critère par défaut]**

- Chaque choix (**EC**) : Une valeur de chaque choix pour chaque catégorie doit être utilisée dans un cas de test.

- Choix de base (**BC**) : Un choix de base est sélectionné pour chaque catégorie. Un premier test de base est formé en utilisant le choix de base pour chaque catégorie. Les autres tests sont ensuite construits en remplaçant un (et un seul à la fois) choix de base par un autre choix de sa catégorie. **[Critère de Compromis]**

Conclusions

- L'identification des paramètres et des conditions d'environnement, et des catégories dépend fortement de l'expérience du testeur.
- Rend le test des décisions explicite (e.g., contraintes), prêt pour l'évaluation.
- Combine l'analyse des valeurs limites, le test de robustesse et le partitionnement par classes d'équivalences.
- Une fois que la première étape est complétée, la technique est simple et peut être automatisée.
- La technique pour la réduction des cas de test le rend utile pour le test pratique.