POLYPUBLIE

Méthodologie d'automatisation de tests appliquée aux routeurs configurables

Omar Mahrez

Document en libre accès dans PolyPublie
Open Access document in PolyPublie

Date	2005
Type	Mémoire ou thèse / Dissertation or Thesis Mémoire de maîtrise/Master's thesis
URL de PolyPublie PolyPublie URL	https://publications.polymtl.ca/7642/
Conditions d'utilisation Terms of use	© Tous droits réservés / All rights reserved
Référence Citation	Mahrez, O. (2005). Méthodologie d'automatisation de tests appliquée aux routeurs configurables [Mémoire de maîtrise, École Polytechnique de Montréal]. PolyPublie. https://publications.polymtl.ca/7642/

Détails de la thèse
Thesis details

Directeurs de recherche Advisors	Yvon Savaria, & Omar Cherkaoui
Programme Program	Non spécifié

Ce fichier a été téléchargé à partir de PolyPublie, le dépôt institutionnel de Polytechnique Montréal
This file has been downloaded from PolyPublie, the institutional repository of Polytechnique Montréal
https://publications.polymtl.ca

UNIVERSITÉ DE MONTRÉAL

MÉTHODOLOGIE D'AUTOMATISATION DE TESTS APPLIQUÉE AUX ROUTEURS CONFIGURABLES

OMAR MAHREZ

DÉPARTEMENT DE GÉNIE ÉLECTRIQUE

ÉCOLE POLYTECHNIQUE DE MONTRÉAL

MÉMOIRE PRÉSENTÉ EN VUE DE L'OBTENTION

DU DIPLÔME DE MAÎTRISE ÈS SCIENCES APPLIQUÉES

Décembre 2005

© Omar Mahrez, 2005.

Library and Archives Canada

Published Heritage Branch

395 Wellington Street
Ottawa ON K1A 0N4
Canada

Bibliothèque et Archives Canada

Direction du Patrimoine de l'édition

395, rue Wellington
Ottawa ON K1A 0N4
Canada

Your file Votre référence
ISBN: 978-0-494-16810-3
Our file Notre référence
ISBN: 978-0-494-16810-3

NOTICE:
The author has granted a non-exclusive license allowing Library and Archives Canada to reproduce, publish, archive, preserve, conserve, communicate to the public by telecommunication or on the Internet, loan, distribute and sell theses worldwide, for commercial or non-commercial purposes, in microform, paper, electronic and/or any other formats.

The author retains copyright ownership and moral rights in this thesis. Neither the thesis nor substantial extracts from it may be printed or otherwise reproduced without the author's permission.

AVIS:
L'auteur a accordé une licence non exclusive permettant à la Bibliothèque et Archives Canada de reproduire, publier, archiver, sauvegarder, conserver, transmettre au public par télécommunication ou par l'Internet, prêter, distribuer et vendre des thèses partout dans le monde, à des fins commerciales ou autres, sur support microforme, papier, électronique et/ou autres formats.

L'auteur conserve la propriété du droit d'auteur et des droits moraux qui protège cette thèse. Ni la thèse ni des extraits substantiels de celle-ci ne doivent être imprimés ou autrement reproduits sans son autorisation.

In compliance with the Canadian Privacy Act some supporting forms may have been removed from this thesis.

While these forms may be included in the document page count, their removal does not represent any loss of content from the thesis.

Conformément à la loi canadienne sur la protection de la vie privée, quelques formulaires secondaires ont été enlevés de cette thèse.

Bien que ces formulaires aient inclus dans la pagination, il n'y aura aucun contenu manquant.

Canada

ÉCOLE POLYTECHNIQUE DE MONTRÉAL

Ce mémoire intitulé :

MÉTHODOLOGIE D'AUTOMATISATION DE TESTS APPLIQUÉE AUX ROUTEURS CONFIGURABLES

Présenté par: MAHREZ Omar

en vue de l'obtention du diplôme de : Maîtrise ès sciences appliquées

a été dûment accepté par le jury d'examen constitué de :

Mme. NICOLESCU Gabriela, Ph.D., Présidente

M. SAVARIA Yvon, Ph.D., membre et directeur de recherche

M. CHERKAOUI Omar, Ph.D., membre et Co-directeur de recherche

M. BOYER François-Raymond, Ph.D., membre

REMERCIEMENTS

Je tiens à remercier mon Directeur de recherche M. Yvon Savaria de m'avoir accepté au sein du laboratoire GRM, et grâce à qui j'ai pu accéder au laboratoire téléinformatique de l'UQAM, dans lequel s'est déroulé la quasi-totalité de ma recherche.

J'aimerais remercier également mon co-directeur et directeur du laboratoire téléinformatique de l'UQAM, M. Omar Cherkaoui, qui m'a offert l'opportunité de réaliser ma recherche en milieu industriel, et qui m'a encadré et soutenu tout au long des projets que j'ai eu à réaliser.

Je tiens à exprimer ma gratitude envers mon ami de longue date, Driss Bohsina pour ses encouragements et son support tout au long de ces années que j'ai passées à l'École Polytechnique.

Je voudrais également remercier toutes les personnes de L'entreprise dans laquelle s'est déroulée ma recherche, qui m'ont aidé à acquérir des connaissances importantes dans le monde des routeurs, et qui m'ont fourni l'environnement adéquat afin de réaliser mes travaux.

Finalement, je suis très reconnaissant envers tous les collègues et amis qui m'ont soutenu pendant les périodes difficiles, et avec qui j'ai partagé des moments très agréables.

Merci Hélène pour ta patience.

RÉSUMÉ

Depuis l'avènement d'Internet, les services réseaux n'ont cessé d'augmenter et leur complexité devient de plus en plus grandissante. Cette croissance est dû essentiellement à l'augmentation de la demande du marché, et elle est accompagnée d'une volonté des concepteurs d'offrir des produits sécuritaires et de qualité. Pour arriver à cette fin, les industries investissent à hauteur de 60% du coût de développement de leurs produits dans les tests.

Pour les entreprises, il est important de fournir des produits de qualité, mais le souci de création de nouvelles stratégies pouvant minimiser les coûts associés aux tests fait partie également de leurs priorités.

Le présent travail a été réalisé au cours de deux projets de recherche au sein d'une entreprise qui commercialise des routeurs et qui développe des services réseaux. Le but principal étant de trouver des méthodologies pouvant améliorer le processus de test afin d'en minimiser les coûts.

Au cours du premier projet, nous avons développé une application permettant la réutilisation des scripts de tests existants tout en modifiant leur contenu selon les nouveaux paramètres de services et d'environnement de test. Un concept d'abstraction de services et de structuration de données a été mis en application afin d'atteindre ces objectifs.

Le deuxième projet est relatif aux tests simultanés de plusieurs services sur plusieurs équipements. Nous avons développé une technique basée sur les attributs des paramètres de cas de test relatifs à des services distincts. Cette technique permet de comparer ces paramètres, de les combiner et ainsi tester plusieurs services simultanément. Des scénarios de tests ont été réalisés grâce à cette technique, et des cas d'interactions de service ont été simulés pour

montrer comment, à travers cette technique, il est possible de détecter d'éventuelles interactions entre les services.

Les tests ont été réalisés sur des bancs d'essais constitués de routeurs et de générateur de trafic de type Agilent.

Les résultats du premier projet se limitent à des Scripts de tests dont la création est partiellement automatisé par l'outil développé à cet effet. Nous avons pu réutiliser des composantes de tests pour créer des nouveaux tests en se basant sur des scénarios de changement d'environnement de test, et de services configurés sur les routeurs.

Pour le second projet, les résultats de tests ont prouvé d'une part, qu'il est possible de combiner de façon automatique des cas de test valides afin d'examiner plusieurs services de façon simultanée, et d'autre part, de simuler des interactions (non désirées) entre les services.

Tout au long des deux projets, plusieurs pistes de recherche ont été explorées. Elles constituent un ensemble d'idées complémentaires au présent travail et peuvent faire l'objet de travaux futurs.

ABSTRACT

Since the advent of Internet, networks' services increased as well as their complexity. This growth is due to the increase of the market demand, and the designers worked more to offer reliable and secure products. To reach this goal, the companies invest around 60% of their products development cost in the testing processes.

For the companies, it is significant to provide higher quality products, but they also care about the associated costs, and therefore, they try to develop and create new test strategies which can minimize their costs.

The present work was completed during two research projects within a company that manufactures routers and develops network services. The main goal is to develop methodologies that can improve the testing process in order to minimize the costs.

During the first project, we developed an application allowing the re-use of scripts of existing tests while modifying their contents according to new services and environment parameters. A concept of abstraction of services and structuring of data was developed to achieve these goals.

The second project was dedicated to the simultaneous tests on several services and equipments. We developed a technique based on attributes of parameters which enable the combination of test-cases and therefore the test of several services simultaneously. Test scenario were developed based on this technique, and some simulations were performed to show how it is possible to detect features interactions.

The tests were carried out on test-beds made up of modular routers and a traffic generator (Agilent).

The results of the first project are limited to test scripts generated automatically by the tool developed for this purpose. We could re-use components of tests to create new tests, while using scenarios for different tests' environments, and different services configured on the routers.

In the second project, the test results proved on one hand that it is possible to combine in an automatic way valid test-cases, in order to examine several services simultaneously, and on the other hand, to stimulate and identify interactions between services.

Throughout the two projects, several research avenues were explored. They constitute a set of ideas complementary to the present work and can be the subject of future work.

