République Algérienne Démocratique et Populaire

Ministère de l'Enseignement Supérieur et de la Recherche Scientifique

Université 20 Aout 1955-SKIKDA

Faculté des sciences

Département d'Informatique

Mémoire fin d'études

Pour l'obtention de diplôme de MASTER en informatique

Option : Génie logiciel avancé et applications

THEME

Une approche de test basée sur les diagrammes d'états finis

Présenté par

BRAHIMI Hadjer

KENEF Ikram

Encadré par

Dr. KISSOUM Y

Année Universitaire 2022/2023

Remerciement

قال الله تعالى:"وإذ تأذن ربكم لئن شكرتم لأزيدنكم ولئن كفرتم إن عذابي لشديد"

صدق الله العظيم **(سورة إبراهيم , الآية7)**

« (Et lorsque votre Seigneur proclama : "Si vous êtes reconnaissants, très certainement J'augmenterai [Mes bienfaits] pour vous) ».

[Coran S14.V7]

Nous remercions tout d'abord le bon ***DIEU*** pour nous avoir donnée le courage et la santé pour accomplir ce travail.

«(Celui qui ne remercie pas les gens, ne remercie pas Allah.)»

[Authentique Hadith]

Ce travail ne serait pas aussi riche et n'aurait pas pu avoir le jour sans l'aide et l'encadrement de ***Dr KISSOUM.Y*** on le remercie pour la qualité de son encadrement exceptionnel, pour sa patience, sa rigueur et sa disponibilité durant notre préparation de ce mémoire.

Notre remerciement s'adresse également à tous les membres du ***jury***, qui ont accepté d'évaluer ce travail.

Nos profonds remerciements vont également à toutes les personnes qui nous aidés et soutenue de près ou de loin.

Toute notre gratitude à vous.

Dédicace

Je dédie ce travail :

A mes très chers ***parents*** qui m'ont transmit la vie, l'amour et le courage ;

A ma ***sœur*** Sara et mes ***frères*** Mohamed, Islam et Abderrahmane, merci d'être dans ma vie ;

A mes chères ***nièces*** Nourcine, Nourhane et mes ***neveux*** Salam et Mohaimen ;

A ma ***grand-mère*** Warda ;

A mes ***grands-pères*** Brahim et Wardi ;

A mes ***oncles*** et mes ***tantes*** et ses enfants ;

A toute ***ma famille*** grand et petit ;

A mon fiancé

A mon binôme ***Ikram*** pour la sœur agréable qu'elle était et qu'elle restera pour moi, et a toute sa famille « kenef » ;

A mes chères ***amies*** de la faculté Maya et Zahra, merci pour les beaux moments passés ensemble ;

A mes professeurs de la faculté ;

Hadjer

Dédicace

Je remercie ALLAH qui m'a donné la santé, la patience et la volonté pour arriver à ce stade et réaliser ce travail.

A la mémoire de mes parents J'aurais tant aimé que vous soyez présent que dieu ait vos âme dans sa sainte miséricorde.

A mon frère YAAKOUB merci pour ton soutien quotidien, ta force intérieure, merci de rendre ma vie éclair en absence de mes parents.

A mes sœurs Asma, Sana merci pour l'amour, le soutien et leur présence toujours à côté de moi.

À ma chère sœur ANISSA mon exemple de vie merci pour l'amour et la soutien, merci d'être toujours à côté de moi.

A mes chers nièces AYA, ALAA, RAHIL et mon neveu YOUNES.

A mon unique tante SIHAM merci des sacrifices et du courage dont vous avez fait pour moi.

A mon cousin RAMI merci beaucoup d'être toujours à côté de moi.

A mon binôme Hadjer plus de 12 ans qu'on préserver cette amitié pourvu que cela dure encore. Merci d'avoir été celle que tu es un binôme génial, tu as et tu restes une superbe rencontre, merci à tout la famille Brahimi.

A mes amis de la fac Zahra, Maya merci pour les moments inoubliable passé ensemble puisse dieu renforce les liens d'amitié qui nous unissent.

A ma belle Fatima merci devenir vrais cousine merci pour tout ce que ce que tu as fait pour moi.

A tous ceux qui ont contribué de près ou de loin pour que ce travail soit possible je vous dis merci.

Ikram

Résumé

La stratégie Round-Trip Path (RTP) est connue pour générer la suite de tests basée sur l'état la plus rentable. Cependant, certaines études ont insisté sur le fait que les suites ne suffisent pas. Ils ont souligné que les suites RTP ne testent pas certains éléments réalistes scénarios, ce qui pourrait entraîner une mauvaise détection des défauts. Un algorithme de parcours empêche inévitablement les suites de couvrant certaines paires d'événements, ce qui pourrait être un facteur qui provoque des omissions de test. Par conséquent, nous étudions comment beaucoup et quelles paires d'événements la suite ne couvre pas. Ensuite, nous déterminons si l'augmentation de la suite pour couvrir toutes les paires d'événements peuvent améliorer l'efficacité de manière rentable. À cette fin, des paires d'événements sont identifiées. Ensuite, les paires d'événements couvertes sont dérivées d'un arbre RTP. Ensuite, les paires d'événements à couvrir sont calculées. Enfin, l'arbre est augmenté pour couvrir les paires de transitions que les paires d'événements à couvrir peuvent déclencher.

Mots-clés : Tests basés sur l'état ; machine d'état UML ; Suite de test de chemin aller-retour ; Paires d'événements ; Scénario d'utilisation Essai.

ملخص

من المعروف ان استيراتيجية مسار الرحلة ذهابا وايابا (RTP) تنشئ مجموعة الاختبار الاكثر فعالية من حيث التكلفة المعتمدة على الحالة . ومع ذلك اصرت بعض الدراسات على أن التتابعات ليست كافية. وأشاروا إلى أن مجموعات هذه الاستيراتيجية لا تختبر بعض السيناريوهات الواقعية ، مما قد يؤدي إلى ضعف اكتشاف العيوب. تمنع خوارزمية الاجتياز حتمًا التسلسلات من امتداد أزواج معينة من الأحداث ، والتي يمكن أن تكون عاملاً يسبب إغفال الاختبار. لذلك ، نقوم بدراسة عدد وأزواج الأحداث التي لا يغطيها التكملة. بعد ذلك ، نحدد ما إذا كانت زيادة التسلسل لتغطية جميع أزواج الأحداث يمكن أن تحسن الكفاءة بطريقة فعالة من حيث التكلفة. لهذا الغرض ، يتم تحديد أزواج من الأحداث. بعد ذلك ، يتم اشتقاق أزواج الأحداث المغطاة من الشجرة . بعد ذلك ، يتم حساب أزواج الأحداث المراد تغطيتها. أخيرًا ، تتم زيادة الشجرة لتغطية أزواج التحولات التي يمكن أن تطلقها أزواج الأحداث المراد تغطيتها.

كلمات مفتاحية

آلة الحالة UML؛ اختبار على أساس الحالة ؛ تسلسل اختبار المسار ذهابًا وإيابًا ؛ أزواج الحدث؛ حالة الاستخدام التجريبي.

Abstract

The Round-Trip Path (RTP) strategy is known to generate the most cost-effective state-based test suite. However, some studies have insisted that suites are not sufficient. They pointed out that RTP suites do not test some realistic scenarios, which could lead to poor fault detection. A traversal algorithm inevitably prevents the suites from covering some event-pairs, which could be a factor that causes test omissions. Therefore, we investigate how many and which event-pairs the suite does not cover. Then, we determine whether augmenting the suite to cover all event-pairs can cost-effectively improve the effectiveness. To this end, event-pairs are identified. Next, eventpairs covered are derived from an RTP tree. Then, event-pairs to be covered are calculated. Finally, the tree is augmented to cover the transition-pairs that event-pairs to be covered can trigger.

Keywords: State-Based Testing; UML State Machine; Round-Trip Path Test Suite; Event-Pairs; Usage Scenario Testing.

Table de matières

Titre **Page**

Introduction Générale **1**

Chapitre 01 : Généralité sur le test

1. Introduction **3**

2. Définition du test **3**

3. Définition du logiciel **3**

4. Que signifie le test de logiciels ? **5**

5. Historique du test logiciel **5**

6. Pourquoi le test logiciel est-il important ? **6**

7. Cycle de vie de test de logiciel **7**

7.1 Qu'est-ce que le cycle de vie? **7**

7.2 Que signifie le cycle de vie des tests logiciels (STLC) ? **7**

7.3 Caractéristiques du cycle de vie des tests logiciels **7**

7.4 Les phases de cycle de vie de test logiciel **7**

7.4.1 Phase d'analyse des exigences **8**

7.4.2 Phase de planification des tests **8**

7.4.3 Phase de développement des cas de tests logiciel **8**

7.4.4 Phase de préparation de l'environnement de test logiciel **9**

7.4.5 Phase d'exécution des tests **9**

7.4.6 Phase des activités de clôture de test logiciel **9**

8. Les différents types de tests **9**

8.1 Tests unitaires **10**

8.2 Tests de module **10**

8.3 Tests d'intégration **10**

8.4 Tests systèmes **11**

8.5 Tests fonctionnels **11**

8.6 Tests de robustesse **11**

8.7 Tests de performance **11**

8.8 Tests d'ergonomie **12**

8.9 Tests de sûreté **12**

8.10 Tests de sécurité **12**

8.11 Test de la boite noire **13**

8.12 Test de la boite blanche 13
9. Conclusion 13

Chapitre 02 : Test basé sur les modèles

1. Introduction 14
2. Model-based Testing 15
2.1 Définition 15
2.2 Avantages économiques du MBT 16
2.3 Processus MBT 16
2.4 Aspects modélisation 18
2.5 Outils de Model-based Testing 19
2.5.1 MaTeLo 19
2.5.2 JSXM 20
2.5.3 Qtronic 20
2.5.4 SpecExplorer 20
2.6 Différents types de modèles pour le Model-Based Testing 21
2.6.1 Machine à états finis 21
2.6.2 UML/OCL 23
3. Test basé sur l'état 25
3.1 Définition 25
3.2 Quand utiliser les tests de transition d'état? 26
3.3 Technique 26
3.3.1 Application 26
3.3.2 Limitation/difficultés/risque 26
3.3.3 Couverture 26
3.3.4 Type de défaut 27
3.3.5 Mise en œuvre 27
3.4 Diagramme d'état 28
4. Conclusion 28

Chapitre 03 : approche proposée

1. Introduction 29
2. Couvertures du test utilisé dans le SBT 29
2.1 Stratégie RTP 29
2.2 Stratégie MTT 30
2.3 Stratégie OTT 30

2.4 Stratégie RTP+SEP & RTP+FEP 31
3. Approche proposée 31
3.1 Algorithme utilisé pour générer une suite de tests à partir d'un arbre de test 31
3.2 Etapes pour augmenter un arbre RTP 32
3.2.1 Étape 1 : Identifiez les paires d'événements 33
3.2.2 Étape 2 : Dériver les paires d'événements couverts 34
3.2.3 Étape 3 : Calculer les paires d'événements à couvrir 34
3.2.4 Étape 4 : Augmenter un arbre RTP 34
4. Conclusion 36
Chapitre 04 : Etude de cas et implémentation
1. Introduction 37
2. Etude de cas 37
2.1 Conception d'études de cas 37
2.1.1 Logiciel sous test de test 37
2.1.2 Test de mutation 41
3. Conclusion 41
Conclusion générale 42

Liste des figures

Chapitre 01 : Généralité sur le test

Figure 1.1 : Démarche de construction d'un logiciel **4**

Figure 1.2 : Représentation des différentes phases du cycle de vie des tests logiciels **8**

Figure 1.3 : Différents types de tests **10**

Chapitre 02 : Test basé sur les modèles

Figure 2.1 : Cycle en V **15**

Figure 2.2 : Processus de model-based testing **17**

Figure 2.3 : Graphe orienté d'un portillon d'accès **22**

Figure 2.4 : Machine à café déterministe (à gauche) et non déterministe (à droite) **23**

Figure 2.5 : Les différents diagrammes UML **25**

Chapitre 03 : approche proposée

Figure 3.1 : arbre RTP construit à partir d'une machine d'état finis **30**

Figure 3.2 : Procédure d'augmentation d'un arbre RTP pour générer une suite augmentée **33**

Figure 3.3 : Chaque arbre augmenté de notre exemple **35**

Chapitre 04 : Etude de cas et implémentation

Figure 4.1 : diagramme d'état de CCS **38**

Figure 4.2 : Interface ModelJUnit **39**

Figure 4.3 : Interface de l'application **40**

Liste des tableaux

Chapitre 03 : approche proposée

Tableau 3.1 : paires d'événements du notre exemple **34**

Tableau 3.2 : chaque ep2bc de machine d'état et son TP correspondant **35**

Chapitre 04 : Etude de cas et implémentation

Tableau 4.1 : Principaux attributs de la machine d'état et implémentation des SUT **40**

Introduction Générale

Le test en informatique désigne une procédure de vérification partielle d'un système. Son objectif principal est d'identifier un nombre maximal de comportements problématiques du logiciel. Il permet ainsi, dès lors que les problèmes identifiés seront corrigés, d'en augmenter la qualité.

D'une manière plus générale, le test désigne toutes les activités qui consistent à rechercher des informations quant à la qualité du système afin de permettre la prise de décisions.

Un test ressemble à une expérience scientifique. Il examine une hypothèse exprimée en fonction de trois éléments : les données en entrée, l'objet à tester et les observations attendues. Cet examen est effectué sous conditions contrôlées pour pouvoir tirer des conclusions et, dans l'idéal, être reproduit.

Le test basé modèle est une activité qui permet de concevoir et de dériver (de manière automatique ou non) des cas de tests à partir d'un modèle abstrait et haut niveau du système sous test (SUT). Le modèle est dit abstrait car il offre bien souvent une vue partielle et discrète des comportements attendus d'un logiciel ou d'un système.

Sur la base de modèles abstraits, des cas de test peuvent être dérivées sous la forme de suites de tests. Ces suites de tests ne sont pas directement exécutables, car elles n'ont pas le même niveau d'abstraction que le code exécutable. Cela demande souvent une intervention manuelle de la part d'un ingénieur de test qui doit concevoir une couche d'adaptation permettant de passer d'une suite de tests abstraites en suite de tests exécutables. Cette étape est généralement appelée étape de concrétisation.

Une fois les cas de tests exécutés, une comparaison est possible entre le comportement réel du logiciel (le logiciel développé) et le comportement attendu (décrit dans le modèle). La comparaison entre ce qui est attendue et ce qui se passe réellement permet d'assigner un verdict de test. Un test est dit non-passant lorsque le comportement réel du logiciel, ou du système, diffère du comportement attendu.

