Fiabilité Logicielle
Introduction

1. Introduction
   - Etat des Lieux
   - Fiabilité
   - Sécurité
   - Validation et Vérification (VV)
2. Un Exemple Introductif
3. Principes et Terminologie
   - Limites du test
   - Base du test
4. Spécifications
   - Correction de Programme
   - Expression Formelles de Spécification
5. Ce qui n'est pas couvert
6. Bibilographie

Introduction

Ce qu'on voit...

- An unexpected exception has been detected in native code outside the VM. Unexpected Signal : EXCEPTION_ACCESS_VIOLATION (0xc0000005) occurred at PC=0x10
- Ariane Flight 501 was the first, and unsuccessful, test flight of the European Ariane 5 rocket. Due to an error in the software design, the rocket veered off its flight path 37 seconds after launch and was destroyed by its automated self-destruct As it was an unmanned flight, there were no victims, but the breakup caused the loss of four Cluster mission spacecraft, resulting in a loss of more than US $ 370 million.
- A critical vulnerability has been identified in Adobe Reader 9. This vulnerability would cause the application to crash and could potentially allow an attacker to take control of the affected system. There are reports that this issue is being exploited.

Deux problèmatiques distinctes

Fiabilité: Le système se comporte toujours comme ce qui est attendu

- Qualité logicielle
- Validation, Vérification
- Test, Simulation, Déboggage, Certification

Sécurité: le système résiste à des attaques malveillantes.

- Virus, vers, malware, phishing, attaque...
- Détection d'intrusion, politique de sécurité, pot de miel...
- Protection de l'information, confidentialité, secret, anonymat

Quel Coût?

- la fiabilité coute cher: 30% à 80% (logiciels critiques) du développement sur vérification et validation.
  - Nécessaire: critères de qualité logiciel demandés par le client
  - Incontournable: logiciels critiques (transport, nucléaire,...)
- mais peut rapporter gros (cout des bugs: 64 Milliards $ en 2002 aux USA, Bug du pentium (94): 500 Millions de $)
  - Difficile à mettre en oeuvre: équipes de tests, méthodologies,...
  - Toujours sacrifié en période de restriction...

Problème: les systèmes sont extrèmement complexes.

- Taille: compilateur GCC dizaine de milliers de loc, milliers de variables.
- Complexité des interactions: systèmes concurrents, systèmes distribués et interactifs, système de très grande taille.
- Diversité des cas d'utilisation: reproduire une erreur survenue en surfant sur le Web.

Maintenance et Réutilisation

Maintenance des logiciels

- Correction d'erreurs
- Ajout de fonctionalités

Logiciel bati sur d'autres versions ou produits.

- mettre à jour les spécifications
- mettre à jour les tests

Qualité logicielle

Exigée par le client! Comment l'assurer:

1. norme de codage et processus de développement
2. test
3. revue de code
4. certification

Logiciel critique (avionique, nucléaire, transport,..): normes, outils de validation.

Sécurité

Pas l'objet de ce cours, voir *Sécurité Internet/Réseau*
Enjeu critiques:

- Attaque pour prendre le controle d'un système: virus, vers,... en utilisant des faiblesses des systèmes d'exploitation et langage de programmation buffer overflow, SQl injection
- Attaque sur le hardware (carte à puce, téléphone,...)
- Attaque sur les protocoles utilisés (man in the middle, attaque de rejeu, ...)

Propriétés spécifiques: secret, confidentialité, anonymat

Un exemple: Protocole d'authentication de Needham-Schroeder

$$
\begin{array}{lcl}
A & \rightarrow & B :< A, \{N_A\}_{K_B} > \\
B & \rightarrow & A :< B, \{< N_A, N_B >\}_{K_A} > \\
A & \rightarrow & B :< A, \{N_B\}_{K_B} >
\end{array}
$$

Deux principaux (agents), role A(lice), et B(ob).

Keys: clefs publiques K_A, K_B or symmetriques utilisées par les fonctions cryptographiques (RSA).

Nonce: nombres aléatoires générés à chaque session N_A, N_B.

Messages: construits avec constantes, paire $< _, _ >$, encryption $\{_\}__$, hashing, etc.

Sessions: un principal peut jouer le role A dans une session, role B dans une autre, sessions en parallèle,...

Hypothèses

Encryption, decryption sont parfaites: RSA, AES,. . . boites noires: Obtenir m depuis $\{m\}_K$ exige de connaitre K (clé symmetrique) ou la clé privée associée K^{-1}.

Le protocole doit établir la propriété dans un environnement hostile.

