Fiabilité Logicielle

Généralités

Arnaud Labourel

2025-26

amU Faculté des sciences
Aix Marseille Université

Section 1

Informations générales sur l'enseignement

Fiabilité Logicielle – Agenda

Les lundis sur 6 semaines :

- 6 cours de 1h30 : 9 heures (A. Labourel)
- 4 TD de 2h heures : 8 heures (A. Labourel et F.-X. Dupé)
- 5 TP de 2h heures : 10 heures (A. Labourel et F.-X. Dupé)

MCC : $(0,5 * CC) + (0,5 * ET)$ avec CC note de suivi de TP (rendu chaque semaine)

Utilisation de la plateforme AMeTICE

- mise en ligne : supports de cours - sujets de TD et de TP
- rendu des travaux pratiques via git (etulab)

Adresse mail :

- arnaud.labourel@univ-amu.fr
- francois-xavier.dupe@univ-amu.fr

Fiabilité Logicielle – Contenu

1. Spécifications (aperçu)
2. Tests (boite blanche et boite noire)
3. Méthodes formelles (évoquées rapidement)

Section 2

Introduction à la fiabilité logicielle

Fiabilité logicielle : une définition

- **Quoi** : assurer un certain nombre de propriétés sur le logiciel en production (dans un environnement donné) qui garantissent son aptitude à remplir sa mission.
- **Comment** : en appliquant des méthodes ou en suivant des méthodologies lors de son développement et de sa maintenance

Fiabilité logicielle : pourquoi ?

- **enjeux économique ou de sureté/sécurité** des personnes, des infrastructures et des organisations du fait de l'omniprésence du logiciel

  exemple : transports, production énergétique, secteur bancaire, ...

- **complexité des logiciels** : taille des développements - inter-dépendance logicielles (API, Applications orientée service, cloud, ...)

  exemple : noyau Linux 17 millions de lignes de code pour 19000 contributeurs, ...

Fiabilité logicielle : concepts en jeu

- *Run Time Error* : erreur à l'exécution
- Adéquation à une spécification
- Maintenance logicielle
- Sécurité
- Performance

Ceci induit des coûts nécessitant du **professionnalisme**.

Fiabilité vs Qualité logicielle

La **qualité logicielle** comprend :

- le suivi d'une méthodologie visant à garantir une certaine "qualité" (norme ISO/CEI 9126)
- la mise en place d'indicateurs permettant une mesure de qualité

La **fiabilité logicielle** est une des composantes de la qualité logicielle qui comprend également :

- la facilité d'utilisation
- la maintenabilité (mesure l'effort pour modifier le logiciel)
- la portabilité (facilité d'utilisation/installation sur différents environnements)

Fiabilité vs Qualité du code

La **qualité de code** comprend :

- lisibilité
- efficacité
- durabilité (maintenabilité - réutilisabilité - portabilité)

Un **code de qualité** tend à augmenter la **fiabilité du logiciel** (relecture plus facile, bonnes pratiques éliminant des constructions exotiques, ...).

Deux problématiques distinctes : Fiabilité vs Sécurité

- **Fiabilité** : le système se comporte toujours comme attendu (spécification) dans le cadre/environnement prévu
  - qualité logicielle
  - test, simulation, débogage
  - validation, certification
- **Sécurité** : le système résiste à des attaques malveillantes (contexte non prévu/prévisible)
  - virus, *malware*, *phishing*
  - détection d'intrusion, pot de miel
  - authentification, confidentialité, secret, anonymat, politique de sécurité

Propriétés fonctionnelles (lien entre entrée et sorties) vs propriétés non-fonctionnelles (tous les autres aspects : performance, sécurité, ...).

Lien entre fiabilité et sécurité

Absence de fiabilité d'un programme peut entrainer des failles de sécurité.