Par ailleurs, les tests de mutation sont utilisés pour concevoir de nouveaux tests logiciels et évaluer la qualité des tests logiciels existants. Les tests de mutation impliquent de modifier un programme de manière mineure. Chaque version mutée est appelée mutant et les tests détectent et rejettent les mutants en faisant en sorte que le comportement de la version originale diffère de celui du mutant. C'est ce qu'on appelle tué le mutant. Les suites de tests sont mesurées par le pourcentage de mutants qu'elles tuent. De nouveaux tests peuvent être conçus pour tuer des mutants supplémentaires. Les mutants sont basés sur des opérateurs de mutation bien définis qui imitent les erreurs de programmation typiques (comme l'utilisation du mauvais opérateur ou du mauvais nom de variable)

ou forcent la création de tests utiles (comme la division de chaque expression par zéro). Le but est d'aider le testeur à développer des tests efficaces ou à localiser les faiblesses dans les données de test utilisées pour le programme ou dans les sections du code qui sont rarement ou jamais consultées pendant l'exécution. Les tests de mutation sont une forme de test en boîte blanche

Une définition plus générale de l'analyse des mutations consiste à utiliser des règles bien définies sur des structures syntaxiques pour apporter des modifications systématiques aux artefacts logiciels. L'analyse des mutations a été appliquée à d'autres problèmes, mais elle est généralement appliquée aux tests. Ainsi, les tests de mutation sont définis comme l'utilisation de l'analyse de mutation pour concevoir de nouveaux tests logiciels ou pour évaluer des tests logiciels existants. Ainsi, l'analyse et les tests de mutation peuvent être appliqués aux modèles de conception, aux spécifications, aux bases de données, aux tests, au XML et à d'autres types d'artefacts logiciels, bien que la mutation de programme soit la plus courante.

Il existe plusieurs méthodes et techniques pour le test de mutation, Parmi ces techniques, nous trouvons le test basé sur les modèles, il est basé sur un modèle de système afin de produire des cas de test abstraits. Pour que ces derniers puissent être soumis au système sous test, les cas de test abstraits doivent être transformés en des cas de test concrets. Notre approche consiste à appliquer le test basé afin de générer les cas de test automatique basé sur le diagramme d'état.

Le présent mémoire est organisé en quatre chapitres structurés comme suit :

- **Généralités sur le test :** une synthèse de l'état de l'art sur le test de logiciel est décrite dans ce chapitre qui se compose de plusieurs parties liées à des informations générales sur le test de logiciel à savoir : sa définition, son historique, son cycle de vie, ses différents types.
- **Test basé sur les modèles :** une synthèse de l'état de l'art sur le test basé sur les modèles décrite dans ce chapitre qui se compose de deux parties :
  - ✓ La première partie concerne le test basé sur les modèles à savoir : sa Définition, ses avantages, son Processus, ses Aspects modélisation, ses Outils, ses différents types.
  - ✓ La deuxième partie concerne le test basé sur l'état à savoir : sa définition, ses Technique.
- **L'approche proposée :** dans ce chapitre on présente notre approche proposée la génération de cas de test basé sur le diagramme d'état.
- **Etude de cas et implémentation :** dans ce dernier chapitre on présente la conception et le résultat de l'étude de cas ainsi que l'environnement et les outils déployés pour le développement de notre application.

Chapitre 01

Généralités sur le test

Chapitre 01 : Généralité sur le test

1. Introduction

Durant le développement et la maintenance d'un logiciel, l'activité de validation requiert une attention particulière de la part du développeur du logiciel. Elle exige, certes, un coût mais elle est nécessaire pour l'assurance de la qualité du logiciel. L'ignorance de cette activité de validation peut se traduire par des pertes financières, humaines, etc. Au-delà des pertes humaines et matérielles, la correction des défauts se ferait avec des couts élevés, d'où l'importance de la validation du logiciel avant sa mise sur le marché. Parmi les moyens de la validation et la vérification des logiciels, se trouve le "test" qui est apparu dans les années 50 et qui a depuis largement prouvé son intérêt économique et qualitatif. Depuis sa création, il s'est imposé comme étant le moyen principal pour la validation du fonctionnement d'un programme. Il a pour objectif d'examiner ou d'exécuter un programme dans le but d'y révéler des erreurs, ce qui augmente la confiance dans le logiciel. Il est souvent défini comme le moyen par lequel on s'assure qu'une implantation est conforme à ce qui a été spécifié. **[1]**

2. Définition du test

Selon l'IEEE (Standard Glossary of Software Engineering Terminology). Le test est la performance ou l'évaluation automatique ou manuelle d'un système ou d'un composant pour s'assurer qu'il satisfait à ses exigences ou pour trouver des variations entre les résultats attendus et réels. **[2]**

3. Définition du logiciel

En informatique, un logiciel est un ensemble de séquences d'instructions interprétables par une machine et d'un jeu de données nécessaires à ces opérations. Le logiciel détermine donc les tâches qui peuvent être effectuées par la machine, ordonne son fonctionnement et lui procure ainsi son utilité fonctionnelle. Les séquences d'instructions appelées programmes ainsi que les données du logiciel sont ordinairement structurées en fichiers. La mise en œuvre des instructions du logiciel est appelée exécution et la machine chargée de cette mise en œuvre est appelée ordinateur ou calculateur. **[3]**

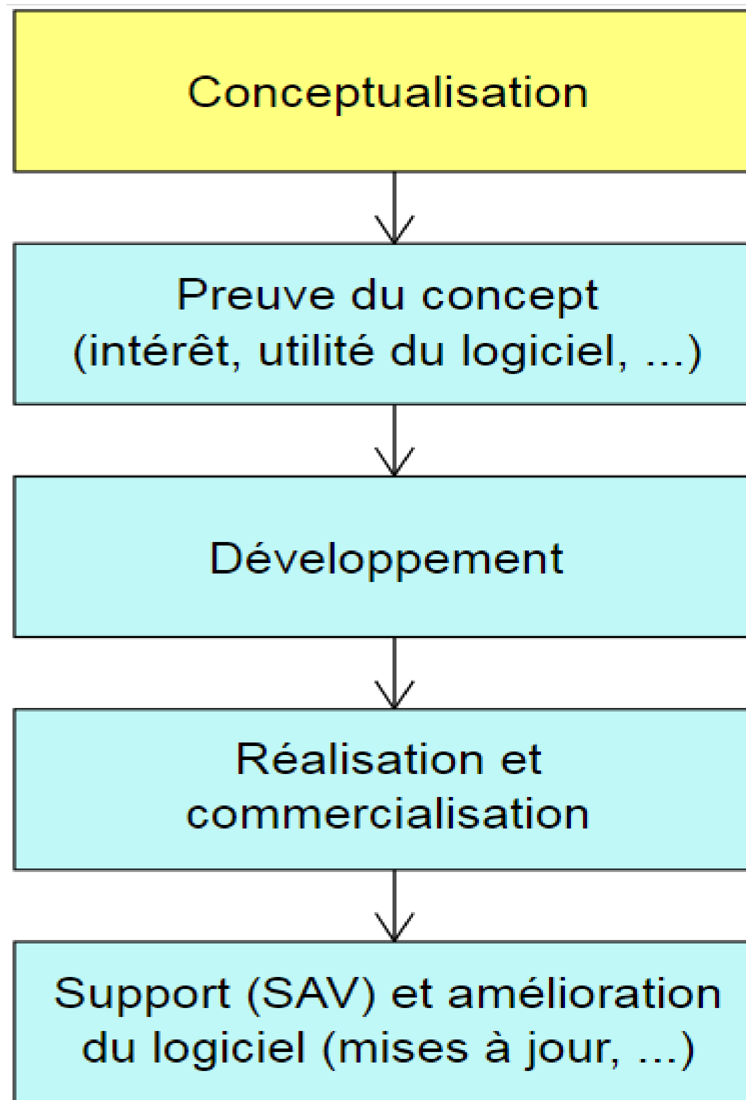

Figure 1.1 : Démarche de construction d'un logiciel

Un logiciel peut être classé comme système, applicatif, standard, spécifique, ou libre, selon la manière dont il interagit avec le matériel, selon la stratégie commerciale et selon les droits sur le code source des programmes. Le terme logiciel propriétaire est aussi employé.

Les logiciels sont créés et livrés à la demande d'un client ou sur l'initiative du producteur, et mis sur le marché, parfois gratuitement. En 1980, 60 % de la production et 52 % de la consommation mondiale de logiciels est aux États-Unis. Les logiciels sont aussi distribués illégalement et la valeur marchande des produits ainsi distribués est parfois supérieure au chiffre d'affaires des producteurs. Les logiciels libres sont créés et distribués comme des commodités produites par coopération entre les utilisateurs et les auteurs.

4. Que signifie le test de logiciels ?

Les tests de logiciels sont un ensemble de processus visant à rechercher, évaluer et vérifier l'exhaustivité et la qualité des logiciels informatiques. Les tests de logiciels garantissent la conformité d'un produit logiciel par rapport aux exigences réglementaires, commerciales, techniques, fonctionnelles et utilisateur.

Les tests de logiciels sont également appelés tests d'applications.

Les tests de logiciels sont avant tout un vaste processus composé de plusieurs processus liés entre eux. L'objectif principal des tests de logiciels est de mesurer la santé des logiciels ainsi que leur exhaustivité en termes d'exigences essentielles. Les tests de logiciels consistent à examiner et à vérifier les logiciels à travers différents processus de test. Les objectifs de ces processus peuvent comprendre:

- Vérification de l'exhaustivité du logiciel par rapport aux exigences fonctionnelles / commerciales
- Identifier les bugs / erreurs techniques et s'assurer que le logiciel est exempt d'erreurs
- Évaluation de l'utilisabilité, des performances, de la sécurité, de la localisation, de la compatibilité et de l'installation

Le logiciel testé doit réussir chacun des tests afin d'être considéré comme complet ou apte à être utilisé. Certains des différents types de méthodes de test de logiciels incluent les tests en boîte blanche, les tests en boîte noire et les tests en boîte grise. De plus, le logiciel peut être testé dans son ensemble, dans des composants / unités ou dans un système sous tension. **[4]**

5. Historique du test logiciel

Le test logiciel est arrivé en même temps que le développement logiciel, qui a débuté juste après la seconde guerre mondiale. C'est à l'informaticien Tom Kilburn que l'on doit l'écriture du premier logiciel qui a été lancé le 21 juin 1948 à l'Université de Manchester, en Angleterre. Il effectuait des calculs mathématiques à l'aide d'instructions en code machine.

Le débogage était la principale méthode de test à l'époque et il l'est resté pendant les deux décennies suivantes. Dans les années 1980, les équipes de développement ne se contentaient plus d'isoler et de corriger les bogues des logiciels, mais elles testaient les applications dans des conditions réelles. Cela a ouvert la voie à une vision plus large des tests qui englobaient un processus d'assurance qualité faisant partie du cycle de vie du développement logiciel.

« Dans les années 1990, il y a eu une transition des tests vers un processus plus complet appelé assurance qualité, qui couvre l'ensemble du cycle de développement logiciel et affecte les processus de planification, de conception, de création et d'exécution des cas de test, le support des cas de test existants et les environnements de test », déclare Alexander Yaroshko dans son article sur le site des développeurs uTest.

« Les tests avaient atteint un niveau qualitativement nouveau, ce qui a conduit au perfectionnement des méthodologies, à l'émergence d'outils puissants de gestion du processus de test et d'outils d'automatisation des tests. » **[5]**

6. Pourquoi le test logiciel est-il important ?

Peu de gens peuvent s'opposer à la nécessité d'un contrôle qualité lors du développement logiciel. Les retards de livraison ou les défauts logiciels peuvent nuire à la réputation d'une marque, et entraîner la frustration et la perte de clients. Dans des cas extrêmes, un bogue ou un défaut peut dégrader les systèmes interconnectés ou provoquer de graves dysfonctionnements.

Prenons l'exemple de Nissan qui a dû rappeler plus d'un million de voitures en raison d'un défaut de logiciel dans les capteurs d'airbags. Ou celui d'un bogue logiciel qui a provoqué l'échec du lancement d'un satellite militaire d'un montant de 1,2 milliard de dollars. 2 Les chiffres parlent d'eux-mêmes. Les défaillances logicielles aux États-Unis ont coûté à l'économie 1,1 billion de dollars d'actifs en 2016. De plus, elles ont touché 4,4 milliards de clients.

Bien que les tests eux-mêmes sont coûteux, les entreprises peuvent économiser des millions par an en développement et en support si elles disposent d'une bonne technique de test et de processus d'assurance qualité. Les tests logiciels précoces révèlent les problèmes avant qu'un produit ne soit mis sur le marché. Plus vite les équipes de développement reçoivent des informations sur les tests, plus vite elles peuvent résoudre des problèmes comme ceux ci-dessous :

_Défauts architecturaux

_Mauvaises décisions de conception

_Fonctionnalité non valide ou incorrecte

_Vulnérabilités de sécurité

_Problèmes d'évolutivité

Lorsque le développement laisse une large place aux tests, il permet d'améliorer la fiabilité des logiciels et de livrer des applications de qualité avec peu d'erreurs. Un système qui répond aux attentes des clients, voire les dépasse, permet d'augmenter les ventes et la part de marché. **[5]**

7. Cycle de vie de test de logiciel

7.1 Qu'est-ce que le cycle de vie ?

En termes simples, le cycle de vie désigne la séquence de changements d'une forme à une autre. Ces changements peuvent concerner n'importe quelle chose physique ou immatérielle. Toute entité a un cycle de vie, depuis sa création jusqu'à son retrait ou sa disparition. De la même manière, le logiciel est également une entité. Tout comme le développement d'un logiciel implique une séquence d'étapes, les tests ont également des étapes exécutées dans un ordre précis.

7.2 Que signifie le cycle de vie des tests logiciels (STLC) ?

A Software Test Life Cycle (STLC) est désigne un processus de test pour tester des produits logiciels. Les tests logiciels sont une partie essentielle de la préparation des logiciels pour l'utilisation. Un STLC contribue comporte des étapes spécifiques à exécuter dans un ordre précis afin de garantir que les objectifs de qualité ont été atteints.

7.3 Caractéristiques du cycle de vie des tests logiciels

C'est une partie fondamentale du cycle de vie du développement logiciel (SDLC), mais il ne comprend que les phases de test. Aussi, il commence dès que les exigences sont définies ou que le document d'exigences du logiciel est partagé par les parties prenantes. STLC donne un processus étape par étape pour assurer la qualité du logiciel.

7.4 Les phases de cycle de vie de test logiciel

Dans le processus STLC, chaque activité s'effectue de manière planifiée et systématique. Chaque phase a des objectifs et des produits livrables différents. Les différentes organisations ont des phases différentes dans le STLC, mais la base reste la même.

Voici les phases de la STLC :

1. Comprendre les exigences
2. Plan de test
3. Développement des cas de test
4. Préparer un environnement de test
5. Exécution des tests

6. Les activités de clôture des tests

Figure 1.2 : Représentation des différentes phases du cycle de vie des tests logiciels

7.4.1 Phase d'analyse des exigences

La phase de test des exigences, également connue sous le nom d'analyse des exigences, est la première étape du cycle de vie des tests logiciels (STLC). Dans cette phase, l'équipe d'assurance qualité comprend les besoins comme les cas de test. Elle peut interagir avec diverses parties prenantes pour comprendre les exigences en détail. Ainsi, les exigences peuvent être fonctionnelles ou non fonctionnelles. Dans cet esprit, la faisabilité de l'automatisation du projet de test se détermine à ce stade.