Needham-Schroeder établit l'authentication: à la fin d'une exécution, Alice s'est authentifiée auprès de Bob et réciproquement

Hypothèses

Encryption, decryption sont parfaites: RSA, AES,. . . boites noires: Obtenir m depuis $\{m\}_K$ exige de connaitre K (clé symmetrique) ou la clé privée associée K^{-1}.

Le protocole doit établir la propriété dans un environnement hostile.

Needham-Schroeder établit l'authentication: à la fin d'une exécution, Alice s'est authentifiée auprès de Bob et réciproquement

$$\begin{aligned} A &\rightarrow B :< A, \{N_A\}_{K_B} > \\ B &\rightarrow A :< B, \{< N_A, N_B >\}_{K_A} > \\ A &\rightarrow B :< A, \{N_B\}_{K_B} > \end{aligned}$$

Attaque

Attaque: une éxecution du protocole qui échoue à établir la propriété voulue.
Modeliser l' environnement: intrus I qui peut

- intercepter and analyzer les messages (cas passif)
- créer et envoyer des messages (cas actif).

Un principal peut être malhonnête!

Les Propriétés

- Secrecy: l'intrus arrive à connaitre certaines données.

Les Propriétés

- Secrecy: l'intrus arrive à connaitre certaines données.
  E-voting:
  vote Obama ou McCain
  encrypté par K_{pub}
  (K_{pub}^{-1} connu par l'autorité)
  On peut intercepter mon vote et trouver mon choix sans connaitre K_{pub}^{-1}!

Les Propriétés

- Secrecy: l'intrus arrive à connaitre certaines données.
- Authentification: prouver son identité.
  nécéssaire pour paiement electronique!
  Réduit à du secret.

Les Propriétés

- Secrecy: l'intrus arrive à connaitre certaines données.
- Authentification: prouver son identité.
  nécéssaire pour paiement electronique!
  Réduit à du secret.
- Anonymity:
  ne pas retrouver l'identité de l'envoyeur.

Les Propriétés

- Secrecy: l'intrus arrive à connaitre certaines données.
- Authentification: prouver son identité.
  nécéssaire pour paiement electronique!
  Réduit à du secret.
- Anonymity:
  ne pas retrouver l'identité de l'envoyeur.
- Non-repudiation
  Signature de contrat.

Vérification de Protocole

Problème difficile: vérifier la correction d'un programme distribué!

A la main: peu sûr $\Rightarrow$ méthodes formelles.

(Fait) Une erreur a été trouvé dans NS 17 ans après sa publication.

L'attaque sur NS.

- 3 principaux a, b honnêtes, c malhonnêtes,
- 2 sessions en parallèle

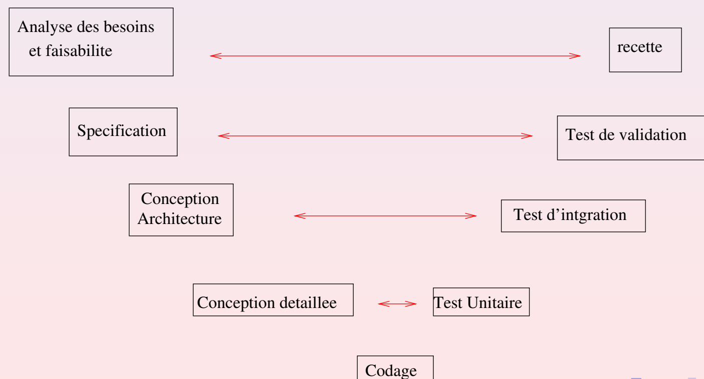
a	$\rightarrow$	$c :< a, \{N_a\}_{K_c} >$	1ere règle, session 1, a joue A, c joue B	*b* pense parler à *a*.
$c(a)$	$\rightarrow$	$b :< a, \{N_a\}_{K_b} >$	1ere règle, session 2, c joue as a pour b	
b	$\rightarrow$	$c :< b, \{< N_a, N_b >\}_{K_a} >$	2eme règle, session 2, b reponds c	
c	$\rightarrow$	$a :< c, \{< N_a, N_b >\}_{K_a} >$	2eme règle, session 2, c joue B, a joue A	
a	$\rightarrow$	$c :< a, \{N_b\}_{K_c} >$	3eme règle, session 1, a joue A, c joue B	
$c(a)$	$\rightarrow$	$b :< a, \{N_b\}_{K_b} >$	3eme règle, session 2, c joue a pour b	

Exercice

Donner une version corrigée de NS.

Developpement logiciel

Le cycle en V (le plus standard):



Vérification et Validation

Vérification: le programme contient-il des défauts (bugs)?

```
if (x==0) {return 1/x;}
```

Validation: le programme est-il conforme aux attentes du client?