Table Des Matières

REMERCIEMENTS iv

RÉSUMÉ v

ABSTRACT vii

Table Des Matières ix

Liste des figures xv

Liste des tableaux xvii

Liste des sigles et des abréviations xviii

Liste des Annexes xix

Avant-propos xx

Chapitre 1 Introduction 1

Chapitre 2 Revue de littérature 4

2.1 Routeur 4

2.1.1 Définition 4

2.1.2 Architecture du routeur 5

2.1.2.1 Mémoire flash 5

2.1.2.2 Mémoire vive (RAM) 6

2.1.2.3 Mémoire morte (ROM) 6

2.1.2.4 Mémoire non volatile (NVRAM) 6

2.1.3 Routeurs modulaires 7

2.1.4 Moteur de routage 8

2.2 Topologie 9

2.3 Trafic 9

2.4 Configuration 9

2.5 Les services réseau configurés sur les routeurs 10

2.6 L'interaction de services 11

2.7 Les tests de régression 12

2.8 Plan de test 13

2.9 Cas de test 13

2.10 Suite de tests 13

2.11 En résumé 14

Chapitre 3 Méthodologies actuelles de test 15

3.1 Approche de test dans un contexte logiciel 15

3.1.1 Stratégie de test 15

3.1.2 Outils 16

3.2 Approche de test dans un contexte matériel 17

3.2.1 Stratégie de test....17

3.2.2 Outils....18

3.3 Similitude avec le test d'un routeur....20

3.4 Cas d'un routeur....21

3.4.1 Considérations de base....23

3.4.2 Spécifications....24

3.4.3 Plan de test....24

3.4.4 Suite de test....25

3.4.5 Cas de test....26

3.4.6 Analyse des résultats....26

3.5 En résumé....28

Chapitre 4 Réutilisation des tests : ScriptMaker....29

4.1 Le modèle Meta-CLI....30

4.2 Abstraction par rapport à l'environnement de test....32

4.3 Structuration des scripts de test....34

4.3.1 Décomposition du script de test....36

4.3.1.1 Service....36

4.3.1.2 Trafic....36

4.3.1.3 Routage 37

4.3.1.4 Validation 37

4.4 La structure du script de test 37

4.5 Le générateur de Scripts : ScriptMaker 41

4.5.1 Architecture du ScriptMaker 42

4.5.1.1 Module de gestion et de contrôle de l'information 43

4.5.1.2 Module de Sélection de données de test 44

4.5.2 Module de Gestion de structures 48

4.5.3 Module d'Instanciation des structures 49

4.5.4 Module Meta-CLI 51

4.5.5 Scénarios d'utilisation du ScriptMaker 51

4.5.5.1 Ajout de cas de test à un script de test 51

4.5.5.2 Modification d'une composante de cas de test 52

4.5.5.3 Modification de l'environnement de test 53

4.6 Résultats découlant du développement de l'outil ScriptMaker 55

4.7 Analyse et discussions 57

4.7.1 Rôle du module Sélecteur dans le ScriptMaker 58

4.7.2 Avantages et lacunes de l'outil ScriptMaker 60

4.8 En résumé 61

Chapitre 5 Combinaisons de cas de test 62

5.1 Détection de l'interaction de Service 63

5.2 Réutilisation des cas de test 64

5.2.1 Cas de test générique 65

5.2.2 Combinaison des cas de test 66

5.2.3 Prise de décision 69

5.3 Cas d'un routeur 71

5.3.1 Hypothèses 72

5.3.2 Règles de combinaison 72

5.3.2.1 Règles structurelles 72

5.3.2.2 Règles comportementales 73

5.4 Description du fonctionnement du banc d'essai 76

5.5 Processus d'exécution des tests 78

5.6 Combiner des tests associés à différents services 78

5.7 Résultats des tests : analyse et interprétation 79

5.7.1 Résultats de tests sur Config1 80

5.7.2 Résultats de tests pour Config2 82

5.7.3 Combinaison des tests pour Config1 84

5.7.4 Combinaison des tests pour Config2 86

5.7.5 Reconstitution d'un scénario d'interaction de Service 87

5.7.5.1 Cas connu d'Interaction 88

5.7.6 Autre scénario d'interaction de service 89

Chapitre 6 Conclusion 91

Références 94

Liste des figures

Figure 1. Topologie minimale pour un routeur....4

Figure 2. Routeur de type modulaire....7

Figure 3. Processus de configuration de service....10

Figure 4. Schéma global de test automatique de SpecmanElite....19

Figure 5. Processus de test d'un routeur....22

Figure 6. Structure générique de service basée sur le Meta-CLI....31

Figure 7. Schéma d'un service dans deux environnements différents....33

Figure 8. Impact de l'environnement sur les paramètres de service....34

Figure 9. Décomposition du script de test en modules réutilisables....35

Figure 10. Schéma global de la structure d'un script de test générique....38

Figure 11. Instanciation d'une structure de script de test....40

Figure 12. Entrée/Sortie de l'application ScriptMaker....41

Figure 13. Architecture de l'application ScriptMaker....42

Figure 14. Interaction entre l'usager et le ScriptMaker....43

Figure 15. Architecture du module Sélecteur....45

Figure 16. Fichiers générés par le Sélecteur....47

Figure 17. Architecture du module Gestion de structures....48

Figure 18. Architecture du module d'Instanciation de structures.....50

Figure 19. Scénario d'ajout de cas de test au script de test.....52

Figure 20. Scénario de modification d'une partie du cas de test.....53

Figure 21. Scénario de modification de l'environnement de test.....54

Figure 22. Banc d'essai pour le ScriptMaker.....55

Figure 23. Regroupement de plusieurs tests dans un script de test.....58

Figure 24. Exemple d'interaction de service 59

Figure 25. Répartition des ressources entre deux services 64

Figure 26. Combinaison de cas de test relatifs à des services distincts.....65

Figure 27. Structure générique de cas de test 66

Figure 28. Comparaison de cas de test par type d'attribut.....67

Figure 29. Algorithme de combinaison de cas de test.....68

Figure 30. Comparaison structurelle 69

Figure 31. Exemple de combinaison de paramètres de type numérique 70

Figure 32. Aperçu du banc de test et son environnement.....77

Figure 33. Temps d'exécution de cas de test et de suite de test 81

Figure 34. Temps d'exécution pour divers cas de test.....86

Liste des tableaux

Tableau 1. Exemple de table de décision pour la combinaison de cas de test..74

Tableau 2. Les résultats prévisibles après combinaison des tests 79

Tableau 3. Tests singuliers sur la configuration Config1 80

Tableau 4. Tests singuliers sur la configuration Config2 83

Tableau 5. Tests combinés sur la configuration Config1 84

Tableau 6. Le nombre d'ACL < n.....88

Tableau 7. Le nombre d'ACL > n.....89

Tableau 8. résultats des tests pour la config4. 89

Tableau 9. Tests combinés sur la configuration Config2 99

Tableau 10. Tableau descriptif des tests 101

Liste des sigles et des abréviations

ACL	:	Access-Control Lists
API	:	Application Programming Interface
ARP	:	Address Resolution Protocol
ATM	:	Asynchronous Transfer Mode
BFM	:	Bus Functional Model
CLI	:	Command Line Interface
CPU	:	Central Processing Unit
EPROM	:	Erasable Programmable Read Only Memory.
FE	:	Fast Ethernet
FIFO	:	First In First Out
GE	:	Gigabit Ethernet
I/O	:	Input / Output
OS	:	Operating System (Système d'exploitation)
IP	:	Internet Protocol
RC	:	Router Card
MPLS	:	Multi-Protocol Label Switching
NVRAM	:	Non-Volatil Memory

OSPF	:	Open Shortest Path First
POS	:	Packet Over Sonet
QOS	:	Quality Of Service
RAM	:	Random Access Memory (Mémoire vive)
ROM	:	Read Only Memory (Mémoire morte)
TFTP	:	Trivial File Transfer Protocol
XML	:	eXtensible Markup Language

Liste des Annexes

Annexe 1. Sommaire des Classes du package Selector 98

Annexe 2. Tests combinés pour Config2 99

Annexe 3. Description des cas de test utilisés.....101

Avant-propos

La plupart des entreprises multinationales possèdent un budget pour la recherche et développement, cela permet d'améliorer la qualité de leurs produits mais aussi d'économiser sur leurs coûts de développements et de production.

Le présent travail s'est déroulé au sein d'une entreprise qui produit des routeurs et des services pour les réseaux. Par conséquent plusieurs règles de confidentialité et de non divulgation d'informations doivent être respectées; on ne peut donc mentionner dans ce document toute information confidentielle appartenant à cette entreprise ou à un de ses collaborateurs impliqués dans le projet; cela inclut les accès aux locaux et laboratoires, les équipements ne faisant pas partie du domaine public, les logiciels, l'identité des personnes avec lesquelles nous avons collaborés, et toute méthodologie ou stratégie non publique.

Chapitre 1 Introduction

L'avènement d'Internet a bouleversé les télécommunications en rendant les réseaux téléinformatiques plus complexes et plus exigeants en termes d'équipements et de logiciels. Le routeur joue un rôle central dans cet échiquier, il fait partie de l'équipement nécessaire à l'acheminement de l'information entre les réseaux et offre un certain nombre de services tels que la sécurité et la qualité de service. L'évolution du routeur s'est accentuée dans les dix dernières années grâce à la perpétuelle croissance des exigences du marché. Ce dernier s'oriente de plus en plus vers un traitement de la convergence de l'information (données, voix, vidéo, messagerie, etc.), cette convergence augmente les défis de qualité de service, de performance et de sécurité. Au-delà des composantes matérielles du routeur, les services qui y sont configurés sous forme logicielle doivent répondre également aux exigences de sécurité, de performance et de qualité de service. Ainsi, la complexité croissante des routeurs et des services qui y sont configurés pousse les concepteurs à aiguiser les phases de tests pour mieux garantir un produit de qualité et éviter ainsi des coûts de correction d'erreurs exorbitants.

Comme dans un cycle de développement logiciel, le déploiement de services sur les routeurs passe aussi par une phase de test. Ce mémoire s'intéresse essentiellement aux méthodes de tests des routeurs configurables, il traite des possibilités de réutilisation des tests unitaires et de la structuration du processus de test. Le cas de l'interaction de services sera examiné à la lumière des contributions apportées dans le cadre de ce mémoire.

Les tests représentent plus de 65% des coûts de développement d'un produit matériel ou logiciel. Ce pourcentage élevé, constitue un véritable défi à relever pour les entreprises et les chercheurs. Par ailleurs, pour tester un routeur, les ingénieurs développent habituellement des scripts de test personnalisés, ce qui

rend parfois leur compréhension et leur réutilisation difficiles. Ainsi, plusieurs équipes peuvent développer des scripts qui serviront à tester les mêmes composantes et services; cette redondance de tests induit forcément une augmentation du coût du produit. D'autre part, si deux services, possédant chacun un ensemble de tests unitaires, se retrouvent configurés sur le même routeur et qu'on désire tester leurs fonctionnalités de façon simultanée, il faut appliquer tous les cas de test de chacun de ces services, ce qui aura comme conséquence, un temps exorbitant d'utilisation du banc d'essai et un manque de contrôle de l'interaction pouvant exister entre ces services. Autrement, il faut créer de nouveaux cas de test regroupant les deux services, ce qui implique des coûts de développements supplémentaires.

L'objectif du présent travail étant d'apporter une contribution visant à améliorer les méthodologies de génération et de réutilisation des tests s'appliquant à des routeurs; il comporte deux volets essentiels : l'automatisation de la création de scripts de tests par composition de modules, puis la combinaison de cas de test dans le but de détecter des interactions de service.

Le premier volet traite de la construction d'un test unitaire par composition du contenu à partir de petits modules réalisant chacun une tâche particulière et dont l'interdépendance est faible. Cette faible dépendance facilite la réutilisation des modules du test et minimise les contraintes à considérer au moment de la création du test. Le banc d'essai utilisé est composé de plusieurs routeurs réunis dans une topologie particulière; les tests s'appliquent à une carte donnée d'un routeur désigné à être sous test. D'autre part, des structures de bases de données et de fichiers libellées sont dressées afin de conserver l'information utile pour les cas de test, et ainsi faciliter son repérage et sa réutilisation.

Le deuxième volet traite de la combinaison des cas de test relatifs à des fonctionnalités ou services distincts. Chaque cas de test est représenté par un

ensemble de paramètres ayant chacun un rôle particulier et une importance plus ou moins élevée. Il s'agit donc d'affecter à chaque paramètre d'un test unitaire des attributs pouvant l'identifier et qui sont représentatifs du test. Ce processus permet, au moment de la combinaison des tests, de comparer les attributs et de prendre une décision pour les paramètres et les valeurs du cas de test résultant. Cette technique a été utilisée dans un banc d'essai comportant un générateur de trafic relié à deux routeurs comportant chacun plusieurs cartes de divers types. Ce banc d'essai permet non seulement les tests de plusieurs services sur une même carte, mais également plusieurs services sur plusieurs cartes de façon simultanée.

Ce mémoire est composé de six chapitres dont la présente introduction. Le deuxième chapitre constitue une revue de littérature ainsi qu'une introduction théorique au domaine du test des routeurs, il permettra au lecteur d'avoir une meilleure compréhension des enjeux des tests dans ce domaine. Le troisième chapitre décrit de façon sommaire les pratiques existantes des tests dans le domaine logiciel, puis le domaine matériel et finalement dans le contexte des routeurs. Le quatrième chapitre est réservé à la composition des tests. On commence par décrire les objectifs d'un test unitaire, montrer comment le décomposer en modules (trafic, topologie, service, validation) et finalement la manière dont ces modules peuvent être structurés et recomposés pour donner de nouveaux cas de test. Le cinquième chapitre est complémentaire au troisième chapitre, il décrit comment on peut réduire le nombre de cas de test en se basant sur les modules et les attributs des paramètres qui les composent. Ce chapitre traitera des résultats obtenus et illustrera l'importance de la technique de composition des tests dans le cas particulier de l'interaction de service. Finalement, le chapitre six viendra conclure ce mémoire en résumant les diverses contributions, les limites et les contraintes rencontrées, ainsi que d'éventuels travaux futurs dans le contexte des routeurs.

Chapitre 2 Revue de littérature

Pour mieux faciliter la lecture des chapitres suivants, le présent chapitre décrit l'environnement matériel et logiciel ainsi que les principaux concepts entourant les tests de régression effectués sur les routeurs. Ainsi on parlera de quelques notions fondamentales sur le routeur, de son système d'exploitation, des services qui y sont déployés et des diverses techniques utilisées pour réaliser des tests.

2.1 Routeur

Cette section décrit l'architecture d'un routeur et le rôle de chacune des composantes qui le constituent [32].

2.1.1 Définition

Un routeur est un dispositif matériel et logiciel qui détermine le prochain point du réseau auquel un paquet de données devrait être expédié en route vers sa destination. Le routeur est relié au moins à deux réseaux (figure 1), il détermine de quelle manière envoyer chaque paquet de données en se basant sur sa représentation courante de l'état des réseaux auxquels il est relié. Les routeurs maintiennent une table des itinéraires disponibles et emploient cette information pour déterminer le meilleur itinéraire pour un paquet de données [6].

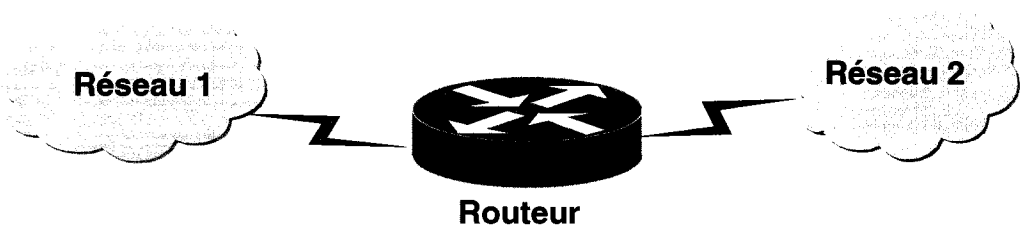


Figure 1. Topologie minimale pour un routeur

Au-delà de la fonction de base du routeur, à savoir le routage des paquets, les routeurs jouent différents rôles selon les services qui y sont configurés, selon leur emplacement dans une topologie, et aussi selon le domaine d'utilisation.

2.1.2 Architecture du routeur

Il existe plusieurs types de routeurs : logiciel [17], matériel [12][24] ou en système embarqué [26][7]. Ce dernier n'est rien d'autre qu'un ordinateur spécialisé utilisant un logiciel d'exploitation fait sur mesure. Afin de stocker son système d'exploitation et la configuration, le routeur dispose de diverses mémoires [23]. Généralement, il y a quatre types de mémoires dans un routeur : flash, RAM, ROM, et NVRAM. Conceptuellement la mémoire Flash et la NVRAM

2.1.2.1 Mémoire flash

Placée dans un support pour mémoire morte programmable effaçable (EPROM), la mémoire flash est employée pour stocker le système d'exploitation. Cette mémoire a la faculté de conserver toutes les données essentielles, même en cas de redémarrage du routeur. Pendant le fonctionnement de ce dernier, le contenu de la mémoire flash est placé en mode lecture seulement.

La taille de la mémoire flash est variable dépendamment du routeur, du système d'exploitation et de la configuration. Généralement, un minimum de 4Mo de mémoire flash est requis pour un routeur doté d'une configuration de base, et 16Mo pour une configuration plus avancée. Lorsque le routeur est redémarré, l'OS présent dans la mémoire flash est chargé dans la mémoire vive.

La mémoire flash représente donc une sécurité importante qui permet au routeur de s'autogérer et de trouver une porte de sortie en cas de bogue majeur. On distingue la mémoire flash de la NVRAM à cause du rôle qui leur est attribué et par un facteur historique lié à la conception des premiers routeurs.

2.1.2.2 Mémoire vive (RAM)

La mémoire vive (RAM) représente l'emplacement de travail non permanent ou volatil d'un routeur, quand celui-ci est mis hors tension, le contenu de la RAM est perdu. Par défaut, la RAM est répartie en deux domaines principaux : mémoire principale du processeur, et mémoire partagée d'entée/sortie (I/O). La mémoire principale du processeur est celle où les tables (routage, ARP, etc.) et la configuration active sont stockées. La mémoire partagée d'entrée/sortie est employée pour stocker temporairement les paquets avant de traiter. La plupart des routeurs ont une mémoire vive intégrée et des fentes pour de la mémoire additionnelle.