- *Heartbleed* (faille OpenSSL) en 2014: utilisation de `memcpy` sans vérification de taille $\Rightarrow$ vol d'information et compromission d'informations sensibles
- Mauvaise implémentation de l'algorithme de signature ECDSA en Java en 2022 : oubli de vérification que les deux parties de la signature sont non nulles (auquel cas la signature est valide quel que soit le message signé)

Causes fréquentes de non-fiabilité :

- manque de robustesse de l'application ou des applications/API tiers utilisées
- mauvaise utilisation des APIs
- mauvaise gestion de la mémoire (fuites, dépassements de tampon, ...)
- oubli de gestions de cas particuliers

Fiabilité logicielle : quelques échecs (1/2)

- Le 4 juin 1996, le premier vol de la fusée Ariane 5 se termina après 37 secondes par la perte de contrôle de l'engin, suivie de son explosion.
  - Réutilisation de code d'Ariane 4 dans Ariane 5 sans prendre en compte que les données d'entrée ne seraient
  - Absence de récupération de l'erreur - redondance matériel inutile
  - Coût : 370 millions $
- En 2015, Toyota rappelle 650000 véhicules hybrides suite à un bug logiciel
  - Les paramètres du logiciel de contrôle des unités de commande des moteurs/générateurs et du système hybride peuvent générer une dégradation de certains composants électroniques". Dans certains cas, le véhicule pourrait passer en mode « sécurité » en limitant la puissance disponible voire même, plus rarement, devenir totalement inopérant.

Fiabilité logicielle : quelques échecs (2/2)

- En novembre 2021, les possesseurs de Tesla n'ont pu ouvrir ou démarrer leur voiture à cause d'un bug informatique.
  - Un problème de serveur internet, rend inopérante l'application smartphone permettant habituellement de commander sa Tesla.
- Entre 1985 et 1987, l'outil de radiographie Therac-25 fut impliqué dans au moins six accidents, des patients recevant des doses massives de radiation.
  - la défaillance est due au logiciel pilotant l'outil.
  - au moins cinq patients décédèrent des suites de l'irradiation.
- projet du logiciel Louvois 1996 - 2013 (gestion des soldes des armées) - développement initial en interne, puis appel à Stéria
  - logiciel complexe qui n'a jamais fonctionné : soldes non versées - trop perçu.
  - abandon définitif du projet en 2013 ; coût : 175 millions €

Fiabilité logicielle et développement logiciel

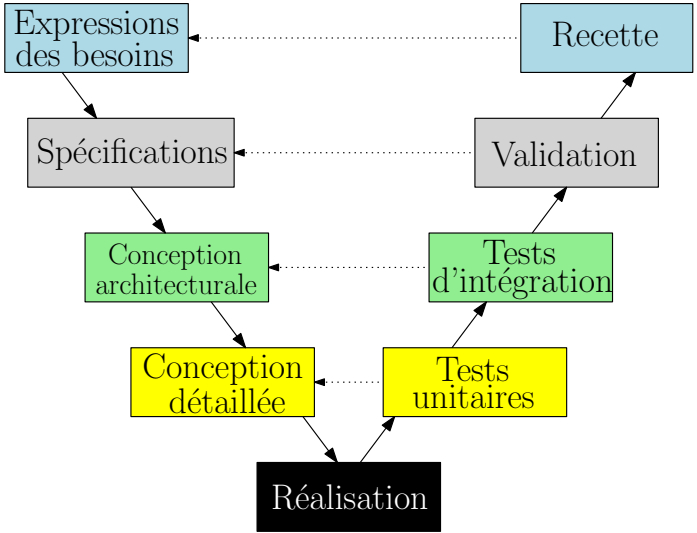
Crise du logiciel : années 1960

explosion de la taille et de la complexité des logiciels $\Rightarrow$ difficultés de développement, de maintenance et de fiabilité

$\Rightarrow$ naissance du terme "génie logiciel" (*software engineering*) avec les méthodologies de gestion de projet en cascade et en cycle en V qui intègrent des phases de validation et de vérification

La fiabilité logicielle est donc un enjeu majeur du développement logiciel depuis plus de 50 ans qui n'est toujours pas résolu de manière satisfaisante.