Q:Avez-vous l'heure?

R:oui

Méthodes de VV

1. Simulation
2. Revue de code
3. Test
4. Méthodes formelles
   1. Génération de code à partir de spécifications formelles (B, Coq,...) ex: ligne 14 du métro
   2. Modélisation et techniques de model-checking (cours MF)
   3. Analyse statique de programme

Noter les différents types de tests:

- Unitaire: fragment le plus simple (procedure, classe ou module entier)
- Intégration: tester l'interaction des composants entre eux
- Validation (ou fonctionnels): conformité avec les attentes du client

Principe: la VV n'est pas à la charge du codeur (équipe de tests)

Une autre approche: Xtreme Programming

Inventée après l'échec d'un système de paie de GM.

- Adaptée aux projets de taille moyenne
- Test et code simultanée
- Cycle d'itération validation de code
- Interaction avec le client pour modifier étendre le produit
- Principe de remaniement constant du code, mobilité des programmeurs
- Nécessité d'outils (CVS, systèmes de reporting,...) et méthodes (test de régression)

Exercice de programmation (Myers)

1. Ecrire une fonction *TypeTriangle* dans le langage de votre choix qui demande trois arguments réels *a,b,c* correspondant aux trois cotés d'un triangle et renvoie le type du triangle:
   - 2 si le triangle est isocèle (2 cotés sont égaux),
   - 3 si le triangle est équilatéral (3 cotés sont égaux),
   - 1 si le triangle est scalène (trois cotés sont différents),
   - 0 si les 3 valeurs ne définissent pas un triangle.
2. Valider le programme à l'aide d'un jeu de test.

Trouver un algorithme

1. $a == b == c$ équilatéral
2. $a == b, b \neq c, a \neq c$ isocèle et cas symétriques
3. $a \neq b, b \neq c, c \neq a$ scalène

Mais seulement si a, b, c définissent un triangle!

Ecrire le programme (C, Java,..)

```
int main(){
int a,b,c
printf("entrer a,b,c\n");
scanf("%d%d%d",a,b,c);
...

return 0;
}
```

Valider le programme

Question le jeu de test peut-il etre défini avant l'écriture du programme?

Limites théoriques

Theorem

Toutes les propriétés intéressantes des programmes sont indécidables.

- Est-ce le programme termine sur cette donnée (resp. toujours)?
- Est-ce qu'une variable prend une valeur donnée?
- Est-ce que cette instruction est exécutée?
- ...

Tout langage de programmation raisonnable est Turing complet.

Indécidabilité de l'arret

Par l'absurde: on suppose l'existence d'un programme *halt(P,D)* qui renvoie True si P(D) termine et False sinon.

1. Hypothèse: programme et données sont de même nature (donc un programme peut être une donnée)
2. Toto(X):

   si halt(X,X) alors boucler
   sinon stop fsi
3. Et en déduire une contradiction.

Que fait Toto(Toto)?

Indécidabilité de l'arret

Par l'absurde: on suppose l'existence d'un programme *halt(P,D)* qui renvoie True si P(D) termine et False sinon.

1. Hypothèse: programme et données sont de même nature (donc un programme peut être une donnée)
2. Toto(X):

   si halt(X,X) alors boucler
   sinon stop fsi
3. Et en déduire une contradiction.

Que fait Toto(Toto)?

Exercice pratique: déterminer si le programme

```
f(n)=si n<=1 alors retourner 1
     sinon si pair(n) alors f(n/2)
           sinon f(3n+1) fsi
     fsi
```

s'arrête pour toute valeur de n.
(Problème de Syracuse)
Résolu depuis cette année! (après 40 ans de recherche).

Limites pratiques

```
long add (int x, int y){
    return x+y;
}
```

x, y sur 32 bits: $2^{31} - 1$ valeurs possibles pour chacun
$\approx 2^{32}$ valeurs à vérifier: $4 * 10^9$
Estimer le temps si un test se fait en $10^{-6}s$
idem si on passe à des entiers longs sur 64 bits.

Paradigmes

Myers: Tester c'est executer le programme dans l'intention d'en trouver des anomalies ou des défauts. The Art of Software Testing (1979)

Dijkstra: Testing can only reveal the absence of error never their absence. Notes on Structured Programming (1972)

```
if (a>= 0) {delta=b*b-a*c;}          Défaut
.
.                                    Infection
.
if (delta==0) {printf("la solution est %d\n",-b/2*a);}    Erreur
```

- Défaut: instruction incorrecte
- Infection : propagation des conséquences du défaut dans la suite du programme
- Erreur: constatation d'une anomalie dans le programme.