2.1.2.3 Mémoire morte (ROM)

La ROM est une zone mémoire à partir de laquelle un routeur amorce son démarrage par l'intermédiaire du microcode (équivalent du BIOS d'un PC).

Lorsque le routeur est mis sous tension, le microcode stocké dans la ROM est exécuté pour s'assurer que les éléments tels que l'unité centrale de traitement, la mémoire, et les interfaces sont capables de fonctionner correctement. Il est également responsable de la localisation et du chargement de l'OS.

Finalement, si le serveur tftp est inaccessible et si la mémoire flash ne contient pas d'image OS, alors, la mémoire ROM enclenche un microcode qui représente une version réduite de l'OS pouvant être chargée au démarrage du routeur.

2.1.2.4 Mémoire non volatile (NVRAM)

La mémoire vive non-volatile (NVRAM) conserve la configuration de démarrage du routeur (startup-configuration). Dès son démarrage, ce dernier copie cette configuration dans la mémoire vive (running-config) et applique les règles et

contraintes de la configuration. Toute modification de la configuration active doit être sauvegardée dans la NVRAM avant un redémarrage du routeur.

2.1.3 Routeurs modulaires

Dans le cadre de ce mémoire, les routeurs utilisés comme banc d'essai sont de type modulaire. En plus des éléments de base cités dans la section précédente, ces routeurs possèdent plusieurs cartes de différents types, chaque carte a plusieurs interfaces, et chaque interface comporte plusieurs ports [16][11].

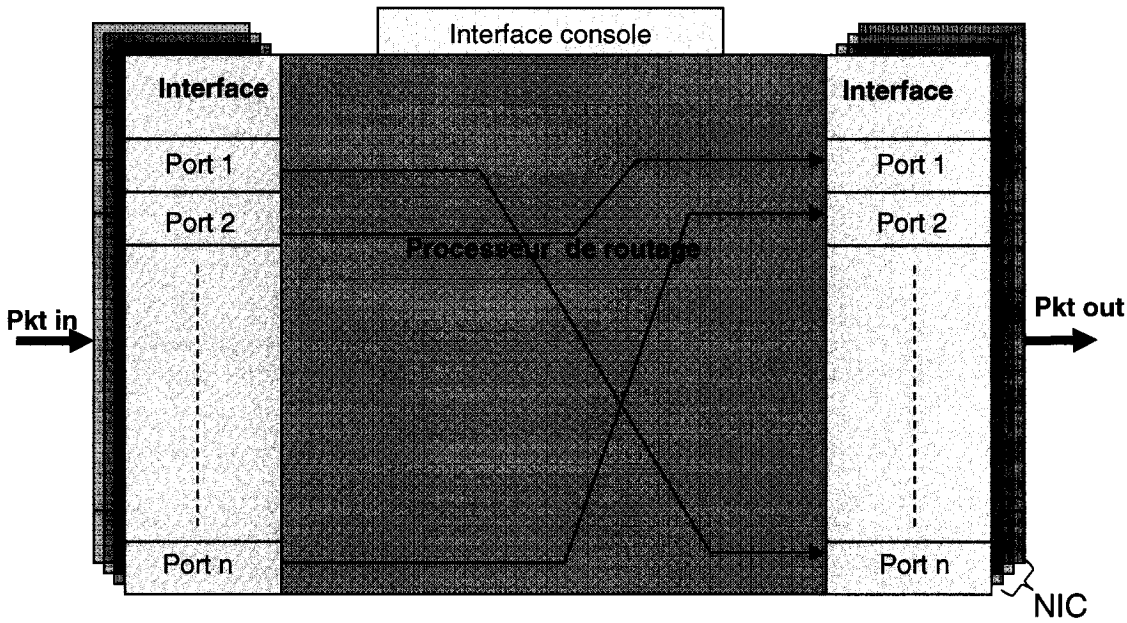


Figure 2. Routeur de type modulaire

Ce routeur modulaire en architecture distribuée possède des fentes d'extension pouvant accueillir plusieurs cartes (NIC). Les NIC du routeur reliées aux réseaux, exécutent chacune une copie du système d'exploitation et assurent le routage des paquets [19].

Au cœur d'une NIC il y a un processeur qui traite les paquets selon les informations qu'ils portent et les transmet à la carte de commutation. Le CPU de la NIC reçoit et transmet les paquets en utilisant des files pour éviter tout engorgement.

Les autres cartes présentes dans le routeur sont chargées d'assurer la gestion des transactions entres les cartes réseau, la communication et le transfert des paquets entre les cartes, et finalement l'ordonnancement et la synchronisation des opérations dans le routeur.

2.1.4 Moteur de routage

Les routeurs modulaires supportent différents types de cartes (ATM, POS, FastEthernet, etc). Chaque carte a ses spécificités et joue un rôle particulier selon le type de moteur de routage qu'elle possède.

Comme pour la version d'OS, le type de moteur de routage peut (ou non) supporter tel ou tel service, parfois même, il peut supporter les services de façon indépendante mais pas conjointement. Par ailleurs, plusieurs types de NIC peuvent avoir le même type de moteur de routage; et un même type de NIC peut avoir plusieurs types de moteurs de routage. Il faut donc considérer ces derniers lors de la configuration du routeur afin d'assurer un bon fonctionnement des services activés.

La disposition des routeurs modulaires dans un réseau est variable, cela leur permet de jouer différents rôles (Ingress, Egress, etc). L'ensemble de ces routeurs forment alors la topologie du réseau [25].

2.2 Topologie

Dans la littérature, on retrouve les termes 'Anneau', 'Étoile' et 'Bus' pour désigner la disposition d'un certain nombre de d'équipements dans un réseau donné. On parle alors d'une topologie en Anneau, ou topologie en Busetc.

Dans le cas des tests de routeurs, le terme 'topologie' désigne la disposition des routeurs et leur rôle dans un banc de test. Ainsi une topologie comporte un petit ensemble de routeur (5 maximum) et des générateurs de trafic (2 max). Le rôle joué par chaque routeur ou carte d'un routeur est défini par sa position dans la topologie et par les services qui y ont été configurés. Les données de topologie sont conservées dans chaque routeur, et elles se transmettent entre routeurs selon les contraintes de la configuration active.

2.3 Trafic

Le trafic représente l'ensemble des activités se produisant à l'entrée et à la sortie des ports d'un routeur. Le trafic peut donc être de type IP, MPLS, Multicast, etc., parfois on parle aussi de flux, de flot ou de 'stream'. Dans le cadre des tests, le trafic est un facteur déterminant, car toute erreur lors de la spécification du trafic désiré au banc d'essai, conduira immanquablement à un faux résultat. Si nous nous retrouvons dans pareil cas, il devient possible de passer plusieurs heures à essayer d'analyser des résultats qui ont été générés par une erreur de formulation du trafic.

2.4 Configuration

Une configuration est un ensemble de lignes de commandes (CLI) comprenant chacune des paramètres et des valeurs [8]. La configuration permet l'activation de un ou plusieurs services et applique des contraintes à ces services. Les configurations dépendent essentiellement du type d'équipement et du système

d'exploitation utilisé. Cependant, il existe des situations où la configuration dépend également de la topologie, du type de trafic et des services qui doivent cohabiter dans le routeur. Par ailleurs, il faut distinguer deux types de configurations : celles que l'on applique au routeur pour activer des services et celles que le routeur nous retourne (à notre demande) pour afficher les services activés ou des informations bien précises sur le routeur et les paquets qui le traversent.

2.5 Les services réseau configurés sur les routeurs

Bien que la fonction de base d'un routeur soit l'acheminement de paquets dans un délai aussi court que possible, il offre cependant bien d'autres services dans un réseau [10]. Toutes les possibilités qu'offre le routeur sont réalisables grâce au système d'exploitation (OS) qui évolue de version en version en proposant des services nouveaux répondant aux nouvelles attentes du marché.

La figure suivante représente le processus de configuration du routeur :

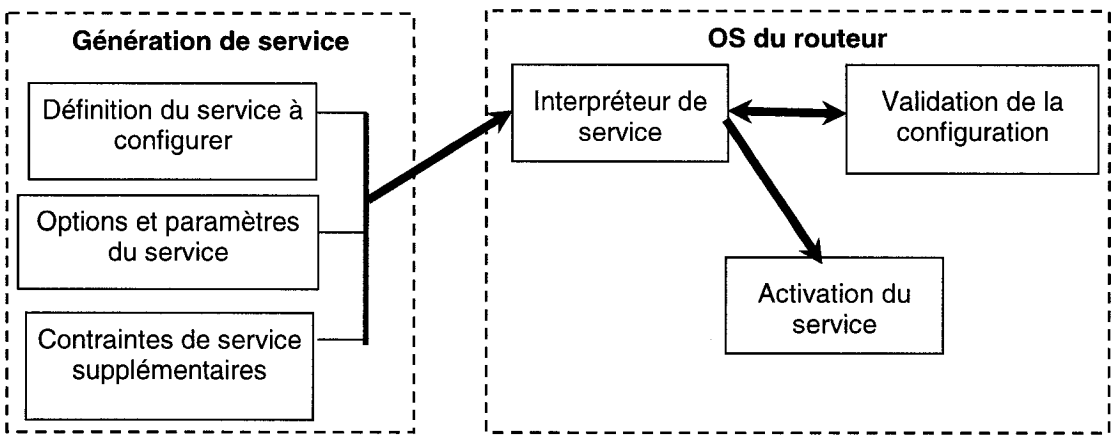


Figure 3. Processus de configuration de service

Exemple de certains services que peut offrir un routeur :

- filtrer des paquets selon des listes d'accès qui sont basées sur une adresse IP ou un protocole particulier,
- comptabiliser des paquets selon des paramètres donnés,
- réguler le trafic sur la base de certains critères,
- jouer un rôle sécuritaire en décelant des attaques et en retraçant les paquets suspects, etc.

Par défaut, les services sont disponibles dans l'OS mais non actifs, il faut donc les activer en utilisant les commandes adéquates dans une configuration. Les paramètres et valeurs de chaque commande représentent les fonctionnalités du service et les contraintes qui s'appliquent aux paquets entrants et sortants du routeur.

2.6 L'interaction de services

Deux ou plusieurs services fonctionnant au sein d'un même système peuvent avoir plusieurs formes d'interactions. D'une part, il y a l'interaction qui se fait de façon homogène et qui fait que deux ou plusieurs services peuvent utiliser les mêmes ressources sans que l'un empêche le bon fonctionnement de l'autre. D'autre part, il y a l'interaction visible (non souhaitée) à cause du disfonctionnement de l'un des services. En effet, dans ce dernier cas, chaque service est parfaitement fonctionnel mais, perd une ou plusieurs fonctionnalités dès qu'il se retrouve en présence des autres services. Par exemple, si un service échatillonne les paquets et un autre les filtre, il suffit de filter en premier pour que l'échantillonnage ne se fasse plus convenablement. Dans la suite de ce document, le terme interaction de services désignera ce dernier cas [21].

Les interactions de services dans les routeurs sont dues à plusieurs facteurs, entre autres la priorité, le conflit de ressources, un manque de mémoire, ou tout simplement une incompatibilité entre services.

2.7 Les tests de régression

Dans la littérature, les tests de régression sont un moyen de vérifier si un système, auquel des modifications ont été apportées, est toujours fonctionnel [30][29]. Par exemple, quand un bogue est résolu, il faut revérifier toutes les fonctionnalités qui n'étaient pas liées directement à ce bogue. Il faut également faire la même opération pour chaque nouvelle version du système.

Dans notre contexte, pour tester un routeur ou un service configuré sur un routeur, les testeurs ne peuvent reproduire tous les cas de topologies possibles, ils limitent alors l'équipement de test à trois ou quatre routeurs puis réutilisent la configuration des services à tester. Cette opération est souvent réalisée suite à une défaillance de service chez un client, ou suite à une modification de configuration du client. Le test de régression pour les routeurs est donc un processus de vérification partielle des fonctionnalités d'un service après une modification ou après la découverte d'un bogue et ce dans un banc d'essai réduit.

Avant d'arriver au stade d'élaboration d'un cas de test, il y a plusieurs étapes à franchir. En effet, il faut définir les fonctionnalités du système à tester, les limites de chaque test, la couverture à atteindre et la stratégie de test à adopter. Pour ce faire, il faut établir un document détaillant toutes ces informations puis implémenter tout ce qui est spécifié dans ce document. Idéalement, il faut organiser toute l'information (documentée ou implémentée) afin de la réutiliser au besoin. Les termes les plus communs pour désigner les documents de spécifications des tests et leur implémentation sont les plans de test.

2.8 Plan de test

Comme dans plusieurs autre domaines, le Plan de test est défini comme étant un document conçu par une équipe de test pour déterminer quels tests doivent être développés, quelle stratégie de test il faut adopter et quelles sont toutes les contraintes matérielles et logicielles qui devront s'appliquer aux diverses composantes du test [20].

Pour les routeurs, un plan de test se doit de définir le contexte dans lequel doit se dérouler le test, à savoir : la version d'OS, la topologie, la configuration, le trafic, le routage[18], et finalement, tous les aspects de la configuration qui doivent être testés en tenant compte des contraintes d'environnement et d'interaction de service.

2.9 Cas de test

Un cas de test est un ensemble de conditions sous lesquelles un usager détermine si une condition s'appliquant à l'unité sous test est partiellement ou entièrement satisfaite [27]. Parfois, il faut beaucoup de cas de test pour que la condition soit entièrement satisfaite. Un cas de test se caractérise par une entrée de données connue et un résultat prévisible établi avant le lancement du test. L'entrée connue devrait examiner une condition préalable (pré-condition) et le résultat prévisible vérifie la 'post-condition'. On parle de test unitaire pour désigner un cas de test, bien que cette appellation puisse causer des confusions.