Fiabilité logicielle dans le cycle en V



Le cycle en V

Les tests forment la remontée après la cascade des analyses

Fiabilité logicielle avec méthodes agiles et TDD

Méthode Agile :

- tests au niveau de chacun des sprints, qui en font pleinement partie
- tests fonctionnels et d'acceptation basés sur les "user stories" du sprint puis correction des bugs avant le sprint suivant

TDD : Test Driven Development - Développement dirigé par les tests

Développement incrémental selon le cycle

- ajout d'une fonctionnalité élémentaire
- création d'une suite de tests pour cette fonctionnalité
- développement de code "minimal" (passant les tests) pour cette fonctionnalité
- tests de non-régression

Fiabilité logicielle avec BDD

BDD : Behavior Driven Development - Développement dirigé par le comportement

- extension du TDD
- utilisation d'un langage commun entre développeurs, testeurs et clients pour décrire le comportement attendu du logiciel
- description des comportements attendus sous forme de scénarios (donnant lieu à des tests automatisés)
- permet une meilleure compréhension des exigences et une collaboration plus étroite entre les différentes parties prenantes du projet

Utilité et coût de la fiabilité logicielle

Utilité de la fiabilité du logiciel :

- Nécessaire : critères de qualité logicielle exigée par le client
- Indispensable : logiciels critiques (transport, nucléaire, ...)
- Coût : de 30 à 80% du coût du développement (vérification et validation)

Mais reste souvent vu comme une activité non productive :

- Difficile à mettre en œuvre : équipe dédiée, méthodologies, ...
- Sacrifié lors des retards de livraison "Ce n'est pas un bug, c'est une feature !"
- Souvent négligé par les étudiants : "ça compile, ça marche"

Cependant, le coût des erreurs logicielles est important :

- bug du pentium (erreur dans l'algorithme de division de l'ALU) : 500 Millions de $
- logiciel du 737 Max : plusieurs milliards de $

Difficultés de la fiabilité lors du développement logiciel

- Taille des logiciels développés : plusieurs millions de lignes de code
- spécification incomplète, ambiguë ou erronée (oublis de cas particuliers)
- Complexité des logiciels développés : systèmes distribués avec de nombreux composants
- Utilisation de code et d'applications tiers (API, bibliothèques, ...)
- Multitude des scénarios d'utilisation et d'interaction (difficulté de reproduction des bugs)

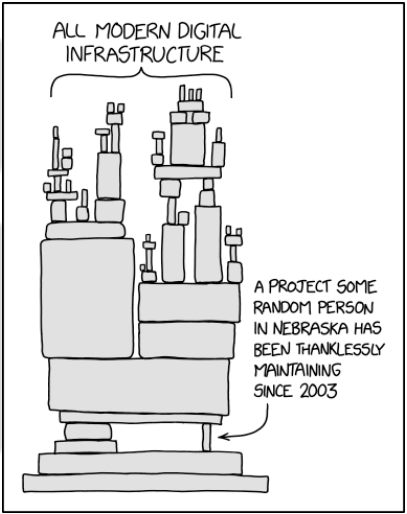
Difficultés de la fiabilité lors du cycle de vie

Maintenance

- Correction d'erreurs
- ajout de fonctionnalités

Réutilisation : logiciel bâti sur d'autres versions ou produits (API)

- mettre à jour les spécifications
- reprendre le processus de validation/vérification (tests, analyses, ...)



Fiabilité logicielle du point de vue du client

Comment assurer la qualité logicielle exigée par le client ?

- processus et méthodologie de développement (cycle en V, méthode agile, normes de codage, ...)
- tests (recette utilisateur/tests d'acceptation)
- revue de code et audit
- certification (normes)
- analyse statique de programmes
- méthodes formelles

En cas de logiciels critiques, les normes et les méthodes de validation sont renforcées.

Fiabilité Logicielle – Contenu

1. Spécifications (aperçu) :
   1. langue naturelle
   2. langages standardisés/normalisés (UML, SDL, SADT, ...)
   3. langages formels (logique, automates, ...)
2. Tests :
   1. tests boite blanche : couverture de code
   2. *mocking* : simulation de composants externes pour tester des parties
   3. tests boite noire : tests sans code accessible
3. Méthodes formelles :
   1. Model-checking : modéliser le système par un automate et vérifier des propriétés exprimées en logique temporelle
   2. Analyse statique
   3. Preuve de programmes - vérification déductive

La spécification en quelques mots (1/2)

Différents formalismes de spécification :

- langue naturelle
- langage standardisé et/ou normalisé
  - UML : diagrammes de cas d'utilisation, de classes, d'activités, de séquences, ...
  - SADT : analyse fonctionnelle descendante représentant les activités et les flux entre elles
  - SDL : spécification de systèmes temps-réel
- langages formels : logique, automates, ...