Test: faire apparaitre les erreurs et documenter.
Deboggage: remonter au défaut
Correction: remédier au défaut (sans en créer de nouveau!)

Un Test:

1. Déterminer un Cas de Test (CT)
2. Déterminer les Données de Test (DT) correspondantes
3. Utiliser un Oracle pour définir le résultat attendu
4. Ecrire un script de test pour exécuter le programme sur les DT
5. Editer un rapport (Pass ou Fail)

Un Test:

1. CT: triangle isocèle.
2. DT: a=1,b=1,c=1
3. Oracle: répondre *isocèle*
4. Script de test
5. Editer un rapport (Pass ou Fail) Test CasIsocèle: statut Pass

Fastidieux: nécessité d'automatiser.

Environnements de test pour les langages usuels.
Java: Junit (incorporé dans Eclipse)

Nécessité de gérer les tests (suites de test), leur résultats et la correction des erreurs (outils de suivi Bugzilla, Mantis,...)

Catégorisation des tests

Test unitaire: unité de programmation

- Définition de cas de test jusqu'à écriture de suite de test.
- Test boite noire: sans connaitre le programme
- Test boite blanche: à partir du programme
- Test probabiliste génération aléatoire de données de tests.

- Test d'Intégration: tester l'interaction des modules entre eux.
  Descendant ou ascendant
  Simulation de modules via leur description formelle ou interface.
- Test Fonctionnel: tester l'application vs les fonctionalités spécifiées.
- Test de non-régression: vérifier que les modifications ne modifient pas le comportement du programme.
  Nécessité 'une automatisation pour réduire le coût du rejeu.

Gestion de test

- Construction et automatisation de tests: suite de tests.
- Documentation des erreurs et du suivi: outil de reporting et gestion (Bugzilla, Mantis,..).
- Debogage: détection des défauts du programme conduisant à une erreur.

Spécifications

```
public static void sort(int[] arr){
    int a=0, b=0, c=arr.length;
    while (b<c){
        if (arr[b]==1){
            arr[a]=arr[a];
            arr[a]=1;
            a=a+1;
            b=b+1;
        } else { if (arr[b]==2){
                b=b+1; } else {
                         c=c-1;
                         arr[b]=arr[c];
                         arr[c]=3;
            }   }  }  }
```

Ce programme est-il correct?

Impossible de le savoir si on ne *spécifie* pas ce qu'il est sensé faire!

Spécifions un tri *int [ ] sort (int [ ] a)*

Spécification: *Cette fonction retourne une version triée de son entrée.*

Plus formel:

- Précondition: établit ce qui est demandé avant l'exécution de la fonction *a est un tableau d'entier*
- Postcondition: établit ce qui est assuré après l'exécution de la fonction. *renvoie un tableau trié*

Tests: sort({})={}

sort({3,2,1})={1,2,3}

sort({2})={2}

Tests:
sort({2,1,1})={-32453,1,2}

Postcondition *le tableau résultat ne contient que des valeurs du tableau a*

Tests:
sort({1,2})={1,2,2}

Postcondition *le tableau résultat contient toutes les valeurs du tableau a avec le même nombre d'occurrence.*

Tests:
sort(null)
nullPointerException

Précondition: le tableau a est non null

Précondition: le tableau a est un tableau non null d'entier

Postcondition: renvoie un tableau d'entier trié par ordre croissant et qui est une permutation du tableau a.

Spécification: description des besoins et attentes.

1. Langue naturelle: flou et ambigu (ou trop verbeux!)
2. Description formelle: logique pour décrire les programmes et leurs propriétés.
3. Logique du premier ordre -ou d'ordre supérieur pour parler de fonction et d'ensemble)- avec des domaines prédéfinis (entiers, booléens,...)
4. Logique modales ou temporelles.

Une partie du cours sera consacré à cette problématique.

- Test d'applications "compliquées" (service Web, Base de données (confidentialité, intégrité,..) et outil de test par jeu de scénario.
- Test Systemes distribués: outil de simulation, méthode de model-checking
- Test systèmes critiques: idem + dérivation de programme à partir de spécifications formelles.
- Principes de déboggage

Bibliographie:

- Cours disponibles sur le net
  - Cours Testing, Debugging and Verification. W. Ahrendt, R. Hahnle (Chalmers University)
  - Cours sur le Test. S. Bardin
- Junit: la page officielle (sourceforge)
- JML: la page officielle (sourceforge)
- Livres
  - The Art of Software Testing. Myers
  - Object Oriented Programming. B. Meyer
  - Unit Testing in Java. J. Links. Morgan Kaufmann
  - Why Programs Fail. A. Zeller.