2.10 Suite de tests

La suite de test est un ensemble de cas de test qui sont regroupés dans le but de valider un ou plusieurs aspects d'un système sous test [13]. La portée d'une suite de test change d'une organisation à l'autre. Il pourrait donc y avoir plusieurs suites de test pour un produit particulier. Dans la plupart des cas cependant, une

suite de test est un concept de niveau élevé, regroupant plusieurs dizaines ou de centaines de tests, ayant une cible commune à examiner.

2.11 En résumé

Les routeurs ont en commun un noyau formé d'un processeur et de mémoire (RAM, ROM, FLASH). Comme dans le cas d'un ordinateur personnel, le routeur démarre par un 'bootstrap' pour effectuer un autotest et lancer ensuite le système d'exploitation (OS). La dernière phase consiste à charger en mémoire la configuration qui contient l'ensemble des services activés et les contraintes liées aux fonctionnalités de chaque service. Certains routeurs sont de type modulaire, ils possèdent plusieurs cartes ayant différentes fonctions et pouvant supporter certains services offerts par l'OS. Les services dépendent essentiellement de la version d'OS, du type de carte et de son type d'unité de traitement. Ainsi pour configurer un routeur, il faut s'assurer que les services inclus dans cette configuration sont supportés par la version d'OS, le type de NIC et son unité de traitement, puis prendre compte de certaines considérations telles que l'interaction de service, les interfaces ou sous interfaces utilisées.

Le processus de test d'un routeur est semblable à celui de tout système, il faut établir un plan de test détaillant l'environnement de test, les contraintes à respecter, ainsi que toute considération d'ordre matériel ou logiciel. Suite au plan de test, il faut charger la configuration à tester sur le banc de test et finalement écrire les cas de test. Les résultats de test peuvent être récupérés manuellement en interagissant avec le routeur, ou stockés en mémoire pendant le déroulement du test, puis récupérés après sa fin.

Chapitre 3 Méthodologies actuelles de test

Les méthodologies de test dépendent essentiellement du champ d'application. En effet, le test d'un environnement matériel est différent d'un environnement logiciel qui lui est différent de celui d'un système embarqué. Cependant, certaines particularités du processus de test sont communes à tous les champs d'applications et à tous les environnements. Les chapitres suivants décrivent quelques approches de test.

3.1 Approche de test dans un contexte logiciel

Cette section décrit les stratégies de test applicables dans un contexte logiciel et les méthodologies qui s'y attachent.

3.1.1 Stratégie de test

Tester un logiciel revient à examiner sa structure pour évaluer la pertinence du code, et à tester ses fonctionnalités pour s'assurer qu'il répond bien aux exigences des requis; l'optimisation et la performance étant considérées comme parties intégrantes des requis [14][4].

Pour effectuer des tests structurels ou fonctionnels, on procède par simulation. Les outils et procédés de simulation sont nombreux, certains sont de bas niveau d'autres plus génériques. Les tests structurels et fonctionnels sont complémentaires et favorisent une augmentation du degré de confiance dans le modèle. Ainsi, un test structurel sert à examiner les branches d'un diagramme d'états, ou les différentes parties du code d'un programme logiciel, afin de le libérer de toutes les redondances et des parties du code non utilisables; cette optimisation réduit les branches et les boucles dans un logiciel, simplifiant ainsi les possibilités de tests. En principe, les tests structurels sont limités en nombre, on peut les quantifier selon la méthode de couverture utilisée. Cependant, ils

sont plus difficiles à réaliser quand il s'agit de modèles logiciels où plusieurs processus concurrents utilisant les mêmes ressources. Dans cd cas, il faut utiliser le test fonctionnel pour examiner certains aspects des parties non couvertes par le test structurel.

Un test fonctionnel [5] assure que le programme fonctionne physiquement comme prévu et que toutes les options exigées sont présentes. Il s'assure également que le programme se conforme aux normes concernant cet environnement; par exemple, dans un programme de Windows, la pression de F1 invoque l'aide.

Finalement, il y a les tests formels qui consistent à utiliser un langage formel basé sur une logique mathématique (ex : eLotos, Z) afin d'examiner la validité de certains aspects des spécifications d'un modèle. Le grand avantage de ces tests, c'est la rigueur avec laquelle les spécifications sont décrites puis testées. On peut ainsi vérifier les fonctionnalités du logiciel, son code et même sa performance, et ce de façon très rigoureuse.

3.1.2 Outils

Actuellement, il y a trois catégories d'outils pour tester les logiciels. La première concerne les tests sur le code source à proprement parler, quelque soit le langage (Java, C, C++). Son objectif étant bien entendu de détecter – de la manière la plus automatisée possible – les anomalies et de les corriger. Par exemple, le 'Rational Purify' de IBM est un outil de test pour le langage C/C++ (détection de corruption en mémoire) et Java, C/C++, Visual C++, C# et VB.Net (détection des fuites mémoire)[22].

La deuxième catégorie concerne les tests fonctionnels. Ces derniers reposent sur l'analyse des spécifications de tout ou d'une partie du logiciel, sans tenir compte de sa programmation proprement dite. La solution est testée de telle

sorte qu'une entrée dans une section donnée doit renvoyer une réponse conforme au document des requis. Cette phase concerne aussi bien les interfaces utilisateurs, les API, la gestion des bases de données, la sécurité, ainsi que le réseau. Les tests fonctionnels font partie intégrante des 'boîtes noires'. L'outil 'Quality Forge' de l'entreprise TestSmith permet l'automatisation des tests fonctionnels et de régression. Les tests concernent essentiellement les sites et les applications Web avec les langages Java et C++.

La catégorie de tests formels est basée sur des langages de spécification tels que le langage Z. Plusieurs outils ont été développés pour effectuer ce type de tests. On peut citer, LOTOS, ou COQ qui permettent non seulement de faire une validation de la spécification formelle d'un logiciel, mais également de lancer des tests déterministes ou aléatoires.

3.2 Approche de test dans un contexte matériel

Les sous-sections ci-dessous décrivent les stratégies de tests qui s'appliquent à un environnement matériel ainsi que les outils et méthodologies qui leurs sont associés.

3.2.1 Stratégie de test

Le matériel a ses propres particularités qui sont parfois très différentes du logiciel. Par exemple, un circuit qui a comme entrée un port de 8 bits et en sortie un port de 16 bits, peut être modélisé par un programme logiciel, mais les tests qui doivent s'appliquer au circuit et au modèle sont différents. Ainsi, avec un circuit, on peut appliquer directement des signaux, les modifier de façon aléatoire, ou encore, tester leurs propriétés [2].

Le test le plus efficace et en même temps le plus laborieux est celui de bas niveau. En effet, ce type de test s'applique au niveau des bus d'entrée du

système sous test, ce qui rend le nombre de possibilités de test proportionnel à une exponentielle du nombre de bits à l'entrée de ce système. Connaissant cette contrainte, ce type de test est donc efficace et même recommandé dans le cadre de systèmes ayant des bus d'entrée de moins de huit bits.

Un test aléatoire consiste à appliquer -sous certaines contraintes- des vecteurs de test à l'entrée d'un système matériel complexe et ce, dans le but d'améliorer la couverture. Le concept de test aléatoire peut être utilisé tant pour le logiciel que pour le matériel; cependant, il reste plus adéquat pour le matériel où les possibilités d'automatisme de test sont plus simples à réaliser et à gérer. En effet, tous les systèmes (matériel) numériques sont basés sur des diagrammes d'états et une horloge indiquant l'état du système au point de vue temporel; par conséquent, il est possible de contrôler le système à tout moment même en appliquant des vecteurs de test dont la valeur est aléatoire, et ceci est plus compliqué à réaliser quand il s'agit de faire des tests aléatoires sur du logiciel.

Les formes de données utilisées par les systèmes numériques sont, dans leurs bas niveaux, représentés par des bits, mais elles ne sont pas considérées toujours sous cette forme pendant les tests. En effet, un système peut avoir comme entrée une forme de donnée abstraite (ex : un paquet de données) ou ayant certains attributs (ex : nom, prénom, lieu de naissance). Les tests abstraits ou génériques sont plus simples à appliquer et n'exigent pas une connaissance approfondie du système sous test, il est donc utile de les employer dans un cadre de vérifications fonctionnelles basées sur des tests déterministes et possiblement sur des tests aléatoires.

3.2.2 Outils

Les langages VHDL et VERILOG sont les plus utilisés dans le domaine matériel pour modéliser des systèmes et générer des tests. Cependant, il y a des outils qui ont été développés dans le but d'automatiser les tests et améliorer leur

couverture. La compagnie Verisity a développé l'outil SpecmanElite basé sur le langage e, et qui comportant plusieurs modules, est capable de valider les spécifications d'un système, générer les tests non prévus explicitement par le plan de test et analyser les vecteurs de tests ainsi que les sorties du système.

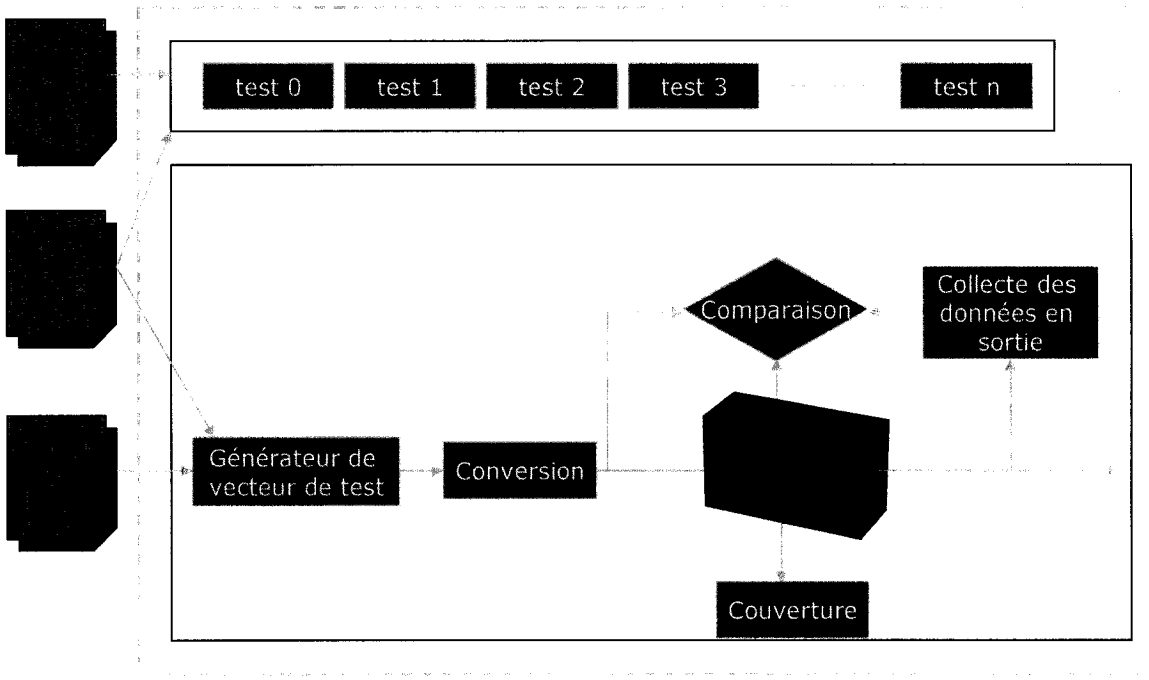


Figure 4. Schéma global de test automatique de SpecmanElite

Un tel processus de test permet une couverture beaucoup plus intéressante que celle offerte par les autres outils. En effet, ce concept a permis de réaliser des modules dont la couverture de test est tellement élevée qu'ils ont été considérés comme des composantes dépourvues d'erreurs et par conséquent sont réutilisables dans n'importe quel système. De telles composantes sont appelées des IP (Intellectual Property) et sont commercialisés pour être réutilisés comme des éléments d'un système dont les fonctionnalités sont garanties et dépourvues d'erreurs à un pourcentage très élevé.

Par ailleurs, il existe d'autres outils développés notamment par les compagnies Cadence, Synopsys, OpenOffice et MentorGraphics pour modéliser et tester des systèmes numériques. Ils ont la particularité de supporter tant les tests de bas niveau que ceux de haut niveau en créant une compatibilité entre les langages. Par exemple, un test de haut niveau, rédigé en langage C est traduit par l'outil en VHDL, puis converti en signaux bas niveau. Ces derniers sont appliqués au système sous test. Cependant, ces outils ne possèdent pas toutes les fonctionnalités offertes par SpecmanElite.

3.3 Similitude avec le test d'un routeur

Sachant que le routeur est un système embarqué, un test doit donc s'appliquer autant sur sa partie matérielle que sur sa partie logicielle [31][36]. Cependant, les tests ne se font pas sur la partie matérielle comme pour un système numérique quelconque, ils doivent garantir que les interfaces répondent aux exigences des spécifications fonctionnelles matérielles et logicielles (OS, services configurés). Par exemple, si on doit tester une carte ATM, cela ne signifie pas que les tests vont s'appliquer aux composantes au niveau signal, mais plutôt aux paquets entrants et sortants pour savoir si les spécifications de la carte sont respectées. On peut ainsi voir si la performance de la carte est respectée et si les services supportés par cette carte fonctionnent comme prévu[34].

Par ailleurs, plusieurs entreprises spécialisées dans les tests ont mis en place des processus de tests à partir d'une compilation de concepts théoriques de tests et de vérification, mais leur applicabilité sur les routeurs reste peu évidente. Comme les outils et méthodes 'génériques' ne sont pas directement utilisables dans le cadre d'un routeur, chaque constructeur de routeurs adopte alors une méthodologie de test répondant aux spécifications de ses produits[33] tout en s'inspirant des stratégies de tests et des concepts théoriques (académiques) et pratiques (industriel).

La similitude entre un routeur et les autres systèmes (matériel, logiciel) se situe plus sur le plan de la stratégie que sur le plan des outils. En effet, un routeur est souvent testé dans un environnement réel, il n'y a pas de simulateurs pour chaque type de routeur ou pour telle ou telle topologie, il faut donc utiliser les tests déterministes ou les tests aléatoire au sein d'outils développés pour évaluer les routeurs et leurs interfaces [3]. On ne peut représenter un routeur sous un diagramme d'état, vu la complexité de son système d'exploitation, du nombre de cartes qu'il peut supporter et des possibilités topologiques; cela rend le contrôle du routeur difficile à réaliser.