La spécification en quelques mots (2/2)

- langage naturel :
  - ⊕ compréhensible par tous (y compris le client)
  - ⊖ sémantique floue, ambigüe (trop verbeux)
- langage standardisé et/ou normalisé
  - ⊕ langage permettant la compréhension entre MOA, MOE et les développeurs
  - ⊖ sémantique pas forcément suffisamment détaillée (seule)
- langages formels : logique, automates, ...
  - ⊕ langage avec une sémantique non ambigüe
  - ⊖ difficile à appréhender par le client (expertise minimale requise pour la compréhension)

Les tests en quelques mots

- Tester mon programme

  pour découvrir des défauts (bugs)

- Automatiser les tests

  pour systématiser et faciliter la répétition des tests (lors de la maintenance par exemple)

- Choisir des tests

  pour maximiser la découverte de défauts avec un minimum de tests

Différents types de tests

Niveaux de tests

1. Tests unitaires (ou test de composants)
2. Tests d'intégration
3. Tests système (ou tests fonctionnels)
4. Tests d'acceptation (ou tests de recette)

Types de tests

- Tests avec données persistantes
- Test de non-régression
- Tests aléatoires (Fuzzing)

Test : erreurs, infections et défauts

```
static int fibonacci(int n) {
  if (n == 0) return 0;
  // error: should be fibonacci = 1
  int previous = 0, fibonacci = -1;
  for (int i = 2; i <= n; i++) {
    int temp = previous + fibonacci;
    previous = fibonacci;
    fibonacci = temp;
  }
  return fibonacci;
```

erreur :
instruction erronée (l'erreur est humaine)
infection :
propagation des conséquences de l'erreur dans la suite du programme
défaut :
constatation d'une faute/ d'un défaut dans le programme

- Test : faire apparaitre les défauts et documenter
- Déboggage : remonter à l'erreur
- Correction : corriger l'erreur

Les méthodes formelles en quelques mots

E. W. Dijkstra (prix Turing 1972)

"Program testing can be used to show the presence of bugs, but never to show their absence!"

Méthodes formelles : prouver que le programme vérifie certaines propriétés.

Quelques exemples :

- analyse statique
- model-checking
- preuve de programmes
- génération automatique du programme

Limites théoriques

Théorème de Rice : toute propriété "intéressante" (non-triviale) d'un programme écrit dans un langage/formalisme Turing-complet est indécidable

- Est-ce que le programme termine ?
- Cette variable prend-elle la valeur 0 durant l'exécution ?
- Cette instruction est-elle exécutée ?
- ...

Preuve : indécidabilité du problème de l'arrêt.

Automatisation de la vérification de propriétés

Objectif : définir des algorithmes de vérification de propriétés

ALogrithme qui, pour un programme P et une propriété ϕ, décide si P vérifie ϕ

- correction : si la propriété est vérifiée, l'algorithme répond vrai

  Les méthodes incorrectes renvoie des faux-négatifs (l'algorithme valide la propriété alors qu'elle est fausse)

- complétude : si la propriété n'est pas vérifiée, l'algorithme répond faux

  Les méthodes incomplètes renvoie des faux-positifs (l'algorithme ne peut décider que la propriété est bien vérifiée)

Les méthodes formelles : Analyse statique

- vérification d'une propriété sans exécution du code
- approximation du comportement du programme
- propriétés spécifiques :
  - non-déréférencement de pointeur NULL
  - vérification des bornes de tableaux
  - race condition
  - ...
- outils : FindBugs, Astrée (AbsInt), Frama-C (CEA), Fortify, Zoncolan (Facebook), ...
- souvent incomplète, parfois incorrecte

Les méthodes formelles : Model-checking

- basé sur un système de transitions "fini" $\mathcal{P}$ et une spécification en logique temporelle $\mathcal{S}$
- algorithme de model-checking : $\mathcal{P} \models \mathcal{S}$
- outils : Spin, Uppaal, ...
- correct et complet, mais travaille sur une abstraction du programme