7.4.2 Phase de planification des tests

La planification des tests est la phase la plus efficace du cycle de vie des tests logiciels, où l'identification de tous les plans de test. Dans cette phase, l'équipe calcule l'effort et le coût estimés pour le travail de test. Cette phase commence une fois que la phase de collecte des exigences se termine.

Sur quelle base est la planification ? Seulement les exigences ?

La réponse est **NON**. Les exigences constituent l'une des bases, mais il existe deux autres facteurs très importants qui influencent la planification des tests. Ce sont :

– La stratégie de test de l'organisation.

– L'analyse et la gestion des risques et leur atténuation.

7.4.3 Phase de développement des cas de tests logiciel

La phase de développement des cas de test implique la création, la vérification et la réécriture des cas de test et des scripts de test une fois que le plan de test est prêt. Au départ, les données de test sont identifiées, puis créées et révisées, avant d'être retravaillées en fonction des conditions préalables. Ensuite, l'équipe AQ commence le processus de développement des cas de test pour les unités individuelles (le test unitaire).

7.4.4 Phase de préparation de l'environnement de test logiciel

La configuration de l'environnement de test est une partie essentielle du STLC. L'environnement de test détermine les conditions dans lesquelles le testeur logiciel teste. Il s'agit d'une activité indépendante qui peut se lancer au même temps que le développement des cas de test. Dans ce processus, l'équipe de test n'est pas responsable. C'est le développeur ou le client qui crée l'environnement de test. Quoi que, l'équipe de test doit effectuer un contrôle de l'état de préparation (smoke testing) de l'environnement donné.

7.4.5 Phase d'exécution des tests

Après le développement du scénario de test et la configuration de l'environnement de test, la phase d'exécution du test commence. Le processus consiste en l'exécution de scripts de test, la maintenance des scripts de test et le signalement des bogues. S'il y a des bogues, ils sont renvoyés à l'équipe de développement pour correction et de nouveaux tests sont effectués.

7.4.6 Phase des activités de clôture de test logiciel

Une fois que tous les défauts sont corrigés, le rapport de synthèse du test sera préparé par le test manager et partagé avec le client pour son acceptation. En conclusion, les produits livrables de la clôture du cycle de test sont :

- Rapport de clôture des tests
- Métriques des tests

C'est l'activité de clôture des tests. **[6]**

8. Les différents types de tests

Il existe de nombreux types de tests de logiciels, chacun ayant des objectifs et des stratégies spécifiques :

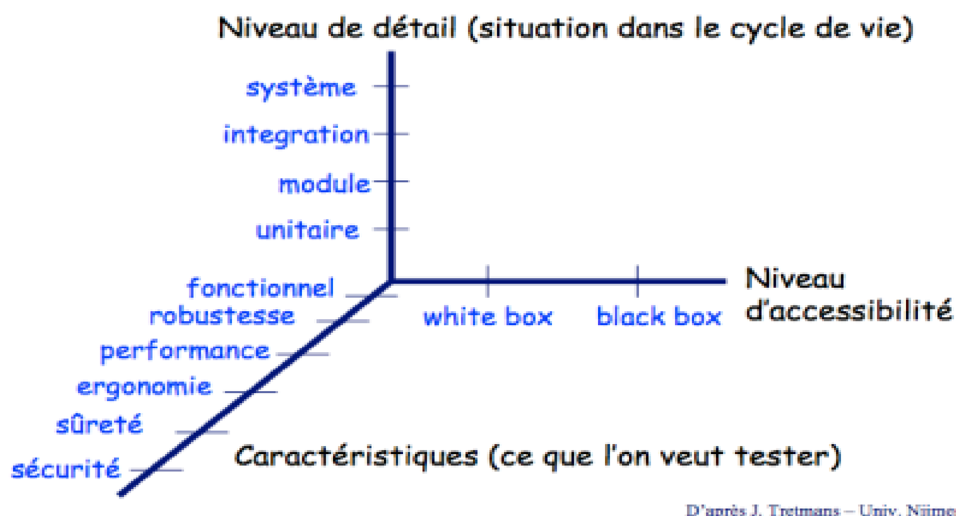

Figure 1.3 : Différents types de tests

8.1 Tests unitaires

Les tests unitaires sont de très bas niveau, près de la source de votre application. Ils consistent à tester les méthodes et fonctions individuelles des classes, des composants ou des modules utilisés par votre logiciel. Les tests unitaires sont en général assez bon marché à automatiser et peuvent être exécutés très rapidement par un serveur d'intégration continue. **[7]**

8.2 Tests de module

Le test de module est défini comme un type de test de logiciel, qui vérifie des sous-programmes, des sous-programmes, des classes ou des procédures individuelles dans un programme. Au lieu de tester l'ensemble du logiciel en une seule fois, les tests de module recommandent de tester les plus petits blocs de construction du programme.

Les tests de module sont en grande partie orientés vers une boîte blanche. L'objectif de faire des tests de module n'est pas de démontrer le bon fonctionnement du module mais de démontrer la présence d'une erreur dans le module.

Les tests au niveau des modules permettent d'implémenter le parallélisme dans le processus de test en donnant la possibilité de tester plusieurs modules simultanément. **[8]**

8.3 Tests d'intégration

Les tests d'intégration vérifient que les différents modules ou services utilisés par votre application fonctionnent bien ensemble. Par exemple, ils peuvent tester l'interaction avec la base de données ou s'assurer que les micros services fonctionnent ensemble comme prévu. Ces types de tests sont plus coûteux à exécuter, car ils nécessitent que plusieurs parties de l'application soient fonctionnelles. **[7]**

8.4 Tests systèmes

À ce niveau, on exécute plusieurs scénarios complets qui constituent les cas d'utilisation du logiciel. Dans le jargon du test informatique, on le qualifie de test de type boîte noire et ils permettent de s'assurer de la fonctionnalité globale du logiciel et de son comportement sur les terminaux d'utilisation. Pour aboutir à des reportings objectifs pour ce type de test, l'équipe qui en a la charge doit se distinguer et assurer une totale indépendance vis-à-vis des équipes de développement. **[9]**

8.5 Tests fonctionnels

Les tests fonctionnels se concentrent sur les exigences métier d'une application. Ils vérifient uniquement la sortie d'une action et non les états intermédiaires du système lors de l'exécution de cette action.

Il y a parfois une certaine confusion entre les tests d'intégration et les tests fonctionnels, car ils nécessitent tous les deux plusieurs composants pour interagir. La différence réside dans le fait qu'un test d'intégration peut simplement vérifier que vous pouvez interroger la base de données, tandis qu'un test fonctionnel s'attend à obtenir une valeur spécifique de la base de données, telle que définie par les exigences du produit. **[7]**

8.6 Tests de robustesse

Ils ouvrent le bal des tests non fonctionnels. Ici, on soumet votre logiciel à une forte activité, ceci pour vérifier qu'elle peut fonctionner de façon optimale sous pression, ce test de robustesse permet également d'en découvrir les limites, ces paramètres sont des données importantes pour l'utilisation et la vie future du logiciel. Les exécutables sont répétés à toutes les étapes des tests du logiciel, ils confirment son aptitude à être disponible et performant même si un grand nombre d'utilisateurs y accèdent simultanément.

Ces tests informatiques intègrent une longue liste de critères qui ne sont pas forcément liés à la fonction initiale du logiciel. Mais il est important de prévoir le comportement du logiciel dans ces cas "extrêmes" pour anticiper et garantir en amont une expérience utilisateur au pire acceptable et au mieux très positive. **[9]**

8.7 Tests de performance

Les tests de performance évaluent les performances d'un système sous une charge de travail spécifique. Ces tests permettent de mesurer la fiabilité, la vitesse, l'évolutivité et la réactivité d'une application. Par exemple, un test de performance peut observer les temps de réponse lors de l'exécution d'un nombre important de demandes ou déterminer le comportement du système face à

une quantité élevée de données. Il peut déterminer si une application répond aux exigences de performances, localiser les goulots d'étranglement, mesurer la stabilité pendant les pics de trafic, et plus encore. **[7]**

8.8 Tests d'ergonomie

Le but ici est presque émotionnel, tenter de palper au plus juste quel sera le sentiment de l'utilisateur à travers l'interface du logiciel : on parle alors d'expérience utilisateur ou UX en anglais. Seront testés ici les caractères externes, tels que le design, voir si les propositions des commandes sont intuitives, il faudra ici toucher à des paramètres subjectifs tels que l'esthétique. Bien qu'ils soient peu courants, ces tests valent toutefois leur pesant d'or. En d'autres termes, ils peuvent se révéler comme un facteur déterminant de qualité visuelle et de l'expérience utilisateur. C'est pourquoi il convient d'associer à cette étape les utilisateurs finaux et/ou des ergonomes ou UX designers lors de la phase de conception notamment. **[9]**

8.9 Tests de sûreté

La sûreté de fonctionnement est la combinaison de différents facteurs [Laprie 1996]

- la fiabilité ;
- la maintenabilité ;
- la disponibilité et les temps de réponse ;
- la confidentialité ;
- l'intégrité : impossibilité d'altération des informations ;
- la sécurité et l'innocuité : absence de conséquences catastrophiques - la prévention des incidents graves issue d'une analyse de risques.

Dans un premier temps, on peut percevoir la sûreté de fonctionnement comme l'empilement des tranches d'emmental ; chaque tranche est un type de test de sûreté dont le niveau peut alors être calculé comme:

- le rapport entre les tranches à appliquer et le niveau de risque acceptable sur une version donnée ;
- le résultat des tests de ces différents contrôles identifiés le plus en amont possible comme une valeur qui sera pondérée par le poids du type de test. **[10]**

8.10 Tests de sécurité

La sécurité devient un enjeu important pour les entreprises, il faut donc intégrer la sécurité comme une exigence non fonctionnelle dans les spécifications, définir les niveaux de sécurité et prévoir les

tests de sécurité associés. En conclusion, ces différents types de tests de logiciel sont déterminants pour réussir des projets IT à succès. Il ne faut pas les concevoir et les réaliser à la fin du développement du logiciel, mais au fur et à mesure de la conception du logiciel. C'est la manière la plus avisée d'alléger votre dette technique future, de gagner du temps précieux et de l'énergie. L'organisation de ces tests ce fait dans un document appelé "stratégie de test". **[9]**

8.11 Test de la boite noire

Le test de la boîte noire, ou test de la boîte opaque, est utilisé en programmation informatique et en génie logiciel pour tester un programme en vérifiant que les sorties obtenues sont bien celles prévues pour des entrées données.

L'expression « boîte noire », ou « boîte opaque », vient du fait que les composants et les processus du dispositif de traitement ne sont pas visibles ou transparents et que le programme testé n'est pas étudié.

Le test de la boîte noire fait référence à un processus de test logiciel qui oblige les testeurs à tester l'application sans connaître la structure interne de l'application. L'application testée peut être vue comme une boîte noire dans laquelle vous ne pouvez pas la voir. **[11]**

8.12 Test de la boite blanche

Le test en boîte blanche (white box testing, en anglais) est une méthode de test logiciel qui utilise le code source d'un programme comme base pour concevoir des tests et des scénarios de test pour l'assurance qualité (QA).

Un test en boîte blanche consiste généralement à inspecter les chemins d'exécution possibles par l'intermédiaire du code pour trouver les valeurs d'entrée qui forceraient l'exécution de ces chemins. Le testeur, en général le développeur, vérifie le code d'après sa conception. Il est donc important que la personne qui exécute le test soit familière du code. **[12]**

9. Conclusion

Malgré les problèmes formels rencontrés par les testeurs, les tests sont désormais un aspect important dans l'amélioration de la qualité du logiciel. Il n'existe aujourd'hui aucune méthode capable de démontrer l'entière exactitude d'un programme. Il est en fait une vocation en soi. C'est la seule activité du cycle de développement où vous pouvez voir toutes les capacités d'un produit logiciel, selon l'industrie. C'est l'étape la plus chère puisqu'elle représente 30 à 40% des dépenses de développement logiciel, selon la criticité du projet. **[2]**

Chapitre 02

Test basé sur les modèles

Chapitre 02 : Test basé sur les modèles

1. Introduction

Depuis le début du développement industriel, l'homme a toujours tenté d'automatiser le plus de tâches possibles. Il a également appliqué ce principe au développement informatique, en cherchant à automatiser les phases de développement et les phases de tests logiciel.

Les tests logiciels permettent de vérifier le bon fonctionnement d'une application. A l'origine, ces tests étaient exécutés manuellement. Le début de l'automatisation des processus de conception logicielle coïncide avec l'augmentation de la qualité demandée par les clients. Pour réaliser cela, les tests logiciels informatiques sont apparus. Il s'agit de codes informatiques testant un autre code. Il faut savoir que les tests représentent environ 50% du coût d'un projet, sachant qu'il y a donc un intérêt croissant pour les techniques de réduction des coûts, il ya donc un fort engouement pour automatiser les tâches. En effet, suivant des études récentes, l'exécution des tests manuels génère un coût linéaire; tandis que les tests automatiques sont plus coûteux au départ mais le coût de maintenance augmente très faiblement.

Les tests sont de plus en plus automatisés; qu'ils soient unitaires, c'est à dire qu'ils testent les fonctions d'un système; fonctionnels, qui testent des fonctionnalités d'un système; ou encore d'intégration, qui vont vérifier la fusion de différentes parties d'un logiciel. Les frameworks de tests utilisés aujourd'hui sont du type JUnit ou TestNG pour valider du Java, Quick Test Pro pour des interfaces graphiques, et bien d'autres encore (série des xUnit).

Une technique de développement appelée modélisation est apparue au début des années 70. Il s'agissait de représenter le système par un modèle. A partir de là, est apparu le MDE, Model Driven Engeneering au début des années 80. Le Model Driven Engeneering est une modélisation d'une application avec un haut niveau d'abstraction, qui est basé sur un paradigme de modélisation qui considère le MDE si le modèle a du sens d'un point de vue utilisateur et qu'il peut servir de base à l'implémentation du système.

Puis, le MDD, Model Driven Development, une extension du MDE, est apparue au début du 21ème siècle. Le MDD est une approche des logiciels de conception destinée au développement d'applications. En fait, MDD est une conception logiciel à partir de son modèle indépendamment de la plate-forme utilisée.

Et le MBT dans tout ça ? Il s'agit de la partie modélisation côté validation.

Depuis la diffusion massive des langages objets, plusieurs éditeurs ont tenté de modéliser les applications qu'ils allaient développer pour gagner du temps et par conséquent de l'argent. Dans la plupart des entreprises de développement logiciel, plusieurs équipes travaillent en collaboration, qu'elles soient orientées développement, intégration, architecte ou autre. Mais les équipes de validation et de développement travaillent sur les revues de spécification transmises par le client ou un service interne, généralement marketing. Les revues de spécification traduisent littéralement une architecture et des fonctionnalités qui seraient souhaitables dans l'application.