Les stratégies de tests de type déterministe, aléatoire ou pseudo-aléatoire, peuvent s'appliquer pour tester un routeur. Il s'agit de construire, comme pour les autres méthodologies, un plan de test et implémenter les cas de test en utilisant des outils basés essentiellement sur les scripts, tout en appliquant une ou plusieurs stratégies[35].

3.4 Cas d'un routeur

Comme décrit précédemment, le routeur possède un système d'exploitation 'OS' pouvant offrir, selon la version, un certain nombre de services. Tester un routeur consiste donc à vérifier le bon fonctionnement de ses interfaces et de ses services dans différents contextes topologiques et avec différents types de trafic moyennant des scripts de tests [9].

Dans un premier temps, il faut reconstituer l'environnement dans lequel les services devront être vérifiés; à savoir, le routeur, les interfaces, le trafic, la topologie et les diverses contraintes de service (voir figure qui suit).

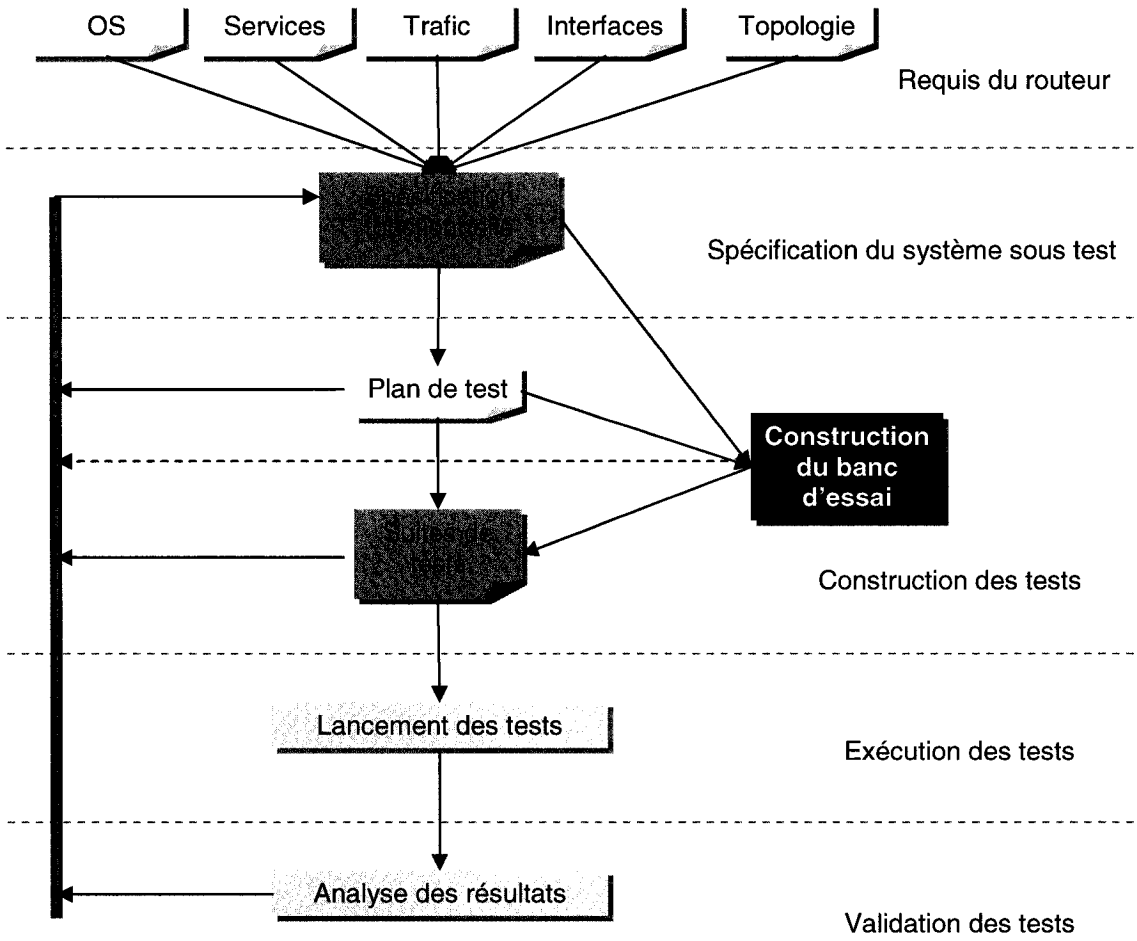


Figure 5. Processus de test d'un routeur

Le processus de test d'un routeur est itératif. Après avoir spécifié les fonctionnalités du système à tester, il faut établir un plan de test indiquant toutes les conditions nécessaires pour réaliser un ou plusieurs tests. Suite à cette phase, il faut procéder à la mise en place du banc d'essai. Cette étape peut paraître simple, car c'est essentiellement du matériel qui est mis en jeu, mais elle s'avère parfois difficile, lorsque les cartes (NIC) prévues dans le plan de test ne sont pas toujours compatibles aux services, ou ne sont pas supportées par le système d'exploitation (OS). Dans de tels cas, il faut faire un retour aux

spécifications fonctionnelles et reprendre la phase du plan de test. De même, l'étape de génération de suite de tests peut révéler l'inconsistance de la présence de plusieurs services sur la même carte, ou l'incompatibilité entre certains services. La dernière phase est le lancement des cas de test qui consiste à lancer le trafic et à valider les sorties du routeur.

Ce processus itératif permet d'améliorer les spécifications fonctionnelles de l'environnement de test et par conséquent le plan de test qui leur est associé, permettant ainsi un meilleur contrôle des fonctionnalités offertes par les services configurés sur le routeur.

3.4.1 Considérations de base

Le test de services multiples, comme dans les phases de régression et d'intégration de tests, est une tâche cruciale mais complexe. Une des difficultés de cette tâche réside dans le fait de fournir les cas et les suites de test qui sont adaptés à plusieurs services.

Même lorsque des services multiples sont configurés sur les mêmes interfaces, ils sont examinés séparément, c.-à-d. chaque service a ses propres suites de test. Ceci signifie que, pour un couple de services déployés sur les mêmes interfaces, des tests disjoints sont appliqués, et que les interactions parmi ces services ne peuvent être testées car les stimuli et la validation se trouvent dans des cas de test séparés [15]. Un bon test pour services multiples doit fournir la configuration, les stimuli (le trafic) et la validation pour tous les services sous test. Chaque service est décrit par des spécifications fonctionnelles et est examiné en utilisant les plans de test qui tiennent compte de ses spécifications.

3.4.2 Spécifications

Pour un nombre donné de services, il y a un nombre énorme de combinaisons parmi ces services qui peuvent être testés ensemble. Ces combinaisons sont si nombreuses qu'il n'est pas pratique de les tester toutes. Cependant, un choix parmi ces combinaisons doit nécessairement être fait, et il n'y a aucune manière claire de choisir les meilleures combinaisons. Par ailleurs, la plupart des services sont généralement déployés et examinés avec une ou plusieurs interfaces spécifiques.

Étant donné le nombre illimité de possibilités de combinaisons de version OS, de NIC et de services, la spécification fonctionnelle du système sous test en devient plus complexe. Par conséquent, la spécification fonctionnelle de chaque service est décrite de façon indépendante. Elle comporte toutes les fonctionnalités et contraintes (connues) relatives au service ainsi que l'environnement qui le supporte (OS, NIC, Interface, sous-interface, etc). La collecte d'information pour réaliser un test doit se faire sur plusieurs fronts : le document de spécification de chaque service, les matrices décrivant quel service est supporté dans quel environnement et finalement le type de trafic et de routage possiblement applicable au contexte de test.

3.4.3 Plan de test

Le plan de test *(Référence IEEE STD 829)* représente l'architecture décrivant chaque entité intervenant dans le test. Par entité on entend, la topologie, le type de trafic, le routage, l'équipement, les services à configurer et la version d'OS. Il comporte également toute considération ou contrainte liée à la coexistence de services et à l'interaction de leurs fonctionnalités [28][15][1].

Le plan de test commence toujours par énoncer le contexte de test, il s'agit donc de décrire quel type d'équipement doit être mis en place en précisant le modèle,

les performances et les spécifications utiles pour le test. Ainsi il doit décrire le type du routeur, son châssis, son processeur, les NIC et la version d'OS utilisées, quels services doivent être configurés sur quelles NIC, et finalement quel type de trafic doit traverser cet équipement. Le plan de test dresse un schéma de la disposition de l'équipement qui doit former une topologie. Sachant que l'on œuvre dans un domaine de test de régression, la topologie est souvent limitée à 2 ou 3 routeurs.

Après le contexte de test, vient le tour des suites de tests. Celles-ci sont délimitées par une stratégie de test et un choix délibéré d'un certain nombre d'actions à exécuter dans le but de vérifier si les services et les routeurs se comportent comme prévu. Le comportement prévu est alors examiné par le processus de validation dans chaque cas de test. Dans le plan de test on retrouve la description de plusieurs cas de test mais leur regroupement en suites de tests se fait selon les comportements que l'on désire vérifier.

Le dernier volet du plan de test concerne essentiellement l'analyse et la documentation des résultats obtenus durant les tests. Chaque résultat de test est collecté et analysé pour en déterminer la pertinence puis pour définir les étapes et décisions à entreprendre par la suite. Ce processus nous amène souvent à revoir les spécifications fonctionnelles de notre système sous test et d'apporter des modifications au plan de test lui-même.

3.4.4 Suite de test

Par définition, une suite de test valide le comportement d'un produit. Il peut y avoir plusieurs suites de test pour un produit. Cependant, dans la plupart des cas, une suite de test est un concept de niveau élevé, groupant ensemble des centaines ou des milliers de tests reliés par un ou plusieurs objectifs.

Pour le routeur, une suite de tests est destinée à valider les fonctionnalités des services configurés sur une ou plusieurs cartes (ou interfaces). Chaque suite de test se distingue d'une autre par l'aspect à vérifier. Par exemple, si on choisit de vérifier la performance, alors la suite de test comportera ces cas de test relatif à la performance de chacun des services configurés, ou des NIC qui les supportent. Une suite de test n'existe pas physiquement, mais on donne en général un nom significatif de l'objectif recherché et on établit un lien symbolique avec les cas de test qui doivent être exécutés.

3.4.5 Cas de test

Le cas de test est une implémentation physique des idées et objectifs énoncés dans le plan de test. Ainsi, on y retrouve tous les liens et connexions vers les APIs (certaines APIs sont utilisées pour générer le trafic, d'autres pour contrôler l'environnement de test), la désignation de chaque élément à tester du système et finalement, les paramètres et valeurs des diverses entités du test. Le cas de test est constitué de :

- Déclaration des variables et procédures globales.
- Définition de la topologie du banc de test et de l'interface ou NIC à tester.
- Mise à zéro des compteurs des routeurs du banc d'essai.
- Configuration des services sur le routeur.
- Configuration et activation du trafic.
- Validation du test.
- Enregistrement des résultats de validation dans les fichiers log pour des fins d'analyse.

3.4.6 Analyse des résultats

Bien que la validation soit intégrée au cas de test, les résultats de tests ne sont pas toujours évidents à interpréter. En effet, souvent la validation permet

d'indiquer ce qui devrait se passer mais, n'indique pas les causes de la défaillance s'il y a lieu. Il faut donc passer à travers toutes les traces enregistrées lors du test et analyser l'ensemble des données pour aboutir à une conclusion. Parfois l'analyse permet de conclure que le test a été mal construit ou que le routeur a été mal configuré.

Il arrive que la validation soit globale pour plusieurs fonctionnalités ou plusieurs interfaces du routeur, dans tel cas, une attention particulière doit être portée à l'interprétation des résultats de test. En effet, ces derniers pourraient indiquer un échec global du test, bien que certaines fonctionnalités ou interfaces aient eu un comportement tout à fait prévisible. Il faut donc tenir compte de ces aspects au moment de la validation afin d'éviter des interprétations erronées et par conséquent une mauvaise compréhension des résultats des tests.

La stabilité du système d'exploitation du routeur représente un aspect non négligeable dans les tests. En effet, certains résultats de test révèlent un disfonctionnement partiel (ex : quelques paquets sont perdus de façon arbitraire), il faut donc se pencher sur la stabilité de l'OS pour s'assurer que tous les services configurés sont bien supportés par le routeur et son OS.

La phase d'analyse permet de remettre en cause toutes les étapes qui la précèdent, à savoir, la spécification, le plan de test, le banc d'essai et les tests eux-mêmes.

3.5 En résumé

Les concepts théoriques relatifs au test sont utilisés dans plusieurs domaines, et ce, quelque soit la nature du système sous test. Cependant, la façon d'utiliser et d'implémenter les tests diffère d'un système à l'autre et d'un organisme à l'autre.

Le domaine logiciel se caractérise par des tests concernant la pertinence du code et la cohésion, alors que le domaine matériel se caractérise par des tests de performance. Quel que soit le domaine, les stratégies de tests (déterministes, aléatoires, pseudo-aléatoiresetc.) sont généralement utilisées dans le but de détecter des disfonctionnements ou anomalies dans les systèmes sous test. Cependant, en microélectronique, on utilise de plus en plus de méthodologies basées sur l'abstraction des tests et la réutilisation.

Le domaine de test des routeurs est encore inconnu du public, car il se pratique la plupart du temps en entreprise, par le biais d'outils et de méthodes très spécifiques. Certaines entreprises se sont hasardées à offrir des outils permettant d'offrir des couvertures de tests pour des services réseau, mais ces outils restent très génériques et leur portée est limitée aux principes généraux dans les réseaux (protocoles, trafic, bande passante, etc.). Dès qu'on parle d'interaction de service, ou de tests simultanés de plusieurs services, ces outils ne sont plus fiables.