Les méthodes formelles : preuve de programmes

- Logiques pour la programmation : logique de Hoare, logique de séparation, ...

$$\{x > y\}\ x := x - y\ \{x > 0\}$$

- Instrumentalisation du code / annotation (notamment les invariants de boucle)
- utilisation de solveurs pour les formules logiques : SMT (Z3 - Microsoft), PVS, ...
- correct mais souvent incomplet

Outil : Frama-C (plugin WP), Why3, ...

```
/* @ ensures A: *a == \old(*b) ;
@ ensures B: *b == \old(*a) ;
@ */
void swap(int *a,int *b) ;
```

Les méthodes formelles : génération automatique

Génération (automatique) de programmes :

- méthode formelle B : raffinements successifs de la spécification vers le code (ligne 14 métro parisien)
- calcul des constructions ROCQ (anciennement COQ) : assistant de preuve basé sur une correspondance entre preuves (calcul des prédicats) et programmes (lambda-calcul typé d'ordre supérieur)
  - la spécification est un type d'ordre supérieur
  - la réalisation (preuve) du type est un programme vérifiant la spécification
  - mais il faut aider à la réalisation de la preuve

Section 3

Spécifications

Spécification : disclaimer

Survol du thème / ne se veut absolument pas exhaustif

- généralités sur la notion de spécification
- spécification abstraite algébrique
- la méthodologie UML

Spécification : vue d'ensemble

Ensemble de documents donnant une description stable, abstraite (orienté client/métier), la moins dépendante possible des contraintes liées au matériel, aux systèmes, à l'environnement.

Ceci comprend notamment :

- les attendus fonctionnels
- les attendus non-fonctionnels (performance, disponibilité, sécurité, ...)
- le contexte de fonctionnement
- ...

Décrit ce **qui doit être fait** (Spécification) et non comment (Code).

Spécification : vue d'ensemble (II)

Qualités d'une bonne spécification :

- claire, la moins ambigüe possible et cohérente
- la plus exhaustive et complète possible
- concise et au bon niveau d'abstraction,

Documents "contractuels" entre :

- le client / MOA (Maitrise d'OuvrAge) et la MOE (Maitrise d'Œuvre)
- l'analyste fonctionnel et le développeur

La spécification précède le développement (ce n'est pas la description du logiciel)

Différents types de spécification

- les spécifications informelles écrites en langage naturelle
- les spécifications semi-formelles écrites dans une syntaxe plus précise et normée et s'appuyant notamment sur des diagrammes d'un formalisme plus ou moins standardisés et souvent annotés (exemple : UML)
- les spécifications formelles écrites dans un formalisme possédant une syntaxe et une sémantique univoque

Spécifications en langage naturel

- compréhensible par tous les partenaires du projet
- source de potentielles ambiguïtés, voir de contradictions.
- différents niveaux de langage
  - de manière complètement libre, littéraire...
  - de manière très encadrée (structurée) par une méthode qui fournit un plan précis de ce qu'il faut décrire

Spécifications semi-formelles

- facilement compréhensible entre analyste fonctionnel et développeur (voir entre le client et l'équipe-projet)
- source de moins d'ambiguïté que la spécification en langue naturelle.
- pas complètement adapté pour un traitement automatisé (mais permet souvent le test et/ou la simulation)

Formalisme de description (de données)

- REGEXP : expressions régulières (présentes dans tous les langages de programmation)
- ABNF : Augmented Backus-Naur Form

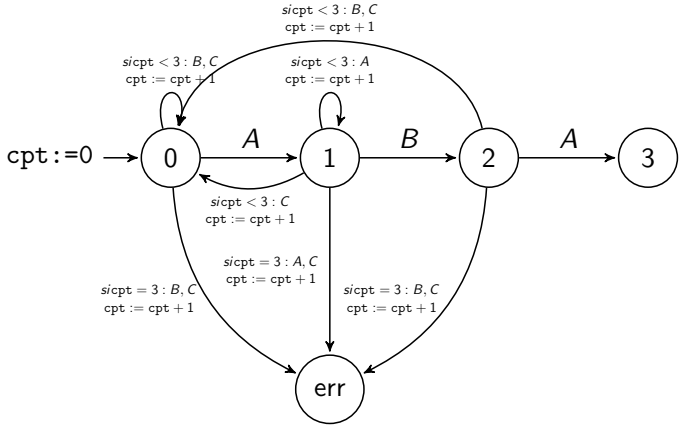
  grammaire (BNF) augmentée pour la définition des normes internet

  le format d'une date dans le RFC 2822 :