L'objectif des équipes de développement et de validation est de fournir une application la plus fiable possible. **[13]**

2. Model-based Testing

2.1 Définition

Le model-based testing (MBT) est une variante des techniques de test qui se base sur des modèles de comportement explicites, décrivant les comportements attendus du système sous test (SUT), ou le comportement de son environnement, construits à partir des exigences fonctionnelles. Le MBT est une approche en évolution qui vise à générer automatiquement à partir de l'un de ces modèles, des cas de test à jouer sur le SUT. En outre, le MBT se positionne dans le cycle en V de développement, entre la phase de spécification et la phase de validation (voir figure suivante). Cependant, l'idée derrière la modélisation explicite du comportement attendu du SUT et éventuellement les comportements de son environnement est de contribuer à atténuer les problèmes de gestion des tests, c'est-à-dire avoir des cas de test bien documentés, facile à reproduire et à maintenir. Traditionnellement la gestion manuelle des cas de test n'est pas évidente. **[14]**

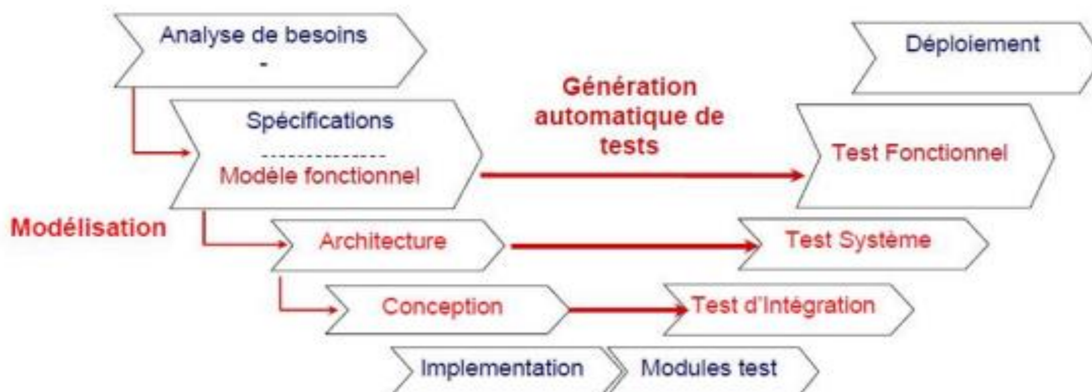

Figure 2.1 : Cycle en V

Le concept du Model-based Testing (MBT) remonte aux années 70. MBT est né du fait que l'on cherche à surélever les tests basés sur des spécifications. L'idée était que, quand une notion devient trop complexe, afin de mieux comprendre, il serait intéressant de la découper en des structures plus petites. Des représentations graphiques peuvent être utilisées pour créer des structures plus petites et pour l'élaboration d'un modèle abstrait pour lequel des tests concrets pourront être utilisés sur le système cible. A la suite de cette génération de tests, un comparatif montrera les résultats obtenus et les résultats attendus. Cette technique de tests consiste à se focaliser sur l'élaboration d'un modèle de tests qui servira pour l'exécution des tests suivant une chronologie déterminée. **[15]**

2.2 Avantages économiques du MBT

L'intérêt d'adopter le model-based testing dans les projets est d'améliorer la détection des bugs du SUT, réduire le coût et le temps de la phase de tests, améliorer la qualité logicielle, la traçabilité et l'évolution des exigences.

Le model-based testing peut amener à réduire le temps et les efforts consacrés à tester si le temps nécessaire pour écrire et maintenir le modèle ainsi que le temps consacré à diriger la génération de test est inférieure au temps de la conception et le maintien d'une suite de test manuellement. Dans ce cas le processus du model-based testing est rentable, car la génération des scripts de test est automatisée, et il rend plus facile la gestion de l'évolution des exigences, en ne modifiant que le modèle et en régénérant les cas de test, plutôt qu'en maintenant la suite de test en-soi. Cela permet de réduire considérablement le coût de la maintenance de test.

Avant d'adopter le MBT, il est souhaitable d'avoir un processus de test assez mature et une certaine expérience avec l'exécution automatique de test. Le MBT a été utilisé avec succès sur de nombreux projets industriels et la plupart des études ont montré qu'il était un moyen efficace de détection des bugs du système sous test, ainsi qu'une technique rentable. D'autre part, utiliser le model-based testing pour tester des lignes de produits peut être aussi efficace dans ce contexte, pour but de bénéficier de la maturité de cette approche et de son succès auprès des industriels. **[14]**

2.3 Processus MBT

Le MBT adopte un processus assez générique illustré par la figure suivante : **[14]**

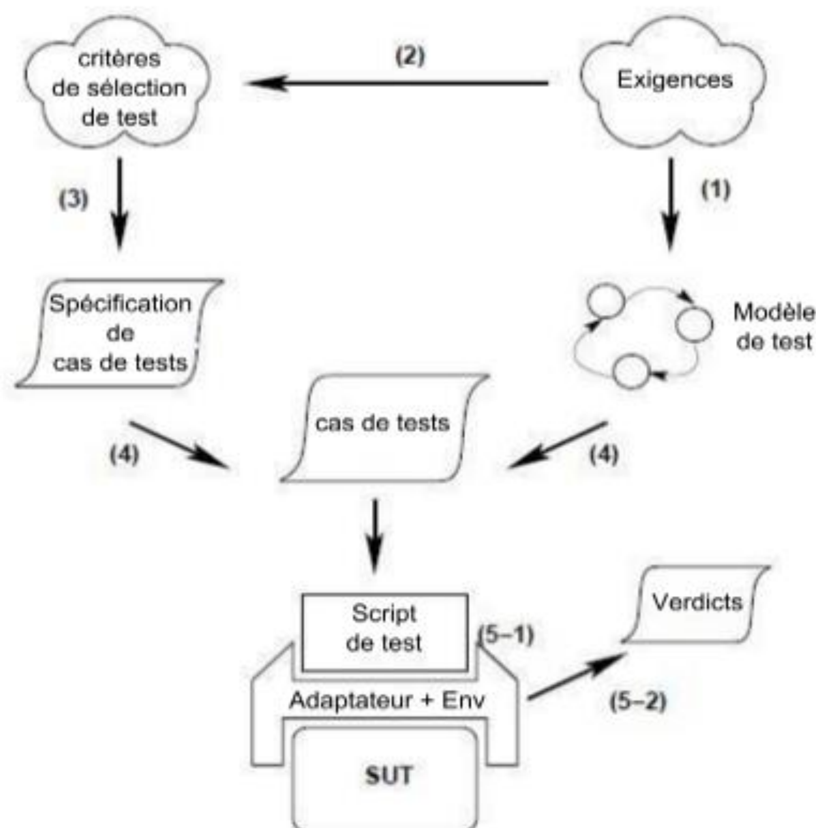

Figure 2.2 : Processus de model-based testing.

La première étape (1) du MBT consiste à construire un modèle abstrait appelé modèle de test à partir des exigences fonctionnelles ou documents de spécification du système logiciel. La validation du modèle signifie vérifier les exigences du système testé pour la cohérence, l'exhaustivité et souvent exposer les erreurs d'exigences. D'autre part, les modèles doivent être suffisamment précis pour être utilisés pour générer automatiquement des cas de test valides. Cela signifie que la génération de tests doit être complète en termes de fonctionnement, de données d'entrée et de résultats attendus pour avoir une réelle valeur ajoutée.

La deuxième étape (2) du processus global consiste à sélectionner les critères de sélection des cas de test pour la génération automatique. Les méthodes de test largement utilisées dans l'industrie, telles que les directives ISTQB, fournissent un processus de vérification générique et personnalisable visant à définir une stratégie de test claire et objective pour la vérification des SUT. Typiquement, ces approches formalisent cette stratégie dans un document de plan de test qui définit le périmètre et les différentes techniques de test à utiliser lors de la phase de vérification du projet de développement (tests d'intégration, tests unitaires, tests système, etc.).

Les critères de sélection de tests peuvent concerner une fonctionnalité spécifique appelée *requirement-based selection critera*, ou la structure du modèle de test, par exemple des critères liés à la couverture des états, la couverture des transitions ou des techniques de couverture de données (pairwise, valeurs limites). Les critères de sélection peuvent aussi être de caractère aléatoire, comme les propriétés d'environnement. Dans certains cas, ils peuvent être reliés à un ensemble bien défini de défauts.

La troisième étape (3) transforme les critères de sélection en des spécifications de cas de test qui se chargent de les formaliser pour les rendre opérationnels. Une spécification de cas de test est une description de haut niveau d'un cas de test souhaité.

La quatrième étape (4) concerne de la génération des cas de test, après la construction du modèle de test et la définition des spécifications des cas de test. L'algorithme de génération automatique peut être confronté à deux situations lors de la dérivation des cas de test. Première situation, les critères de sélection ne sont pas satisfaits, il n'y a donc pas de génération possible. La deuxième situation, plusieurs cas de test sont générés qui peuvent satisfaire les critères de sélection. L'algorithme de génération sélectionne un seul cas de test parmi la sélection opérée pour chaque critère.

La cinquième étape (5) identifie deux moyens d'exécuter une suite de test. L'exécution des tests peut être manuelle à l'aide d'une personne physique, ou automatique via un environnement dédié (banc de test), qui fournit des facilités pour exécuter automatiquement les cas de test et pour enregistrer les verdicts. Cependant, l'exécution débute par la concrétisation des entrées de tests et par l'envoi de ces données concrètes au système sous test afin de le stimuler et capturer par la suite les résultats attendus, pour les comparer avec les résultats attendus (verdicts de test).

Un cas de test peut être réalisé aussi sous forme du code exécutable appelé script de test, capable d'exécuter le cas de test avec ces entrées sur un banc de test et de calculer le verdict.

En résumé, le processus du model-based testing se base sur la réalisation d'un modèle de test (1) la définition d'une stratégie de test (2,3) pour générer des cas de test (4) qui peuvent être complétés par les vérifications à réaliser (si celles-ci ne sont pas incluses dans le modèle), accompagnées par une traduction éventuelle de cas de test pour une exécution automatique (5-1) et achevées par une exécution et un verdict sur les résultats issus des tests.

2.4 Aspects modélisation

Plusieurs formalismes ont été utilisés pour décrire un modèle, les machines à états, les systèmes de transitions étiquetées, le diagramme d'activité UML [OG09], les réseaux de Pétri, les chaînes de Markov (pour les modèles probabilistes), etc.

En outre, chaque modèle offre des possibilités différentes concernant (par exemple) la prise en compte et la résolution du non-déterminisme (conditions, probabilités, etc.), la manipulation des entrées, des sorties et des variables. Tous ces modèles sont complémentaires et offrent toutes les possibilités requises pour la modélisation et la génération de cas de test automatiques. Pour la correction du modèle, plusieurs solutions sont possibles, comme la simulation, l'exploration, l'animation, le modèle checking, analyse statique et les preuves.

Nous nous intéressons dans ce travail aux types de modèles assimilés à des états -transitions. Cette technique modélise les états du SUT, les transitions entre ces états, les actions à l'origine des transitions et les actions pouvant résulter d'une transition. Pendant la conception des cas de test, les exigences fonctionnelles fournissent l'information état - transitions, parfois cette information peut être aussi tirée des artefacts de conception qui prennent en charge la notation état-transition, comme les diagrammes d'activité dans UML, le modèle d'usage.

Les cas de test sont conçus pour opérer les transitions entre les états et spécifier :

– L'état initial du système sous test ;

– Les entrées du SUT ;

– Les résultats attendus du SUT ;

– L'état final du SUT. **[14]**

2.5 Outils de Model-based Testing

Un certain nombre d'outils de model-based testing existent. Nous identifions trois différentes catégories : outils commerciaux, propriétaires et académiques. Nous avons choisi de présenter MaTeLo, JSXM, Qtronic et Microsoft SpecExplorer. Ils ont été choisis en fonction de leur maturité. **[14]**

2.5.1 MaTeLo

MaTeLo est l'acronyme de *Markov Test Logic* développé par ALL4TEC. C'est un outil commercial de génération automatique de tests fonctionnels et de validation, basé sur l'approche de l'ingénierie dirigée par les modèles. La construction des modèles de test, la génération automatique des cas de test et des données de test ainsi que l'analyse de la campagne de test sont les trois fonctionnalités majeures proposées par l'outil.

MaTeLo a son propre modeleur basé sur les chaînes de Markov comme paradigme de modélisation pour représenter le comportement attendu du système sous test. La sélection des cas de test est

orientée par les probabilités associées aux transitions du modèle de test. MaTeLo génère des cas de test avec des données d'entrée et de sortie. Les cas de test générés peuvent être exportés en script sous plusieurs formats selon le banc de test connecté.

2.5.2 JSXM

JSXM est un outil académique, développé par l'université de *Sheffield* sous Java, qui permet la spécification de modèles JSXM, leur animation et la génération automatique des cas de test.

Les modèles JSXM sont un type particulier de machines finies d'états étendues, appelé Flux X-Machines (SXMS). SXMS permet de décrire à la fois le contrôle et les données d'un système. Le langage de modélisation JSXM est un langage basé sur XML avec Java Code en ligne. L'entrée et les symboles de sortie sont également décrits dans un code XML.

L'animation du modèle consiste à exécuter le modèle en fournissant un flux d'entrée et en observant le flot de sortie résultant. L'animation interactive ou par lots, sont les deux types d'animation pris en charge par l'outil, elles permettent au concepteur de valider la spécification, c'est-à-dire vérifier que la fonctionnalité sous test est correctement modélisée. Une fois que le modèle est validé, il peut être utilisé pour l'algorithme de génération automatique des cas de test. Les cas de test qui sont générés par JSXM sont disponibles en format XML et ils sont indépendants de la technologie ou le langage de programmation de l'application. Ces cas de test général peuvent ensuite être transformés par l'outil JSXM à des cas de test aux langages de programmation du système sous test.

2.5.3 Qtronic

Qtronic est un outil commercial de génération de test basé sur un modèle développé par Conformiq. Le modèle de conception peut être exprimé comme un ensemble de fichiers textes et/ ou des modèles graphiques. La notation textuelle est définie par la Qtronic Modeling Language (QML). QML signifie un diagramme *Statecharts* étendu avec du code Java ou C#. Le langage d'action d'un diagramme d'états UML tels que des événements, des actions, des conditions de garde et d'autres contraintes spécifiques sur la transition sont décrites en utilisant QML.

L'outil utilise essentiellement le concept de l'exécution symbolique du modèle pour la génération de cas de test. L'outil peut traduire les cas de test générés à n'importe quel format exécutable. Qtronic prend en charge les modèles *multi-thread* concurrents, et support également le test des systèmes non-déterministes en mode en ligne.

2.5.4 SpecExplorer

Microsoft SpecExplorer [VCG+08] est un outil de test basé sur un modèle développé à l'origine chez *Microsoft Research* et maintenant intégré dans la plate-forme de développement *Visual Studio*. Il étend l'environnement de programmation pour modéliser le comportement des logiciels, analyser et visualiser graphiquement le comportement modélisé et la génération automatique des suites de test.