Le contexte de test, le plan de test, les cas de test, le banc d'essai et la validation, restent définitivement les ingrédients nécessaires et suffisants pour réaliser une bonne couverture de test des services configurés sur un routeur. La pertinence et la modularité des composantes de chacune des ces étapes, reste le moyen le plus sûr de garantir une meilleure réutilisation des tests et une diminution de leurs coûts.

Chapitre 4 Réutilisation des tests : ScriptMaker

Dans un réseau, l'état du routeur est en perpétuel changement et ce, de façon très rapide; les paquets qui le traversent, les services qui y sont déployés et le changement topologique en sont les principales causes. Par conséquent, le contrôle de son état devient très complexe et difficilement réalisable. Néanmoins, les services configurés sur un routeur sont testés de façon indépendante; leurs fonctionnalités sont évaluées grâce à des méthodes et des outils plus ou moins spécifiques; ce qui conduit forcément à une augmentation des coûts de conception et de développement des tests.

Ce chapitre décrit une méthodologie de construction de scripts de tests à partir d'un modèle d'abstraction de configuration de service, et de l'environnement de test. Il décrit également une approche qui fournit les modèles génériques et réutilisables de configuration qui sont adaptés aux changements de l'environnement et des états du banc d'essai. Cela signifie que, lorsque les changements se produisent dans l'environnement ou les paramètres de banc d'essai, les configurations de service de réseau peuvent être automatiquement mises à jour; ainsi que le trafic et la validation. L'abstraction de l'information du contexte de test et des éléments de configuration améliore également la capacité de réutilisation et la modularité des cas de test, ce qui a comme conséquence la réduction de la complexité de génération des cas de test et des coûts associés.

Le générateur de scripts de test développé à cette fin, appelé 'ScriptMaker', est un outil basé sur un concept d'abstraction de configuration de routeur appelé le Meta-CLI. Le modèle de Meta-CLI a été mis en application dans le ScriptMaker dans le but de favoriser la détection de l'interaction des services pendant les phases de tests de régression.

Les sections qui suivent dans ce chapitre présentent une vue globale du modèle Meta-CLI, une description des aspects et contraintes d'environnement de test, la structuration des éléments de test, et finalement l'outil ScriptMaker permettant l'automatisation et la réutilisation des cas de test dans le but d'examiner l'interaction de services.

Le ScriptMaker est une solution englobant les concepts d'abstraction de services et d'environnement de test, de réutilisation des cas de test et d'automatisation de la génération des scripts de test.

4.1 Le modèle Meta-CLI

Ce modèle propose une approche basée sur l'abstraction du CLI (Command Line Interface) pour la construction de structures génériques pouvant être réutilisées afin de produire des configurations de service. Ce modèle est un formalisme qui peut réduire la complexité et augmente l'efficacité de création des configurations de service.

Le modèle Meta-CLI traduit l'information de configuration CLI des services configurés sur le routeur en structures arborescentes. Pour déployer un service sur un routeur ou une interface, certaines commandes de configuration et paramètres doivent être créés ou modifiés. La structure arborescente des services inclut les commandes, les modes de configuration, les paramètres et leur contexte, et finalement l'environnement valide pour chaque paramètre.

La figure qui suit décrit une structure générique du modèle de configuration de service basé sur le Meta-CLI.

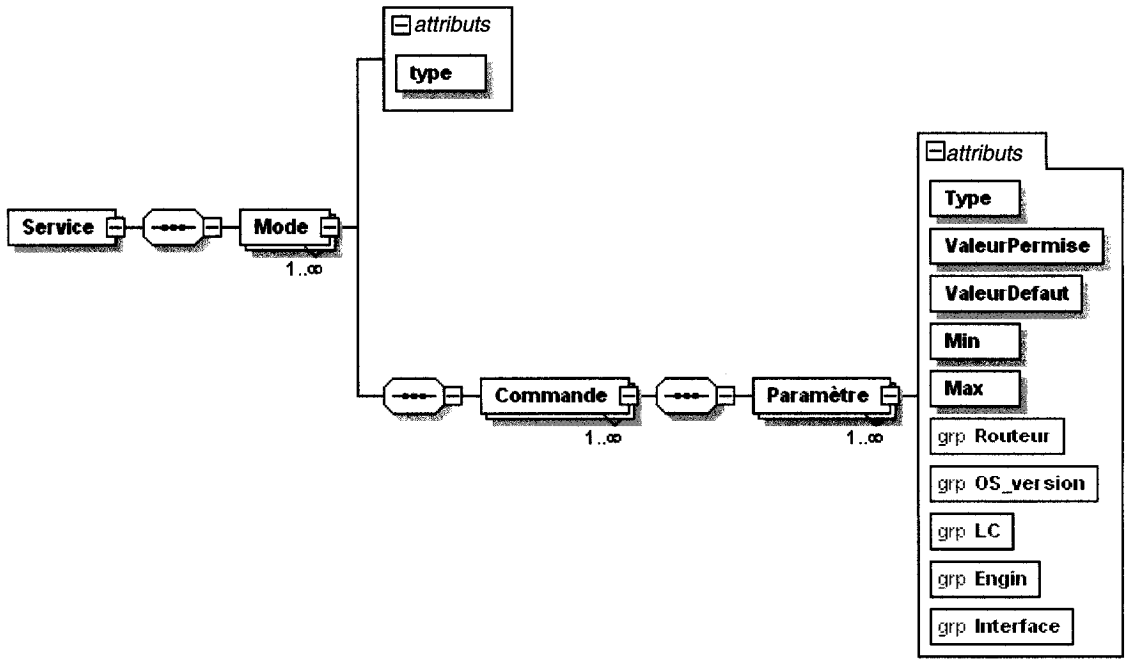


Figure 6. Structure générique de service basée sur le Meta-CLI

Le modèle Meta-CLI décrit les contraintes de configuration sur les nœuds de l'arbre, ces contraintes sont employées pour produire des ensembles de règles pour la validation des services configurés sur un routeur. La validation du service employant ces règles assure l'intégrité de l'information de configuration.

La racine de la structure générique représente le nom du service, il s'agit d'une chaîne de caractère symbolisant le service ou la fonctionnalité du routeur à configurer. Chaque service possède une ou plusieurs commandes qui permettent de l'activer dans le routeur et qui sont stockées dans des nœuds enfants de la racine. Sachant que le routeur possède plusieurs modes de configuration (usager, privilégié, etc.), chaque commande peut posséder différents paramètres selon le mode de configuration. Finalement, chaque commande dans un mode donné possède plusieurs paramètres ayant chacun des attributs (valeurs, valeurs limites, contraintes d'environnementetc.)

Il existe plusieurs types de dépendances entre les divers paramètres et commandes des services déployés sur les routeurs. Par exemple, deux interfaces d'un routeur, sur lesquelles un même type de service a été configuré, peuvent subir les mêmes contraintes. Ces contraintes se matérialisent dans les attributs des paramètres. Ces paramètres constituent donc un point de rencontre des dépendances entre les services ou tout simplement entre les paramètres d'un même service.

4.2 Abstraction par rapport à l'environnement de test

L'environnement de test est constitué de l'OS, du type de routeur, des NIC et des interfaces. Chaque paramètre d'une commande, nécessaire pour configurer un service, possède des attributs ou un groupe d'attributs pouvant identifier les environnements dans lesquels il est valide. Ainsi, on peut retrouver à partir des données d'environnement tous les services, commandes, modes et paramètres qui peuvent s'appliquer à une configuration.

À la base, le modèle Meta-CLI proposait une abstraction par rapport aux valeurs et au type (service, commande, etc.). Le modèle étendu du Meta-CLI offre une extension qui consiste en une abstraction de la configuration par rapport à l'environnement de test. Par exemple, en modifiant un aspect de l'environnement, on peut parcourir l'arbre Meta-CLI et retracer les éléments de la configuration à modifier ou à considérer. On peut aussi déterminer quels services sont supportés dans un environnement donné. Le modèle le fait en vérifiant la valeur de la contrainte d'environnement.

L'exemple suivant montre un arbre représentant un service générique dont les paramètres varient selon le changement d'environnement.

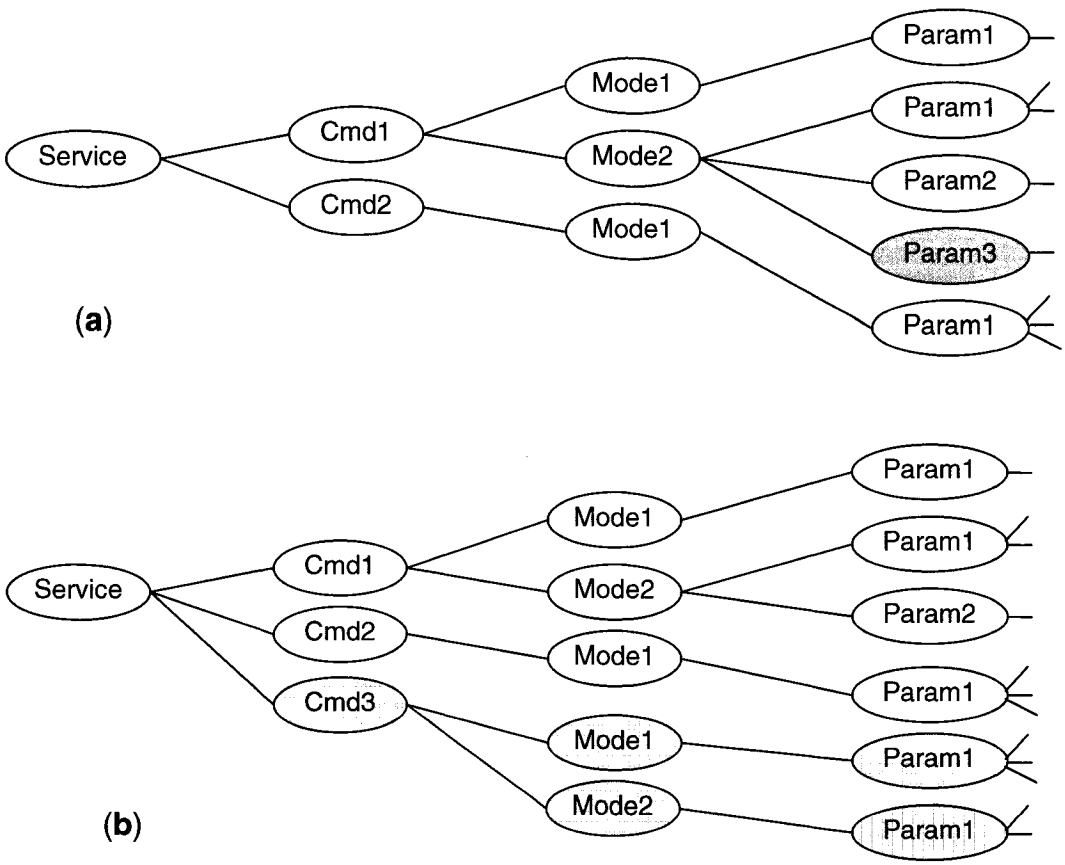


Figure 7. Schéma d'un service dans deux environnements différents

La fig 7(a) montre que le service possède seulement deux commandes, alors que dans la fig 7(b) il y a 3 commandes. D'autre part, dans (a) le service possède trois paramètres en mode 2 de la commande 1, ce qui n'est pas le cas en (b). Cette différence est due au fait que l'environnement ait changé en passant de (a) vers (b). Il se trouve que le paramètre qui était supporté dans l'environnement en (a) ne l'est plus en (b), et que le service possède plus de commande dans l'environnement en (b), ce qui explique la présence de la commande cmd3 dans l'arbre en (b).

L'abstraction des paramètres CLI de la configuration par rapport aux valeurs, aux types, et à l'environnement permet une meilleure réutilisation des scripts de test.

En effet, dans chaque script de test, on retrouve une partie liée à la configuration de services, cela nous amène donc à une situation où on n'utilise pas d'abstraction, et où on doit vérifier la forme et le contenu de chaque commande et valider son existence dans un environnement donné. L'arbre de Meta-CLI valide donc les paramètres de configuration dans les cas de test et permet la réutilisation de façon systématique.

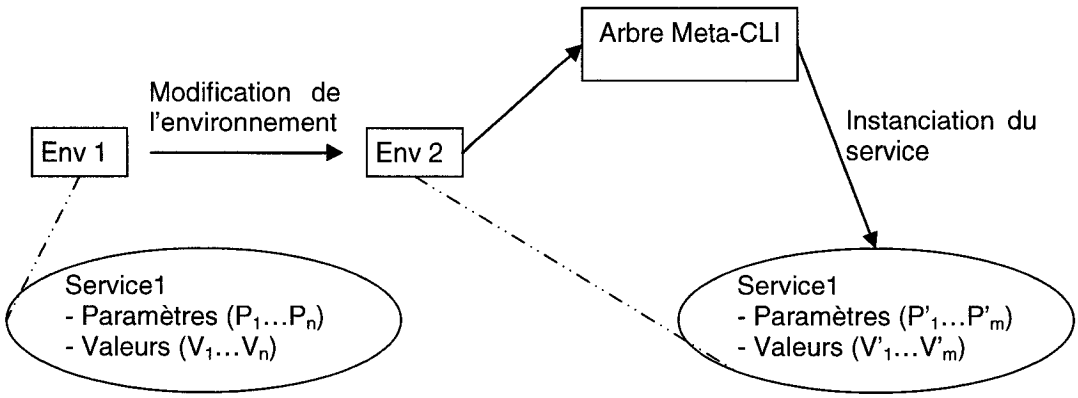


Figure 8. Impact de l'environnement sur les paramètres de service

4.3 Structuration des scripts de test

Pendant la phase de test de régression, on s'assure que la modification de l'OS (nouvelle version, nouvelles fonctionnalités, nouveau service, etc.) ne perturbe pas les fonctionnalités des services existants. Généralement, on fait des associations de services pour examiner de façon conjointe leurs fonctionnalités; cela peut se traduire par un nombre exponentiel de tests possibles en fonction du nombre de fonctionnalités.