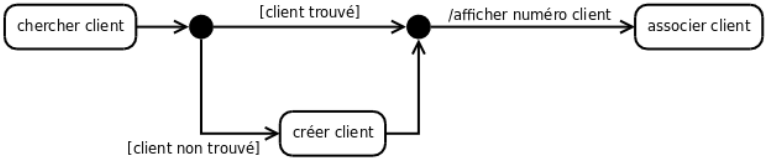
```
date            =       day month year
year            =       4*DIGIT / obs-year
month           =       (FWS month-name FWS) / obs-month
month-name      =       "Jan" / "Feb" / "Mar" / "Apr" /
                       "May" / "Jun" / "Jul" / "Aug" /
                       "Sep" / "Oct" / "Nov" / "Dec"
day             =       ([FWS] 1*2DIGIT) / obs-day
```

Automates

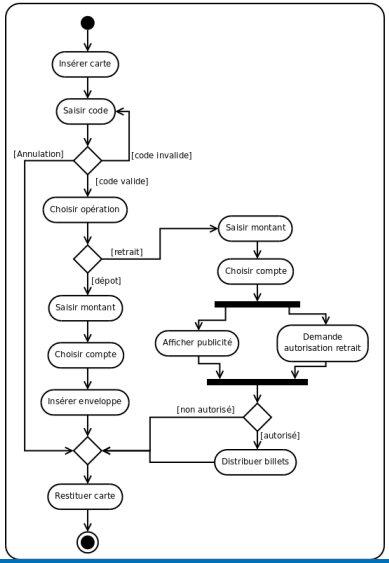
- Automates d'états finis : machine de Mealy, machine de Moore
- Automates étendus exemple : digicode A-B-A en 3 essais max



Diagrammes UML d'états-transitions ou d'activité



- diagramme états-transitions : automates à états finis
- diagramme d'activités : flux de contrôle (avec parallélisme possible)



Formalismes logiques

- logiques "génériques" interprétées sur un domaine spécifique : logique du premier ordre, logiques d'ordre supérieur,

$$\forall i, j \in [0, n-1], \quad i \leq j \Rightarrow T[i] \leq T[j]$$

- logiques dédiées à la spécification :
  - LTL : logique temporelle de temps discret

$$G(\text{req} \rightarrow F \text{ granted})$$

  - TLTL, MTL, MITL : logiques temporisées de temps continu

$$G(\text{req} \rightarrow F_{\leq 3} \text{ granted})$$

OCL (Object Constraint Language) au sein d'UML

- propriétés invariantes :

```
context Banque
 inv: not ( clients -> exists (age < 18) )
```

Il n'existe pas de clients de la banque dont l'age est inférieur à 18 ans

- définition de pré- et post-conditions

```
context Compte::débiter(somme : Integer)
 pre: somme > 0
 post: solde = solde@pre - somme
```

  - la somme à débiter doit être positive pour que l'appel de l'opération soit valide
  - après l'exécution de l'opération, l'attribut solde doit être débité du bon montant

Méthodes B et Z

Formalismes formels de spécification basés sur un langage de haut-niveau incluant la logique et les ensembles

- la spécification est précisée via un mécanisme de raffinement
- l'ultime étape de raffinement permet une implémentation directe.

Exemple B simple

```
MACHINE
   swap // échange les valeurs de deux variables xx et yy
VARIABLES
   xx, yy // déclarations des variables
INVARIANT
   xx : NAT & yy : NAT // les variables restent des naturels
INITIALISATION
   xx :: NAT || yy :: NAT // initialisation parallèle
OPERATIONS
   echange =
   BEGIN
       xx := yy || yy := xx
   END
END
```