SpecExplorer prend en entrée un jeu d'ensembles de modèles .NET créés à partir du modèle de programme et un ensemble de scripts de coordination, pour configurer l'exploration du modèle et de test ainsi que la composition des scénarios. Les scénarios sont utilisés pour limiter le comportement général du programme du modèle à une tranche qui est testable. SpecExplorer propose plusieurs stratégies pour gérer l'exploration du modèle, y compris la couverture des valeurs de paramètres des données et l'espace d'états du modèle, ainsi que les critères structurels du modèle tels que la couverture de toutes les transitions. Les cas de test sont générés comme une collection de fichiers C# qui peuvent être consommés par n'importe quel framework .NET de test.

2.6 Différents types de modèles pour le Model-Based Testing

On va aborderer plus en détail les différents types de modèles applicables pour cette méthode : Machine à états finis, UML/OCL.

2.6.1 Machine à états finis

Pour mieux comprendre ce que sont les machines à états, il est utile de mettre en évidence le côté pratique de leur définition ainsi que leur explication. Tous les développeurs voudraient augmenter la maintenabilité du code ainsi que sa fiabilité tout en réduisant les délais de mise en œuvre. La machine à états serait une bonne solution. Si elle est bien connue, elle n'est guère utilisée surtout par les développeurs autodidactes. Et pourtant, s'il y a bien des outils utiles et simples à disposition ce sont bien les machines à états. Elles donnent lieu à la possibilité de diviser les algorithmes longs et complexes en des éléments plus petits et donc des éléments plus faciles à gérer. Quand un algorithme peut être représenté par un organigramme ou un digramme d'états il peut être implémenté grâce à l'architecture de programmation des machines à états. Quand les machines à états sont petites on les représentes par des diagrammes à bulles, ou des diagrammes d'états/transitions, qui modélisent le système du point de vue dynamique. **[15]**

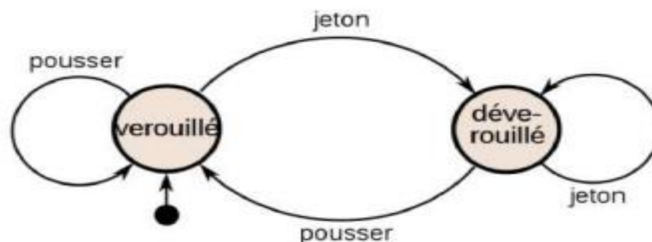

Figure 2.3 : Graphe orienté d'un portillon d'accès

Les bulles représentent les états de repos, des "états simples", illustrant les différents états dans le système et les lignes qui les relient sont les lignes de transitions qui modélisent les actions. Ces actions, transactions, peuvent être soit externes soit internes. Externe quand l'action se situe entre deux états et interne quand l'action reste dans un même état. Quand un changement d'état du système est la réponse à un évènement il s'agit d'une transition. Celle-ci peut être définie par trois éléments : évènement, garde et action. L'évènement, appelé aussi "trigger", l'évènement est annoncé en tant qu'une opération de la machine d'état. La garde, appelée "guard", est sollicitée pour activer la transition au moyen d'une expression booléenne qui doit être satisfaite. Comme il n'est pas possible à une machine à états d'avoir plusieurs états en même temps, elle doit suivre les conditions signalées et les transitions précisées au préalable pour transiter d'un état à l'autre. Quand la garde est satisfaite, une action peut être exécutée. À partir de l'objet utilisé l'action peut, sur les objets visibles, agir de manière directe ou indirecte. Il est important de savoir que le code examine uniquement les conditions de sortie d'état, peu importe comment elle était arrivée à l'étape précédente, le code s'occupera seulement de l'état suivant.

Cette situation est comparable à une course d'orientation, un parcourt imposé et des balises à passer dans un ordre prédéfini (numérotées). Exemple : Si l'on a bien respecté le parcourt on sait que depuis la balise sept tout ce qu'on doit faire c'est d'aller à la balise huit et on ne s'occupe plus des balises trois, cinq ou deux, elles sont déjà faites. Les machines à états font de même, elles s'occupent de ce qu'elles doivent faire sans se soucier de comment elles sont arrivées à un état précis. Elles sont de bons outils pour tout problème dans lequel il y a des états de repos bien définis. De plus, leur conception accepte facilement des modifications d'un état sans avoir à craindre une quelconque régression, il n'est pas difficile d'ajouter de nouvelles fonctionnalités.

Pour aborder un modèle de Model-Based Testing, on doit garder à l'esprit qu'il doit être le plus petit possible et donc on doit savoir quoi éliminer afin que la vue que l'on conserve soit assez abstraite pour que la machine d'états finis puisse la traiter.

Il existe deux sortes de machines d'états finis : celle déterministe à états finis et celle non déterministe à états finis. La première, déterministe, n'accepte qu'une entrée pour deux options possibles : "Si" alors "faire ceci" sinon "faire cela". La seconde, non déterministe, pour une entrée bien plus d'options à exécuter sont possibles.

Pour illustrer ces propos, voici un exemple simplifié de deux graphes : l'un montrant une machine à café déterministe proposant seulement du café, l'autre montrant une machine à café non déterministe proposant café ou thé.

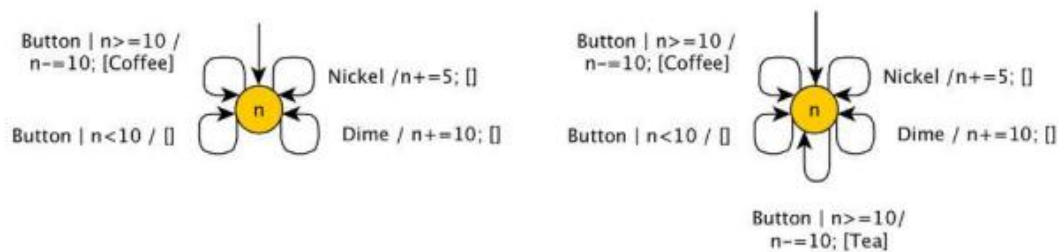

Figure 2.4 : Machine à café déterministe (à gauche) et non déterministe (à droite)

Dans la pratique, les tests de conformité des machines à états finis sont très utiles et ce malgré la simplicité de la notation "Finite State Machine" (FSM). Les circuits numériques, les systèmes de contrôle intégrés et d'autres types de systèmes, comprenant les protocoles, ont été directement définis par les FSM. Pour définir des systèmes ou des parties de systèmes, il existe différentes notations formelles très semblables aux machines à états finis et parmi elles on peut citer les diagrammes d'états UML.

2.6.2 UML/OCL

UML n'est pas une méthode mais un langage de modélisation qui n'a cessé d'évoluer depuis les années '80. Ce langage est composé essentiellement de graphiques, diagrammes et pictogrammes. La version UML 2.5 offre le choix parmi quatorze diagrammes. Les différents diagrammes correspondent aux différents points de vue d'un modèle. On laisse à chacun le choix des diagrammes à utiliser. On peut comparer ces différents points de vue aux différents plans d'une construction de maison. Les corps de métier intervenants ont chacun leur plan pour un même bâtiment, le plombier aura un plan différent du maçon qui sera lui aussi différent de celui de l'électricien.

Le langage UML permet surtout d'exprimer des contraintes structurelles.

Pour mieux expliquer UML, on peut ajouter qu'il est composé de trois parties :

- Les vues, qui correspondent aux différents points de vue qui peuvent décrire le système.

  Ces points de vue sont très variés et adoptés en fonction de leur utilité. On peut citer : les points de vue logiques, architectural, dynamique, temporel, géographique, organisationnel, etc ;

- Les diagrammes, qui sont les descriptions graphiques de la composition des vues ;
- Les modèles éléments, qui sont les composants graphiques formant les diagrammes.

Les vues, elles peuvent se superposer pour améliorer la compréhension du système.

La vue de cas d'utilisation, "use-case view", décrit les attentes de chaque acteur. Répond aux questions "quoi ?" ou "qui ?".

La vue logique c'est la description vue de l'intérieur expliquant comment satisfaire les attentes des acteurs. Répond à la question "comment ?".

La vue d'implémentation détermine les différentes dépendances entres les différents constituants.

La vue de processus est la vue technique et temporelle qui spécifie les tâches concurrentes, le contrôle, la synchronisation...

La vue de déploiement qui correspond à la question "où ?" pour situer chaque composant du système géographiquement et décrit l'architecture physique des composants.

Les diagrammes, ils se complètent tout en étant dépendants de façon hiérarchique. Depuis la version 2.3 on distingue quatorze diagrammes : sept statiques ou de structure, trois comportementaux et quatre dynamiques.

Les diagrammes de structure ou statiques : Diagrammes de classes, d'objets, de composants, de déploiement, des paquets, de structure composite, de profils.

Diagrammes de comportement : Diagrammes des cas d'utilisation, état-transitions, d'activité.

Diagrammes d'interaction ou diagrammes dynamiques : Diagrammes de séquence, de communication, global d'interaction, de temps.

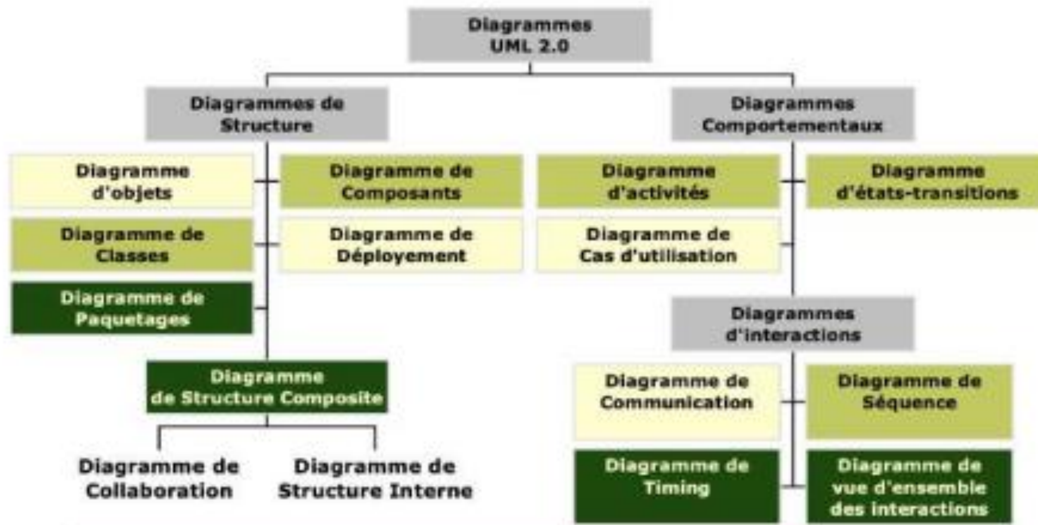

Figure 2.5 : Les différents diagrammes UML.

Modèles d'éléments

Un stéréotype, à définir entre guillemets.

Un classeur, *un rectangle* regroupant des éléments ayant la même structure, comportement.

Un paquet qui regroupe des unités ou des diagrammes.

Chaque objet ou classe est défini par " : :". Exemple : Une classe X hors de son classeur ou de son paquet sera exprimé par "Paquet A : : Classeur B : : Classe X".

Ce langage présente quand même quelques insuffisances qui sont comblées par le langage OCL

(Object Contraint Language). Le langage est utilisé quand les contraintes exprimées s'avèrent être trop complexes pour bien les exploiter en UML et n'y sont pas déjà prédéfinis. OCL est un sous ensemble d'UML, davantage orienté vers l'aspect dynamique du système. Pour le Model-Based Testing il existe des langages adaptés, qui découlent d'UML et OCL, il s'agit respectivement d'UML4MBT et OCL4MBT. **[15]**

3. Test basé sur l'état

3.1 Définition

Le test de transition d'état permet de tester la capacité d'un système à entrer dans des états définis et à en sortir par des transitions valides et invalides. Des événements particuliers font passer le système d'un état à un autre et déclenchent certaines actions. **[16]**

3.2 Quand utiliser les tests de transition d'état?

Les tests de transition d'état peuvent être utilisés dans les situations suivantes:

- Lorsque l'application testée est un système en temps réel avec différents états et transitions englobés ;
- Lorsque l'application dépend de l'événement / des valeurs / des conditions du passé ;
- Quand la séquence des événements doit être testée ;
- Lorsque l'application doit être testée par rapport à un ensemble fini de valeurs d'entrée. **[17]**

3.3 Techniques

3.3.1 Application

Cette technique peut s'appliquer à tous les niveaux de test. Elle est particulièrement conseillée dans les situations suivantes:

- logiciels embarqués ;
- logiciels web (chaque page peut être vu comme un état) ;
- logiciels transactionnels ;
- logiciels de contrôle (feux de signalisation par exemple) ;
- globalement tout logiciel pouvant être modélisé en diagramme de transition d'état. **[16]**

3.3.2 Limitation/difficultés/risque

Le plus difficile dans cette technique est d'identifier la liste des états et transitions à partir de la spécification. En oublier sera forcément source d'erreur dans la génération du graphe et cela se traduira dans des oublis de cas de test pertinents pour tester le logiciel.

L'autre difficulté est de définir le niveau de couverture qu'il faut atteindre pour satisfaire le risque. En effet, plusieurs niveaux de couverture peuvent être définis et chaque niveau permettra de découvrir des défauts différents. **[16]**

3.3.3 Couverture

L'ISTQB définit que le degré minimum acceptable est d'avoir traversé au moins une fois tous les états et toutes les transitions.

Plus globalement, nous parlerons de couverture d'aiguillage-N (N-switch en anglais). Par exemple, une couverture d'aiguillage-0 signifie que toute séquence valide d'une seule transition a été testée au moins une fois. Pour une couverture d'aiguillage-1 signifie que toute séquence valide deux transitions successives a été testée au moins une fois. En conclusion, une couverture d'aiguillage-N signifie que toute séquence valide de N+1 transitions successives a été testée au moins une fois.

L'ISTQB défini aussi la « couverture Aller-Retour » qui correspond aux situations dans lesquelles les séquences de transitions forment des boucles. 100% de « couverture Aller-Retour » est obtenu lorsque toutes les boucles partant d'un état et revenant vers ce même état ont été testées. Cela doit être testé pour tous les états inclus dans des boucles.

Enfin, la couverture peut aussi être enrichie avec les transitions invalides. Une transition invalide est une transition s'exerçant sur un état alors que cela ne devrait pas arriver. Sur un diagramme états-transitions, elle n'apparaît pas. **[16]**

3.3.4 Type de défaut

Les principaux défauts sont:

- Omission de la prise en compte d'une transition dans un état donné ;
- Prise en compte d'une transition invalide ;
- Mauvais traitement d'une transition selon son événement et mauvaise action déclenchée. ;
- Mauvais traitement dans un état courant à cause de traitements dans les états précédents. ;
- Contradiction entre les transitions et états.

Dériver le diagramme états-transition et la table de transitions d'états depuis la spécification permet de critiquer cette dernière et s'assurer que la machine à état à développer sera la bonne. **[16]**

3.3.5 Mise en œuvre

Un distributeur de boissons est un très bon exemple pour une telle technique.