Par ailleurs, les cas de test sont normalement écrits de façon manuelle avec des langages de script et ils sont souvent mal commentés. Par conséquent, leur compréhension et leur réutilisation sont très laborieuses. Il arrive qu'un usager

lise le test réalisé par une autre personne, et ne comprenant pas son contenu, réécrive un autre test. Cette redondance implique une perte de temps considérable. Il est donc utile de construire une structure standard d'un cas de test pour uniformiser son utilisation et permettre ainsi une réutilisation plus facile de son contenu.

Une approche possible pour structurer les cas de test consiste à d'abord décomposer le script de test en plusieurs modules pouvant interagir de façon cohérente, puis à créer des structures externes pouvant abriter l'information de configuration, de trafic, de routage et de validation. Le script de test devient alors lui-même une structure dans laquelle s'intégreront ces différents modules (voir figure suivante).

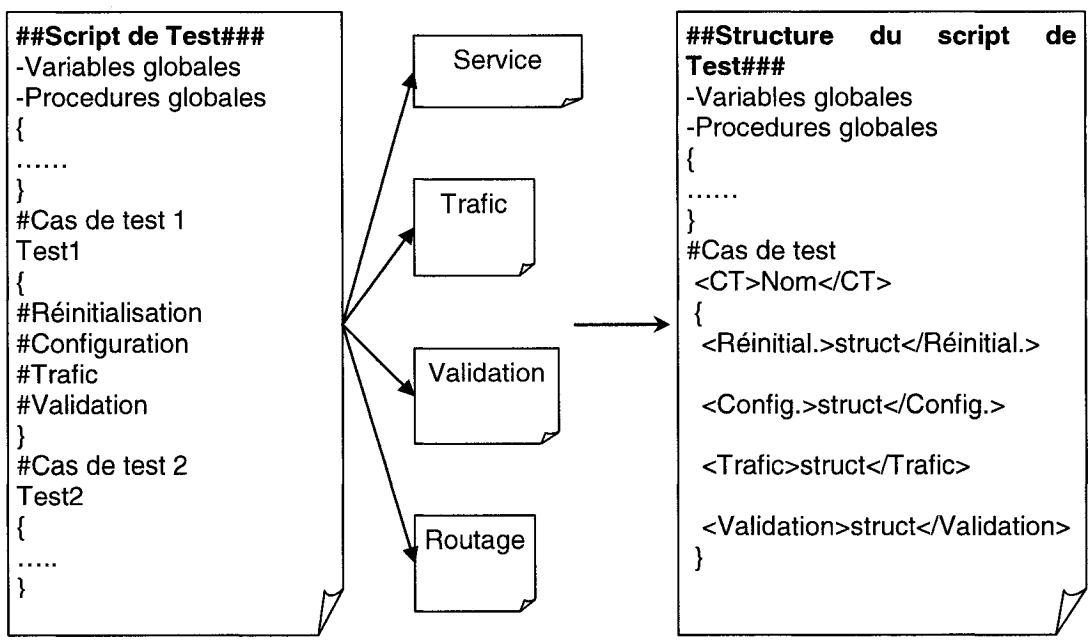


Figure 9. Décomposition du script de test en modules réutilisables

4.3.1 Décomposition du script de test

Pour arriver à décomposer le script de test, il faut distinguer l'information qui peut faire l'objet d'une réutilisation, de l'information statique. Ainsi, les variables globales et les procédures globales font parti des éléments qui sont spécifiques à chaque test. Elles peuvent donc être considérées comme de l'information statique du test. Par contre, la configuration, le trafic et le routage sont des éléments qui peuvent être déterminés à partir des paramètres de l'environnement, ou des choix de l'usager pendant l'élaboration du test. La validation est une composante qui se trouve à mi-chemin entre le statique et le réutilisable. En effet, il est possible de valider la configuration, le trafic, et le routage, comme il est possible de valider la topologie, ou une interface d'une NIC du routeur, ou encore la version de l'OS; il faut donc considérer les deux possibilités lors de la création des cas de test.

4.3.1.1 Service

Pour permettre de tester plusieurs services dans un même script de test, chaque service doit avoir, selon le contexte de test, un certain nombre de paramètres CLI et de valeurs réutilisables. Cette réutilisation permet de configurer simultanément plusieurs services sur le banc d'essai. Le service est donc une composante de la structure du script de test et dont la validité des paramètres et valeurs est réalisée à travers le Meta-CLI.

4.3.1.2 Trafic

Il existe plusieurs types de trafic (IP, MPLS,etc.). Il faut par conséquent séparer le script de test du trafic afin d'en préserver le caractère générique. Ainsi, selon le choix de l'usager, un type de trafic est généré moyennant des paramètres (nombre de paquets, taille des paquets, etc). L'environnement de test est

également pris en considération, car le trafic dépend aussi de l'équipement qui le génère.

4.3.1.3 Routage

Pour router les paquets arrivant vers un routeur ou une interface d'un routeur, nous considérons trois facteurs : l'adresse de destination, le protocole de routage, et la table de routage. Chaque élément du routage représente une composante réutilisable pour la construction des tests. On peut ainsi utiliser un protocole Ospf, avec une table de routage correspondante et l'adresse réseau se trouvant sur les paquets entrants. Toutes ces données sont représentées par des paramètres liés au routage, et leur exécution se fait par des procédures propres au cas de test ou globales dans le script de test.

4.3.1.4 Validation

Valider les tests consiste à vérifier le prévisible avec le réel. Dans ce sens, la validation pourra être dans certains cas statique, donc spécifique à l'environnement ou réutilisable quand il s'agit d'examiner un service, un routage ou un trafic donné. Dans les deux cas, la validation nécessite la définition d'un certain nombre de paramètres qui sont utilisés comme arguments dans des procédures globales.

4.4 La structure du script de test

Dans la section précédente, le script de test est décomposé en plusieurs modules. Cette décomposition a permis de tracer le squelette de la structure du script de test où chaque module pourra s'insérer pour construire, selon les besoins des testeurs, des instances de test.

La figure suivante décrit de façon globale la structure du script de test et les composantes de chacun des modules décrits précédemment; il ne s'agit pas d'une structure fixe, mais bien d'un ensemble de modules dont les paramètres peuvent évoluer avec le temps.

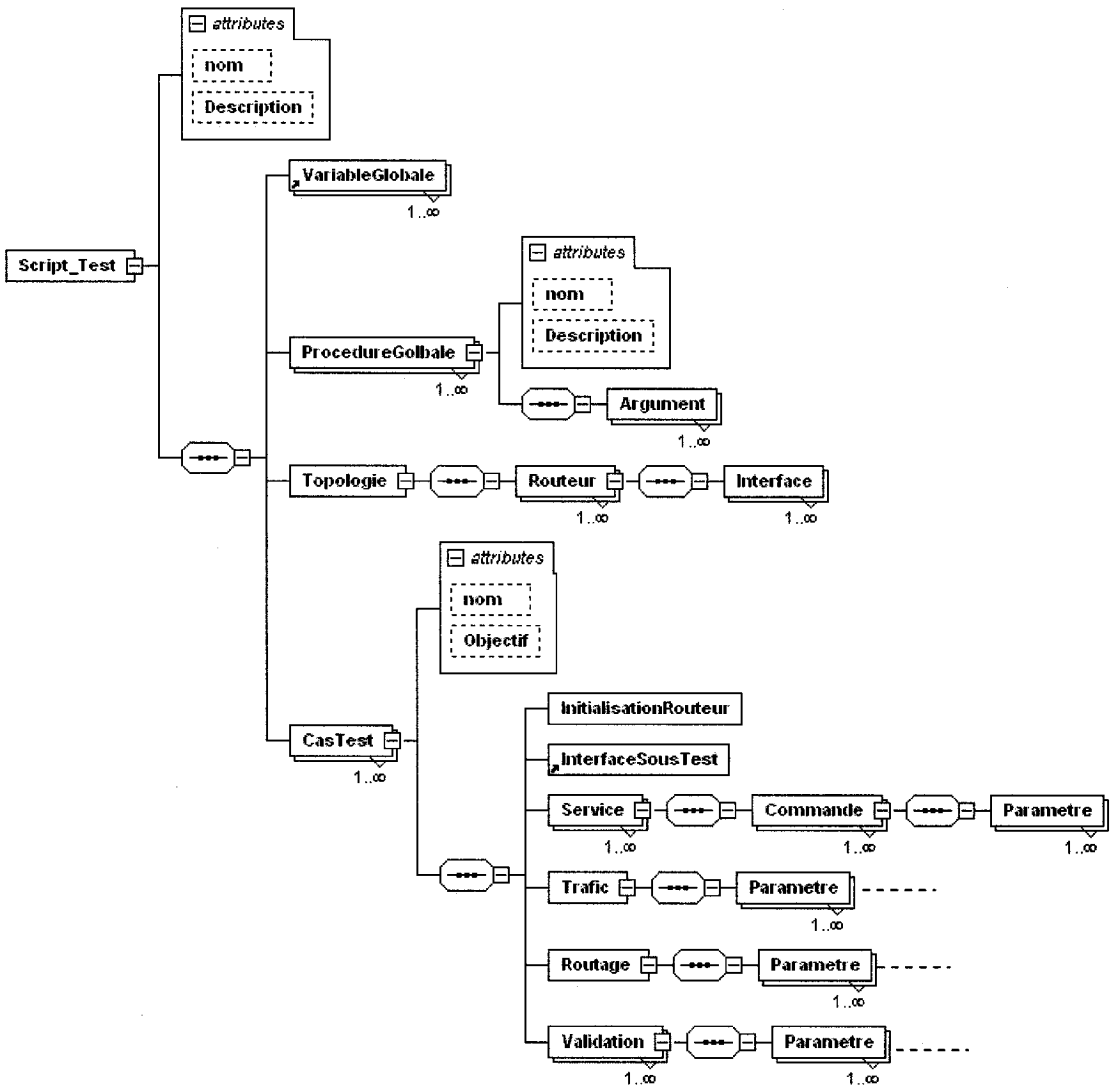


Figure 10. Schéma global de la structure d'un script de test générique

La racine de cette structure est le script de test dont les attributs décrivent le nom de fichier du script, et le but de ce fichier (objectifs, plan de test utilisé, suite de test, cas de test, etc.).

Les variables globales sont réutilisées dans les procédures globales et dans les modules du script de test. On peut citer à titre d'exemple, le nom du routeur sous test, le nom du générateur de trafic, etc.

Les procédures globales correspondent à tous les traitements qui doivent s'exécuter durant la phase de test. Qu'il s'agisse d'une ouverture de connexion vers le banc de test, ou d'une communication avec l'API du générateur de trafic, ces procédures sont appelées par leur nom et reçoivent un certain nombre d'arguments qui serviront à leur exécution. On parle de procédures globales, car souvent dans les scripts de test on retrouve plusieurs cas de test, voire plusieurs suites de tests; dans ce cas, plusieurs cas de test peuvent réutiliser les mêmes procédures en faisant des appels avec les arguments nécessaires. L'implémentation des procédures n'est pas forcément faite dans les scripts de tests. Il est aussi possible de les mettre dans des fichiers séparés ou même dans un réseau séparé.

Le module topologie est très important dans le script de test, il permet de préciser le type d'équipement présent dans le banc de test, et les interfaces qui seront soumises aux tests. Généralement, une variable globale est associée au routeur sous test.

Dans la structure, on ne voit pas de module concernant la suite de test, pour la simple raison qu'une suite de test n'est rien d'autre qu'un lien symbolique pour représenter un ensemble de cas de test, par conséquent, son influence est minime pour la structure, même si elle peut être significative pour le test lui-même.

Le module de cas de test est lui-même composé de sous-modules. Il s'agit de tous les éléments qu'on retrouve dans un cas de test spécifique au routeur. Chacun de ces sous-modules est décrit dans la section précédente.

Cette structure de script de test sert essentiellement à regrouper les différents éléments constituant un test (fig 12) et qui sont dépendants du contexte de test. Ce regroupement se fait de façon automatique à travers l'outil ScriptMaker.

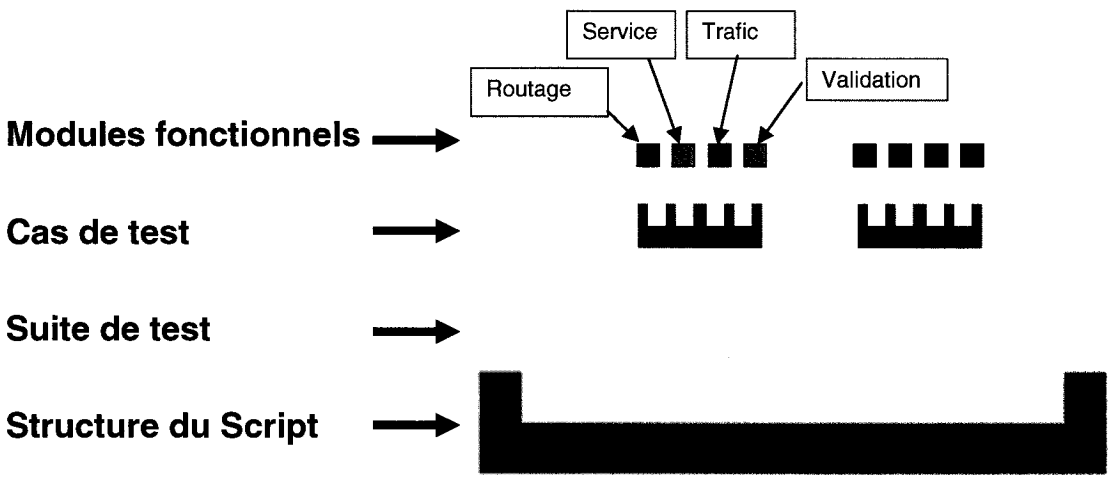


Figure 11. Instanciation d'une structure de script de test

La construction du script de test commence par une collecte des modules fonctionnels qui sont associés à l'environnement de test (banc d'essai, OS, etc.), la place de chaque module est libellée dans la structure de cas de test, il est ainsi plus simple d'insérer les modules sélectionnés. Les cas de test eux, sont regroupés sous forme de suites de tests, cette association est généralement symbolique, même si la structure du script de test ne l'identifie pas de façon claire. Finalement, la structure du script de test abrite l'ensemble des cas de test qui sont associés à l'environnement mis en place pour réaliser les tests.