Prenons le distributeur suivant (volontairement simple):

- le distributeur permet au client de choisir parmi plusieurs boissons en tapant un chiffre ;
- le distributeur accepte uniquement des pièces de monnaies ;
- le distributeur prend en compte les pièces de monnaies uniquement avant et après la sélection d'une boisson par l'utilisateur ;
- lorsque le montant de la boisson est atteint ou dépassé, le distributeur procède à la préparation de la boisson puis rend la monnaie supplémentaire.

La technique ne s'intéresse qu'aux états et transitions. Je vous ai mis quelques actions afin de rendre le diagramme plus réaliste et montré que ce dernier peut être complexe.

Quand Nous allons chercher à obtenir une couverture d'aiguillage-0. Rappelez-vous que cela signifie que toute séquence valide d'une seule transition a été testée au moins une fois.

Pour cela, deux possibilités:

soit à partir du graphe en créant des cas de test permettant de couvrir chaque transition partant de chaque état en prenant soin de considérer chaque condition d'une transition conditionnelle de manière unique.

Soit en faisant la table de transition d'état et créer des cas de test pour couvrir chaque cases non vides. **[16]**

3.4 Diagramme d'état

Un diagramme états-transitions est un schéma utilisé en génie logiciel pour représenter des automates déterministes. Il fait partie du modèle UML et s'inspire principalement du formalisme de statecharts et rappelle les grafcets des automates. S'ils ne permettent pas de comprendre globalement le fonctionnement du système, ils sont directement transposables en algorithme. En effet, contrairement au diagramme d'activité qui aborde le système d'un point de vue global, le diagramme états-transitions cible un objet unique du système. Tous les automates d'un système s'exécutent parallèlement et peuvent donc changer d'état de façon indépendante.

4. Conclusion

Le Model Based Testing est une approche assez lourde de génération de tests mais très intéressante. Il permet de générer des tests non s'en avoir une certaine connaissance technique de la modélisation. Par exemple, un modèle trop grand va entrainer un développement plus important et une génération de tests encore plus longue, et aussi plus difficile à maintenir. Une erreur de conception dans le modèle va créer une série de tests erronés. En contre partie, l'efficacité à générer des tests n'est pas négligeable. Le MBT est surtout utilisé pour la non-régression, ce qui entraine beaucoup de question sur la sauvegarde des modèles et des tests avec un numéro de version.

Pour générer ces tests, le travail collaboratif entre des équipes de marketing et de validation est une solution qui permet de rendre l'ensemble du travail efficace pour éviter de modifier le modèle constamment.

Malgré les avantages qu'on peut trouver à cette méthode, les outils sont en manque de maturité pour le moment. Ils n'offrent pas une facilité de mise en œuvre pour gagner du temps de conception.

Le travail de recherche ou de conception du harnais (où il est nécessaire d'avoir une bonne collaboration entre développement et validation) et la conception d'un modèle est un travail conséquent. Lorsque le squelette de test est généré, il faut encore développer le contenu des méthodes des tests (la couche fonctionnelle). Il y a donc des difficultés d'évaluation du ROI.

On remarque donc des avantages et des inconvénients au MBT, mais je pense que c'est un concept qu'il va falloir suivre dans les années qui viennent. **[13]**

Chapitre 03

Approche proposée

Chapitre 03 : Approche proposée

1. Introduction

Parmi les techniques de test basée sur les modèles, on trouve la technique basée sur les machines d'états finis, également appelée State-Based Testing (SBT) **[19]**. L'un des principaux avantages de cette technique est qu'elle permet aux tests d'être directement liés aux exigences du logiciel sous test. Cela permet de contribuer à la lisibilité, la compréhensibilité et la maintenabilité des tests. De plus, il a été démontré qu'elle fournit de bons critères de couverture pour tous les comportements du logiciel sous test.

2. Couvertures du test utilisé dans le SBT

De nombreux critères de couverture ont été proposés comme moyen d'évaluer une suite de tests basée sur les machines d'état finis. Les critères les plus étudiés sont la couverture de tous états (AS), la couverture de toutes transitions (AT), All Transition-Pairs (ATP) et Round-Trip Path (RTP), qui mesurent le degré d'exécution des états, des transitions, des paires de transitions et des chemins aller-retour. De nombreuses études empiriques ont montrés que la couverture de test ATP est plus efficace. Cependant, son coût est plusieurs fois supérieur. D'autres ont montrés que les couverture AS et AT ne fournissent pas un niveau adéquat de détection de fautes. Plus récemment, des études ont montré que la couverture RTP est raisonnablement efficace pour détecter les défauts. Son coût est également beaucoup moins cher que celui d'une suite ATP.

2.1 Stratégie RTP

La stratégie RTP qui génère une suite RTP a été suggérée par Binder **[20]**. Dans cette technique on parcourt une machine d'état pour construire un arbre de transition qui inclut toutes les transitions dans le graphe. Le parcours s'arrête chaque fois que l'état visité a été déjà rencontré dans cette traversée. Selon Binder, une suite RTP peut détecter les défauts de contrôle, tels que transitions incorrectes ou manquantes, à un coût relativement faible. La figure 3.1 montre une machine d'état finis et le graphe correspondant générer selon la technique RTP.

Cependant, certaines études ont rapporté qu'une suite RTP est peu susceptible d'être suffisante dans la plupart des situations. Ils ont insisté sur le fait qu'elle ne couvre pas certaine utilisation réaliste du système sous test, ce qui pourrait entraîner une mauvaise détection des défauts. En effet, Mouchawrab et al. **[21]** ont montrés que certains chemins, même lorsqu'ils représentent des scénarios d'utilisation courants, ne sont pas nécessairement testés par cette technique. Ainsi, Briand et al. **[22]** ont suggéré une alternative qui exécute des scénarios plus réalistes pour exercer correctement le code.

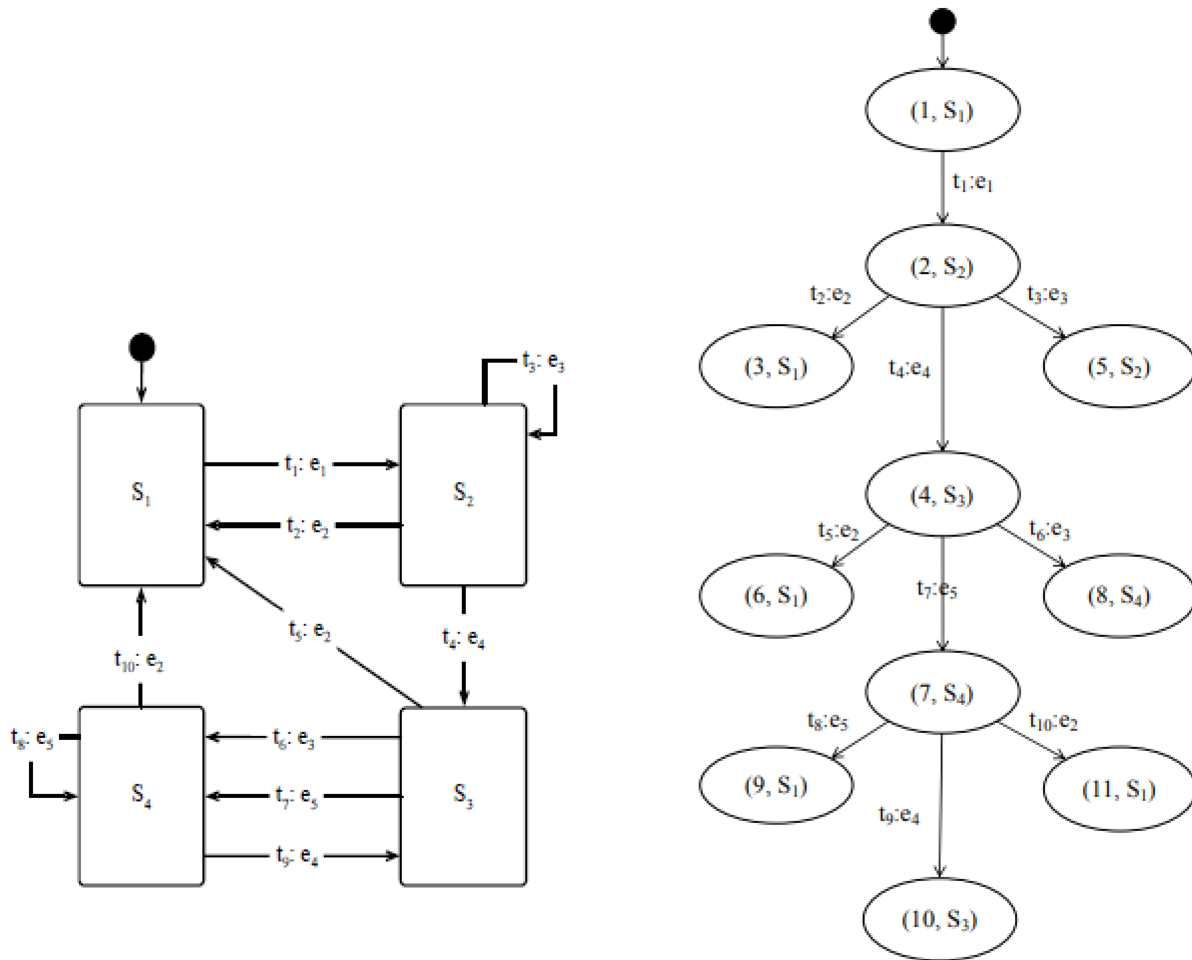

Figure 3.1 : arbre RTP construit à partir d'une machine d'état finis.

En figue 3.1 par exemple, <t1, t2>, <t1, t4, t5> et <t1, t4, t7, t8> sont les chemins aller-retour qui commencent et se terminent par S1. Étant donné que les traversées se terminent en S1, les paires d'événements avec e2 en tant que premier événement ne peut pas être couvertes. De plus, cette méthode empêche certaines paires d'événements d'être couvertes. Dans la figure 1, cette méthode génère <t1, t4, t6>, ce qui empêche les paires d'événements avec e3 en tant que premier événement d'être couvertes.

Ceci peut être un facteur qui empêche de tester des scénarios d'utilisation critiques. Par conséquent, nous avons essayé de compléter la suite RTP pour couvrir toutes les paires d'événements. De cette manière, des scénarios d'utilisation plus diversifiés pourraient être testés.

2.2 Stratégie MTT

La stratégie d'arbre de transition modifié (MTT) construit un arbre de transition modifié qui inclut des chemins aller-retour complets **[22]**, **[23].**

2.3 Stratégie OTT

La stratégie d'arbre de transition (OTT) construit un arbre de transition optimal qui combine différentes parties des alternatives à maximiser la couverture des flux de données **[22]**. Khalil et al. **[24]** ont proposé un nouvel algorithme de construction d'un arbre de transition. Le nouvel arbre génère une suite de tests qui exécute non seulement toutes les transitions, mais aussi des couples événement/état qui ne sont pas exécutés.

2.4 Stratégie RTP+SEP & RTP+FEP

La stratégie RTP + SEP augmente un arbre pour couvrir l'une des paires de transition que chaque paire peut déclencher. La stratégie RTP+FEP l'augmente pour les couvrir tous. Les stratégies génèrent une suite RTP+SEP et une suite RTP+FEP.

3. Approche proposée

3.1 Algorithme utilisé pour générer une suite de tests à partir d'un arbre de test

L'algorithme suivant présente ma manière utilisée pour générer une suite de tests à partir d'un arbre de test. La fonction test suite prend un arbre de test en entrée et renvoie un ensemble de cas de test. L'algorithme fonctionne de manière récursive en parcourant l'arbre et en créant une séquence de transitions pour chaque cas de test.

```
1      fonction suite de tests (tree) retour TS
2      tree: un arbre de test (V, v1, ARC)
3      TS: un ensemble de cas de test
4      debut
5      TS = Ø;
6      tc = <>;
7      cas de tests (v1, tc, TS);
8      retourner TS;
9      fin
```

```
1. fonction cas de tests (v, tc, TS)
2. v: un sommet
3. tc: une sequence de transitions
4. TS: un ensemble de cas de tests
5. debut
6.    tc = con*(tc, <tr(in(v))>);
7.    si OUT(v) = Ø alors TS = TS ∪{ tc };
8.    sinon
9.             pour chaque arc ∈ OUT(v)
10.             debut
11.             v = rst(arc);
12.              cas de tests (v, tc, TS);
13.              fin pour
14. fin si
15. fin
```

L’algorithme initialise d'abord un ensemble vide de cas de test (TS) et une séquence vide de transitions (tc). Il appelle ensuite la fonction cas de test avec le sommet racine de l'arbre (v1) et la séquence vide de transitions (tc). Le cas de test prend un sommet v, une séquence de transitions tc et l'ensemble actuel de cas de test TS en entrée. La fonction ajoute d'abord la transition entrante de v à la séquence de transitions tc (c'est-à-dire la transition qui a conduit à l'état actuel représenté par v). S'il n'y a pas d'arcs sortants de v, cela signifie que nous avons atteint la fin d'un cas de test. Dans ce cas, la séquence courante de transitions (tc) est ajoutée à l'ensemble des cas de test (TS). S'il y a des arcs sortants de v, la fonction s'appelle récursivement pour chaque arc et le sommet de destination correspondant. Le sommet de destination devient le nouveau v et la séquence de transitions est mise à jour avec la transition représentée par l'arc courant. La fonction continue de s'appeler récursivement jusqu'à ce que tous les chemins possibles depuis v aient été explorés. Dans l'ensemble, l'algorithme parcourt de manière exhaustive l'arbre de test pour générer tous les cas de test possibles pour un SUT.

3.2 Etapes pour augmenter un arbre RTP

La figure suivante résume les différentes étapes de l’approche proposée.

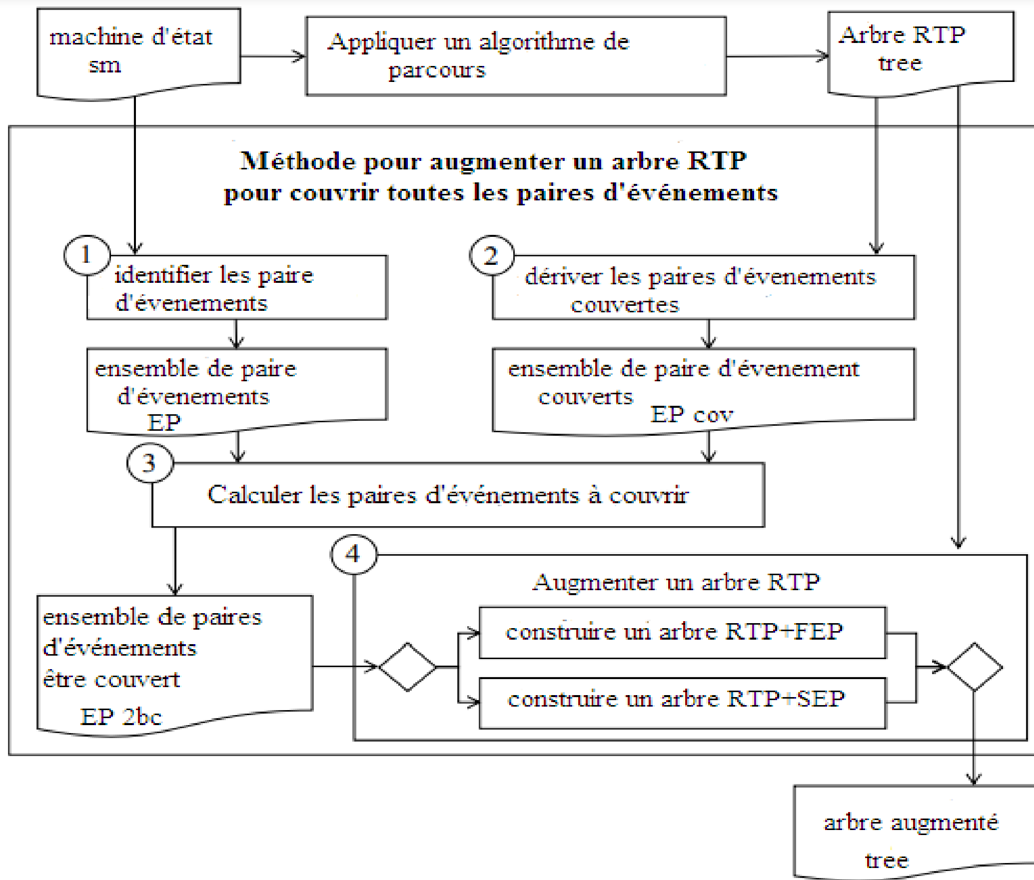

Figure 3.2 : Procédure d'augmentation d'un arbre RTP pour générer une suite augmentée.

3.2.1 Étape 1 : Identifiez les paires d'événements

Une paire de transition peut être déclenchée par deux événements acceptés successivement par une machine d'états finis. Par exemple, dans la figure 3.1, la paire de transitions (t1, t3) est déclenchée par deux événements, e1 et e3, que le diagramme accepte successivement. Nous avons défini une paire d'événements pouvant déclencher des paires de transition en tant que paires d'événements.

---

*(**Définition**. Paire d'événements) Étant donné une machine d'états finis sm, une paire d'événements (ei, ej) ∈ EP(sm) est une paire d'événements, où EP(sm) = {(ei, ej) / ∃(ti, tj) ∈ TP(sm), ei = événement(ti), et ej = événement(tj)}.*

Le tableau suivant représente les paires d'événements du notre exemple. Dans ce tableau, les cellules blanches représentent des paires d'événements du notre exemple. Par exemple, (e2, e1) dans la ligne 2 et la colonne 1 est une paire d'événements qui peut déclencher des événements (t2, t1), (t5, t1) ou (t10, t1). Les cellules notées ☒ représentent une paire d'événements qui ne peuvent pas déclencher de paires de transition.

	e_1	e_2	e_3	e_4	e_5
e_1	⊠	O	O	O	⊠
e_2	X	⊠	⊠	⊠	⊠
e_3	⊠	X	X	X	X
e_4	⊠	O	O	⊠	O
e_5	⊠	O	⊠	O	O

Tableau 3.1 : paires d'événements du notre exemple.

3.2.2 Étape 2 : Dériver les paires d'événements couverts

La suite de tests basée sur l'état couvre les paires de transition d'une machine d'état. Par exemple, un cas de test <t1, t4, t5> ∈ suite_de_tests couvre (t1, t4) et (t4, t5) du diagramme d'état. Une suite de tests qui couvre une paire de transitions couvre également la paire d'événements qui la déclenche. Par exemple, un cas de test <t1, t4, t5> couvre (e1, e4), qui déclenchent (t1, t4). Dans le tableau précédent, les cellules avec ◯représentent les paires d'événements couvertes par la suite de tests. Nous définissons dans ce suit une paire de transition couverte et une paire d'événement couverte.

---

*(**Définition**. Paire de transition couverte) Étant donné une machine d'états finis sm et un arbre RTP basé sur sm, une paire de transition (ti, tj) ∈ TPcov(sm, arbre) est une paire de transition couverte, où TPcov(sm , arbre) = {(ti, ti+1) ∈ TP(sm) | ∃<t1, ..., tn+1> ∈ suite de tests(arbre), 1 ≤ i ≤ n}.*

*(**Définition**. Paire d'événements couverte) Étant donné une machine d'états finis sm et un arbre RTP basée sur sm, une paire d'événements (ei, ej) ∈ EPcov(sm, arbre) est une paire d'événements couverte, où EPcov(sm , arbre) = {(ei, ej) | ∃(ti, tj) ∈ TPcov(sm, arbre), ei = événement(ti), et ej = événement(tj)}.*

3.2.3 Étape 3 : Calculer les paires d'événements à couvrir

Le reste EPcov(sm, tree) de EP(sm) représente un ensemble de paires d'événements que la suite de test ne couvre pas. Dans le tableau, les cellules avec ✕ les représentent. Nous donnons la définition d'un couple d'événements qu'une suite ne couvre pas comme un couple d'événements à couvrir.

---

*(**Définition**. Paire d'événements à couvrir ; ep2bc) Étant donné une machine d'états finis sm et un arbre RTP basé sur sm, une paire d'événements (ei, ej) ∈ EP2bc(sm, arbre) est une paire d'événements à couvrir (ep2bc), où EP2bc(sm, arbre) = EP(sm) – EPcov(sm, arbre).*

3.2.4 Étape 4 : Augmenter un arbre RTP

Un ep2bc peut être couvert en couvrant les paires de transition qu'il déclenche. La paire de transitions qu'un ep2bc peut déclencher est définie comme suit :

---

*(**Définition**. Paire de transition- qu'un ep2bc peut déclencher ; tp2bc) Étant donné une machine d'états finis sm, un arbre RTP basé sur sm et ep2bc = (ei, ej) ∈ EP2bc(sm, arbre), une paire de de*

transition(ti , tj) ∈ TP(sm, ep2bc) est une paire de transitions dans laquelle un ep2bc peut déclencher (tp2bc), où TP(sm, ep2bc) = {(ti, tj) ∈ TP(sm) | événement(ti) = ei et événement(tj) = ej }.

Étant donné une machine d'états finis et une suite de test, (e2, e1) est un ep2bc qui peut déclencher (t2, t1), (t5, t1) et (t10, t1). Autrement dit, TP_{cov}(machine, (e2, e1)) = {(t2, t1), (t5, t1), (t10, t1)}. Le tableau 2 montre chaque ep2bc de machine d'état et son TP correspondant (machine, ep2bc).

ep_{2bc}	$TP(sm_{ex}, ep_{2bc})$
(e_2, e_1)	$(t_2, t_1), (t_5, t_1), (t_{10}, t_1)$
(e_3, e_2)	$(t_3, t_2), (t_6, t_{10})$
(e_3, e_3)	(t_3, t_3)
(e_3, e_4)	$(t_3, t_4), (t_6, t_9)$
(e_3, e_5)	(t_6, t_8)

Tableau 3.2 : chaque ep2bc de machine d'état et son TP correspondant.

La figure 3 montre chaque arbre augmenté de notre exemple.

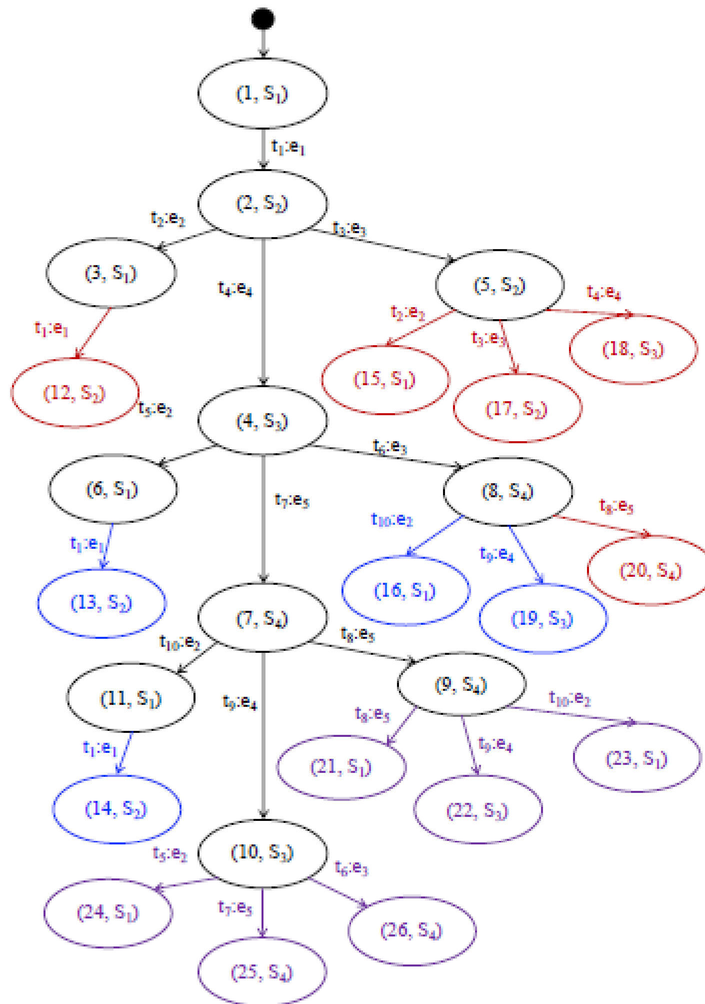

Figure 3.3 : Chaque arbre augmenté de notre exemple.

4. Conclusion

Le test de transition d'état est une approche utile lorsque différentes transitions de système doivent être testées pour des systèmes à états finis.

Les tests de transition d'état sont une approche de test unique pour tester des applications complexes, ce qui augmenterait la productivité d'exécution des tests sans compromettre la couverture des tests.

La limitation de cette technique est qu’elle ne peut pas être utilisée tant que le système testé n’a que des états finis.

Chapitre 04

Etude de cas et implémentation

Chapitre 04 : Etude de cas et implémentation

1. Introduction

Nous présentons dans ce chapitre une stratégie de test basée sur un modèle et nous construisons un outil pour fournir une approche standard et automatisée. Une brève étude de cas est utilisée pour montrer la faisabilité et l'intérêt de la technique proposée. Les chercheurs ont été plus intéressés par les tests basés sur des modèles, d'autant plus que les modèles sont devenus plus répandus dans conception et développement de logiciels.

2. Etude de cas

2.1 Conception d'études de cas

2.1.1 Logiciel sous test de test

Comme étude de cas nous avons considéré le système de régulation de vitesse d'une voiture (Cruise Control System, CCS). Cet exemple a été largement utilisée dans des études antérieures pour évaluer l'efficacité et le coût des suites de tests. CCS simule le moteur de la voiture et son régulateur de vitesse. Il se compose d'états simples et de transitions sans conditions de garde. La figure suivante montre le diagramme d'état de cet exemple.

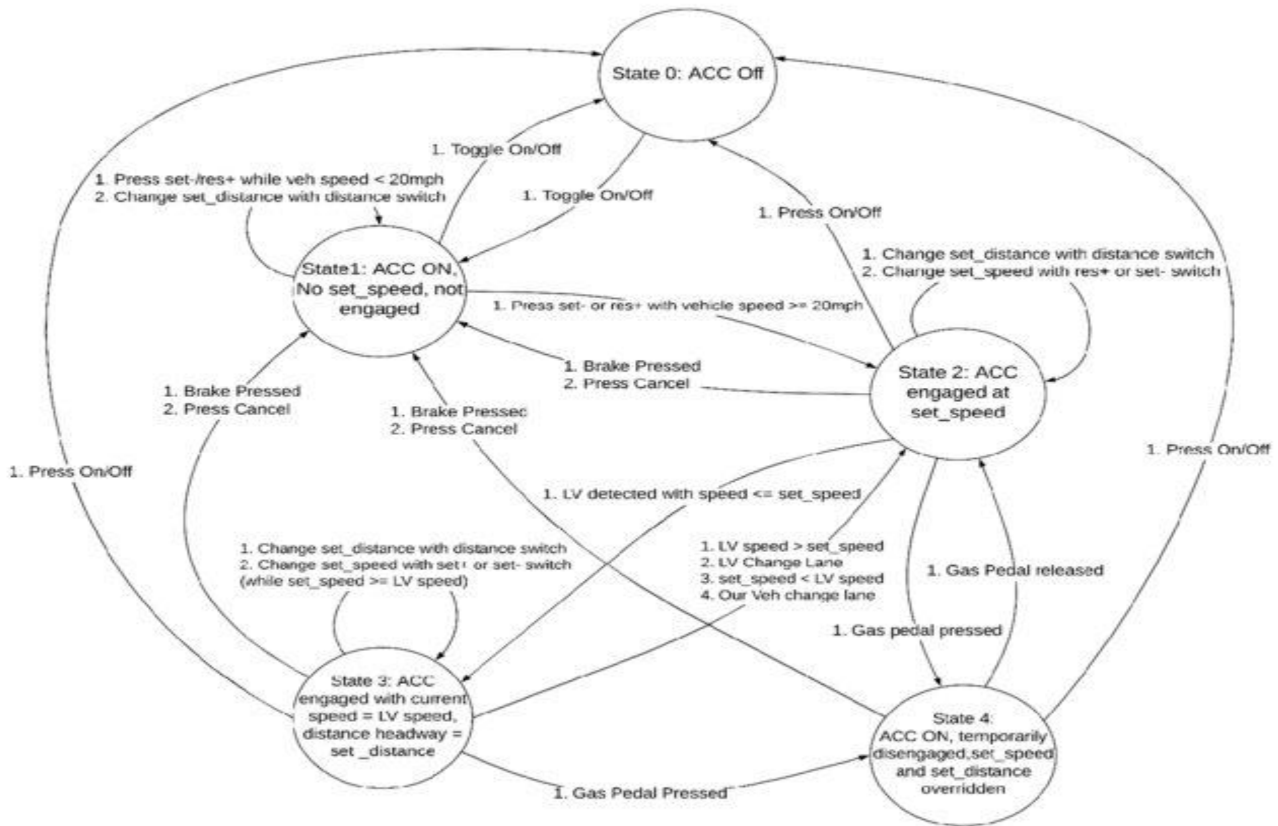

Figure 4.1 : diagramme d'état de CCS

Nous avons modélisé ce système en utilisant ModelJUnit qui est une bibliothèque Java et interface graphique pour la prise en charge des tests basés sur des modèles. Les modèles sont des machines à états finis étendues (EFSM) écrites en Java.

Dans ModelJUnit, modèle de test (FSM) doit être écrit comme une classe java qui implémente plusieurs interfaces définies dans la bibliothèque tels que :

- **FsmModel :** Il s'agit de l'interface de base que toute classe (E) FSM doit mettre en œuvre pour la génération de tests basé modèle
- **TimedFsmModel :** Il s'agit d'une interface spéciale qui étend les fonctionnalités de l'interface FsmModel et construit le FSM qui utilise le framework temporisé de ModelJUnit

La classe de modèle créé à l'aide des interfaces ci-dessus doit avoir des méthodes suivantes:

- **Object getState() :** Cette méthode renvoie l'état actuel du modèle, qui est généralement un objet. Elle exécute la tâche de mappé l'état interne du modèle EFSM à l'état visible réelle représentée dans le modèle par le concepteur du modèle
- **Void reset (boolean) :** Cette méthode effectue la tâche de remise à zéro du SUT à l'état initial ou la création nouvelle instance de la classe SUT. Elle est utilisée généralement dans les tests

online, où nous avons besoin de réinitialiser SUT à l'état initial. La valeur par défaut du paramètre booléen est "vrai"

- **@ Action void name() :** Ces types de méthodes sont utilisées pour définir les actions dans le modèle. Ces actions modifient l'état d SUT. Il peut y avoir plus d'une méthode d'action définies dans un modèle. Ces méthodes ne nécessite que les annotations @ action et n'ont pas de paramètres. Chaque méthode d'action ne peut exister avec ou sans guard. Ce gaurd contrôle l'activation de l'action lors de la génération de la séquence de test. Dans le cas où ce gaurd n'est pas définit sa valeur par défaut est vraie
- **Boolean nameGuard() :** Il définit le gaurd pour les méthodes d'action définis dans le modèle. Cette méthode renvoie toujours booléen. Le nom de la méthode doit être le même que le nom de l'action (pour qui le gaurd est défini), avec le mot ajouté «gaurd», à son extrémité**.**

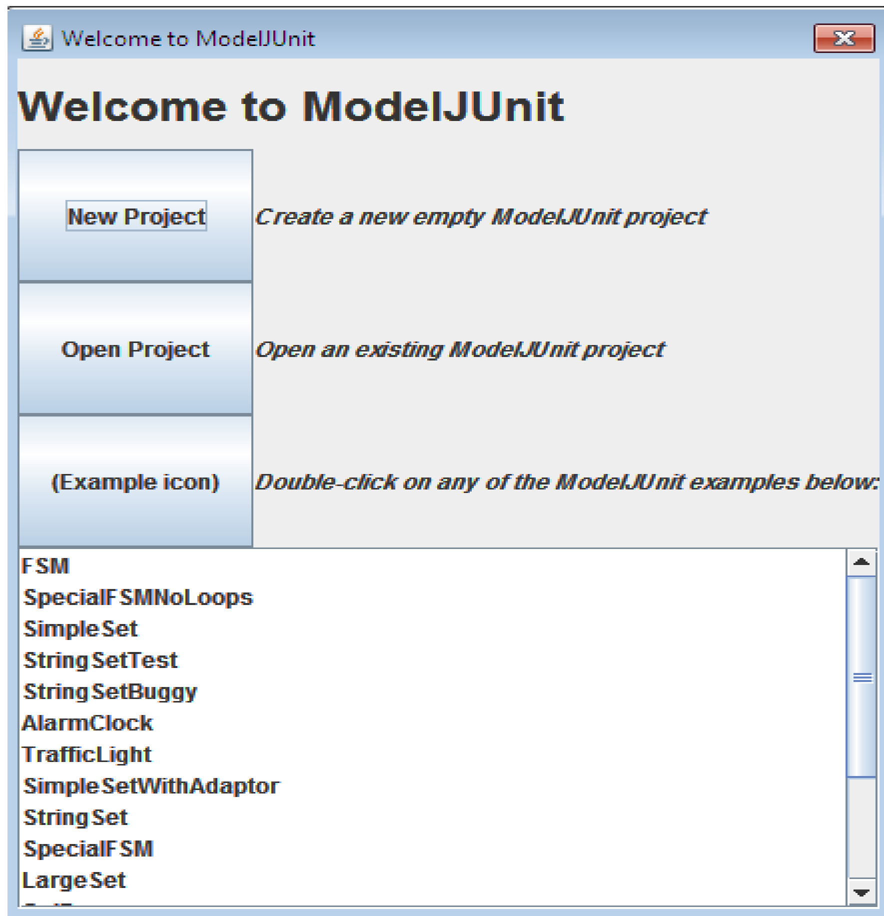

Figure 4.2 : Interface ModeJUnit

Cet exemple a été implémenté en Java en utilisant le modèle d'état. Le tableau 1 présente les principaux attributs de la machine d'état et l'implémentation du SUT.

Attributs du SUT		SUT CCS
attribut de machine d'état	N.d'états	5
	N.de transitions	17
	N.d'évenements	7
	N.de gardes	0
Attribut d'implémentation	Langue	Java
	N.de classes	7
	N.de méthodes	34
	N.d'attributs	14

Tableau 4.1 : Principaux attributs de la machine d'état et implémentation des SUT

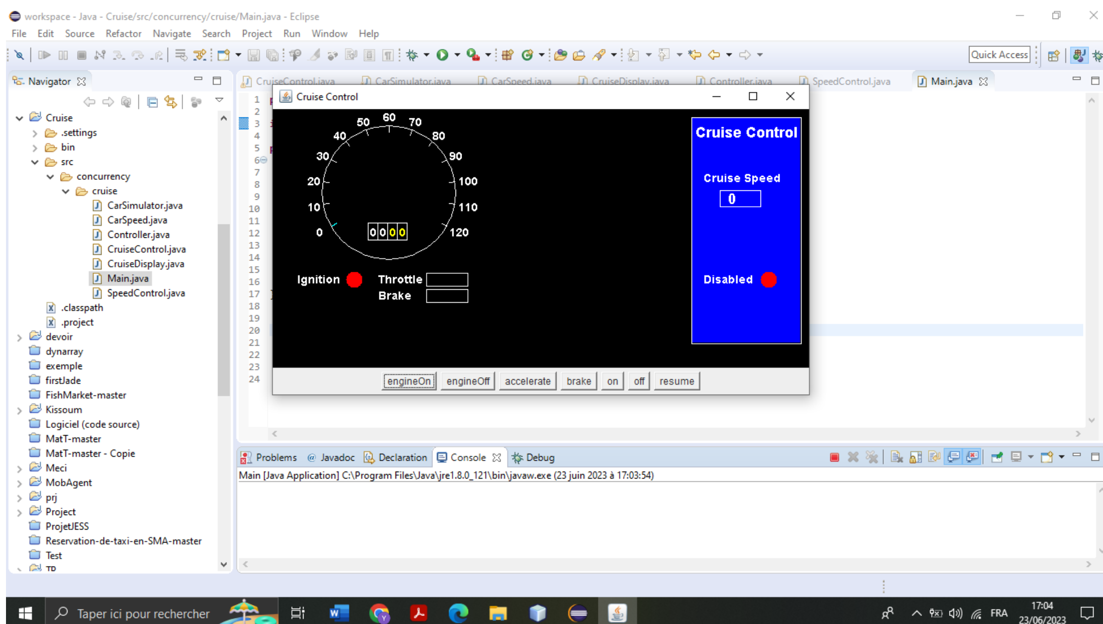

Figure 4.3 : Interface de l'application.

2.1.2 Test de mutation

Pour évaluer l'efficacité des suites de tests, nous avons effectué une analyse de mutation. L'analyse de mutation est une technique de test de logiciel qui évalue la qualité d'une suite de tests. Le processus consiste à introduire une petite modification (mutation) dans le code source du programme, qui simule une éventuelle erreur qu'un programmeur pourrait commettre lors du codage. La mutation aboutit à une nouvelle version du programme, appelée le mutant. Après avoir créé les mutants, la suite de tests est exécutée sur chaque mutant pour vérifier si la suite de tests peut détecter le changement dans le mutant. Si la suite de tests peut détecter le changement, le mutant est considéré comme tué. Si la suite de tests ne détecte pas le changement, le mutant est considéré comme vivant. Cela peut aider à identifier les faiblesses d'une suite de tests et à améliorer la qualité du logiciel testé. Un problème dans l'analyse des mutations est le coût de la génération de mutants. Nous créons des mutants automatiquement en utilisant muJava **[27]** car cela permet aux utilisateurs de générer rapidement un grand nombre de fautes. Les mutants sont générés en appliquant toutes les mutations applicables pour réduire le biais causé par les opérateurs de mutation. Au cours de l'analyse, il est important d'identifier les mutants équivalents qui présentent un comportement identique à celui du SUT d'origine et ne peuvent pas être tués par une suite de tests. Une heuristique couramment utilisée considère les mutants qui ne sont tués par aucun cas de test dans le groupe de test global comme des mutants équivalents **[25]**. En conséquence, nous avons considéré les mutants dans lesquels une suite ATP ne peut pas tuer comme des mutants équivalents et les avons exclus des mutants générés.

µJava (muJava) est un système de mutation pour les programmes Java. Il génère automatiquement des mutants pour les tests de mutation traditionnels et les tests de mutation au niveau de la classe. µJava peut tester des classes individuelles et des packages de plusieurs classes. Les tests sont fournis par les utilisateurs sous forme de séquences d'appels de méthode aux classes testées encapsulées dans des méthodes dans des classes séparées des classes JUnit.

3. Conclusion

Dans ce chapitre nous avons mené une étude de cas sur l’exemple de CCS, on a vu son diagramme d’état et s’application et les tests de mutation pour évaluer l'efficacité de la suite RTP et des suites proposées dans la détection des défauts.

Conclusion Générale

Dans la culture actuelle, où les logiciels sont plus répandus dans notre vie quotidienne, la qualité des logiciels est devenant de plus en plus important. Pour caractériser et découvrir les défauts d'un logiciel, plusieurs vérifications approches ont été créées. Le test est l'approche la plus souvent utilisée.

L'objectif du génie logiciel est d'économiser de l'argent, de réduire le temps de développement et d'assurer une haute qualité des logiciels publiables. C'est le rêve de tous, qu'ils soient clients, utilisateurs, développeurs, ou testeurs, car le but des tests est d'évaluer la qualité plutôt que de l'améliorer. Son but est pour évaluer ou exécuter un programme afin de trouver autant de défauts que possible et d'exposer tout potentiel questions. Ou la conduite n'est pas celle que vous anticipiez. Les tests restent un moyen efficace d'évaluer un logiciel ou une fiabilité du composant logiciel.

Dans ce travail, nous avons présenté une approche de test basé modèle pour le système de régulateur de vitesse de voiture (CCS). Cette approche capable de concevoir et de dériver (de manière automatique ou non) des cas de tests à partir d'un modèle abstrait et haut niveau du système sous test (SUT). Le modèle est dit abstrait car il offre bien souvent une vue partielle et discrète des comportements attendus d'un logiciel ou d'un système.

Sur la base de modèles abstraits, des cas de test peuvent être dérivées sous la forme de suites de tests. Ces suites de tests ne sont pas directement exécutables, car elles n'ont pas le même niveau d'abstraction que le code exécutable. Cela demande souvent une intervention manuelle de la part d'un ingénieur de test qui doit concevoir une couche d'adaptation permettant de passer d'une suite de tests abstraites en suite de tests exécutables.

Une fois les cas de tests exécutés, une comparaison est possible entre le comportement réel du logiciel (le logiciel développé) et le comportement attendu (décrit dans le modèle).

Cependant, certains utilisateurs affirment que l'utilisation des MBT peut être un réel retour sur investissement avec un gain de productivité et une qualité augmentée. En effet, l’automatisation des tests a des avantages directs pour les équipes responsables des tests :

- Si le modèle est bien fait, évite des cas de test mal conçus, défectueux ou manquants, du même coup, accroît la couverture de test
- Réduit les coûts pour les tests (tests de non régression)
- Améliore la qualité du processus de test
- Réduction des délais d’exécution des tests

Dans notre travail on a testé « l'application de système de régulateur de vitesse », on a générer des cas de test basée sur le diagramme d'état.

Bibliographie

[1] Mémoire La génération des cas de test basée sur le diagramme de séquence AUML

[2] Mémoire Model-Based Testing of Multi-Agent Systems

[3] http://fr.m.wikipedia.org/wiki/logiciel

[4] https://fr.theastrologypage.com/software-testing

[5] Qu'est-ce que le test logiciel et comment fonctionne-t-il ? IBM https://www.ibm.com/fr-fr/topics/software-testing

[6] Qu'est-ce qu'un cycle de vie de test logiciel (STLC) ? |2022| ITTest Blog https://ittestgroup.com/conseils-astualites/cycle-de-vie-de-test-logiciel-stlc/

[7] Les différents types de tests logiciels | Atlassian https://www.atlassian.com/fr/continuous-delivery/software-testing/types-of-software-testing

[8] Meet Guru99 – Free Training Tutorials & video for IT Courses https://www.guru99.com

[9] Différent types de tests https://www.all4test.fr/differents-test-logiciel/

[10] Le test en mode Agile - - Tests de sureté de fonctionnement | Edition ENI https://www.editions-eni.fr/open/mediabook.aspx?idR

[11] Test de la boite noir https://fr.wikipedia.org/wiki/Test_de_la_bo%C3%AEte_noire

[12] Que signifie Test en boite blanche ? – Définition IT de whatis.fr https://www.lemagit.fr/definition/Test-en-boite-blanche

[13] Le MBT, Model Based Testing par Patrick Allgeyer

[14] Mémoire Test base sur les modèles appliqué aux lignes de produits « Hamza Samih »

[15] Mémoire Génération automatique de test d'acceptance par une approche de tests dirigée par les modèles pour une application web « Gautier Lebourg »

[16] Techniques basées sur les spécifications (4/7) – les tests de transition d'état

[17] https://fre.myservername.com/state-transition-testing-technique

[18] Diagramme états-transitions-Wikipédia https://fr.wikipedia.org/wiki/Diagramme_%C3%A9tats-transitions

[19] Turner et Robson 1993 Journal of King Saud University - Computer and Information Sciences Augmenting a round-trip path test suite to cover all event-pairs for testing more diverse usage scenarios

[20] Binder 2000 Journal of King Saud University - Computer and Information Sciences Augmenting a round-trip path test suite to cover all event-pairs for testing more diverse usage scenarios

[21] Mouchawrab et al., 2011 Journal of King Saud University - Computer and Information Sciences Augmenting a round-trip path test suite to cover all event-pairs for testing more diverse usage scenarios

[22] Briand et al., 2004 Journal of King Saud University - Computer and Information Sciences Augmenting a round-trip path test suite to cover all event-pairs for testing more diverse usage scenarios

[23] Khalil 2017 Journal of King Saud University - Computer and Information Sciences Augmenting a round-trip path test suite to cover all event-pairs for testing more diverse usage scenarios

[24] Khalil et Labiche 2010 Journal of King Saud University - Computer and Information Sciences Augmenting a round-trip path test suite to cover all event-pairs for testing more diverse usage scenarios

[25] DeMillo et al., 1978, Baldwin et Sayward, 1979 Journal of King Saud University - Computer and Information Sciences Augmenting a round-trip path test suite to cover all event-pairs for testing more diverse usage scenarios

[26] Briand et al., 2010, Pastore et al., 2020 Journal of King Saud University - Computer and Information Sciences Augmenting a round-trip path test suite to cover all event-pairs for testing more diverse usage scenarios

[27] Ma et al., 2005 Journal of King Saud University - Computer and Information Sciences Augmenting a round-trip path test suite to cover all event-pairs for testing more diverse usage scenarios