Les autres composantes du script de test (variables globales, procédures globales) ne figurent pas parmi les modules fonctionnels pour la simple raison que ces derniers sont les seuls qui dépendent de l'environnement de test.

4.5 Le générateur de Scripts : ScriptMaker

L'automatisation de certains aspects de la construction des scripts de tests, et le gain que procure la réutilisation des données de test, sont la raison d'être du ScriptMaker. En effet, cet outil permet au testeur de choisir un environnement de test (routeur, OS, NIC, interfaces,etc.) et des tests à réaliser. Il traite cette information comme entrée afin d'aller chercher toutes les données nécessaires à la construction du script de test, pour finalement générer un script de test sous forme de fichier (fichier texte et XML). La structure du script de test décrite dans la section précédente représente le squelette qui est instancié par le ScriptMaker pour réaliser les fichiers scripts.

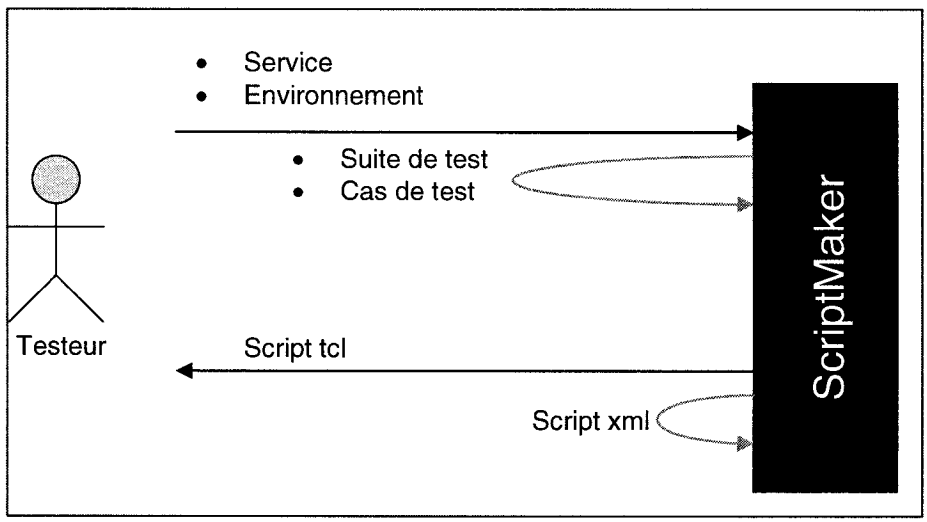


Figure 12. Entrée/Sortie de l'application ScriptMaker

4.5.1 *Architecture du ScriptMaker*

L'outil ScriptMaker est une application composée de plusieurs modules complémentaires, chaque module réalise un traitement particulier pour aboutir au script de test final. Il comprend donc un module de sélection des éléments de test, un second pour la gestion des services, un troisième pour l'assemblage et la création des scripts de tests. Il y a également un module pour implémenter l'interface usager et gérer l'échange d'information entre les composantes de l'application (voir la figure qui suit)

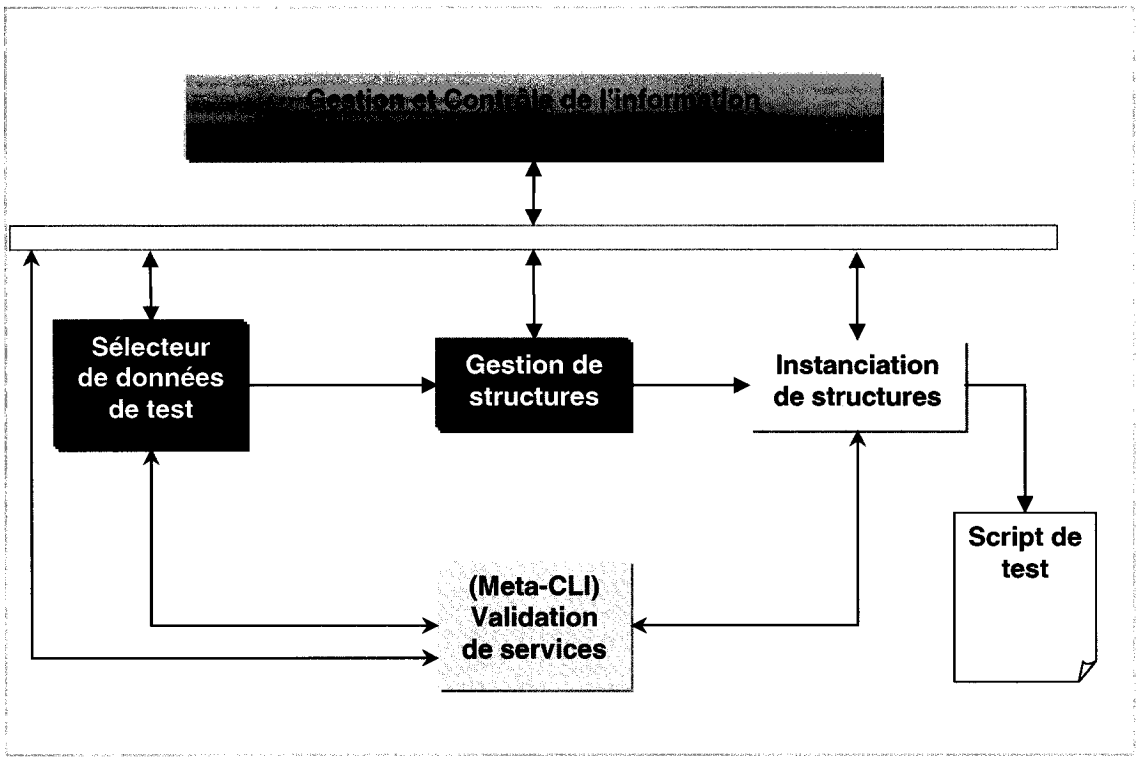


Figure 13. Architecture de l'application ScriptMaker

L'application ScriptMaker utilise plusieurs sources de données. Certaines d'entre elles serviront à proposer au testeur les différentes possibilités qui s'offrent à lui. D'autres serviront à mémoriser des données relatives à la configuration, au routage, ou à la validation, etc. Les structures et les liens de ces données avec

l'environnement de test dépendent de la nature des modules concernés. Par exemple, la validité des services dans un environnement de test est gérée par une base de données relationnelle, alors que le trafic a une structure en xml qui s'insère directement dans celle du script de test.

4.5.1.1 Module de gestion et de contrôle de l'information

L'usager de l'application ScriptMaker interagit avec une interface graphique qui lui permet de sélectionner l'environnement de test, les services à tester et le type de test qu'il désire effectuer. Cette opération est assurée par le module de gestion et de contrôle. Celui-ci, se charge ensuite de communiquer le choix de l'usager au module concerné. Le choix de l'usager se fait à travers une base de données contenant tous les services, les différents types d'environnements possibles et les différents cas de test disponibles.

Ainsi l'usager doit choisir un système d'exploitation, un type de routeur, le type de NIC, l'interface sous test, et finalement le ou les services à configurer.

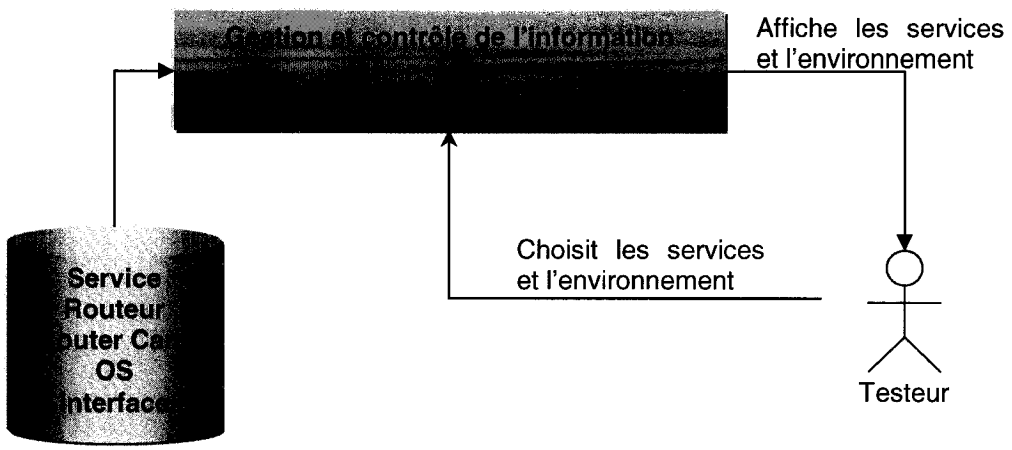


Figure 14. Interaction entre l'usager et le ScriptMaker

Selon les services et l'environnement choisis par l'usager, le module de gestion communique à l'usager les possibilités de cas de test ou de suites de tests possibles. Une fois que l'usager ait réalisé son choix, les paramètres d'environnement, de service et de tests sont transmis au module de sélection.

Dès que le module de Sélection a terminé de regrouper les différents éléments, le module de Gestion actionne le module Meta-CLI pour valider l'information de configuration, puis appelle le module de structuration qui met en place les différentes composantes structurelles du Script de Test, et finalement, active le module d'instanciation de structures qui est responsable de la génération du script de test.

Le module de Gestion et de contrôle de données a également un rôle de gestionnaire de la base de données, il permet, en mode administrateur, d'enrichir la base de données avec de nouveaux cas de test, nouveaux environnements, de nouveaux services, etc.

4.5.1.2 Module de Sélection de données de test

Ce module reçoit sous forme d'arguments, l'OS, la liste des routeurs, les interfaces et les services, choisis par le usager pour former le contexte de test. À partir de ces arguments et d'un ensemble de données emmagasinées dans la base d'information, le module Sélecteur détermine quelles composantes sont valides pour la construction du script de test désiré et les transmets au module de gestion de structures à travers le module de gestion.

Le module Sélecteur est composé à son tour de 4 sous-modules : le contrôleur, le lien, la validation et l'organisation de cas de test. Les sous-modules utilisent des structures génériques pour choisir, valider et organiser les composantes du script de test.

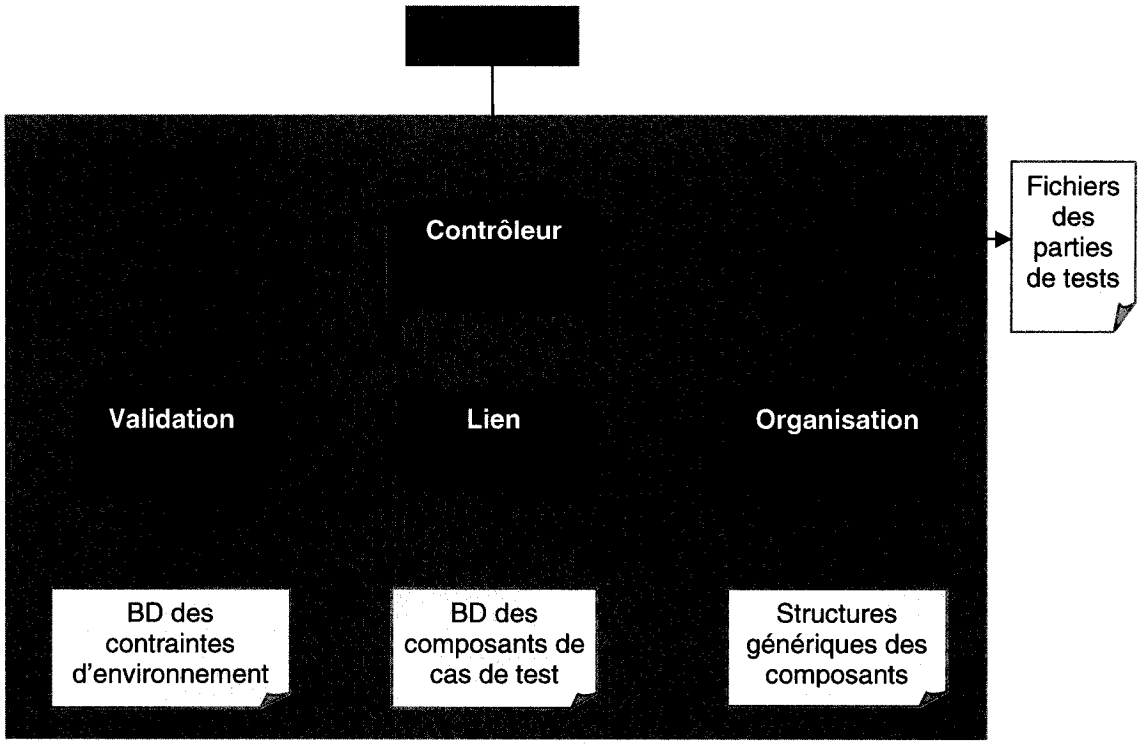


Figure 15. Architecture du module Sélecteur

Le sous-module Contrôleur du Sélecteur reçoit sous forme d'arguments le choix de l'usager (service, environnement, tests). Il valide ces arguments puis les transmet au sous-module lien. Un double vecteur a été utilisé pour contenir tous les arguments.

Environnement	**Service1**	**Service2**	**.....Service**
OS	$Test_1S_1$	$Test_1S_2$	$Test_1S_n$
Routeur	$Test_2S_1$	$Test_2S_2$	$Test_2S_n$
NIC	.....	....	....
	$Test_xS_1$	$Test_yS_2$	$Test_zS_n$

Le premier élément du double vecteur contient les informations sur l'environnement de test, puis les autres éléments décrivent les services configurés et les tests qui leur sont associés et qui ont été choisis par l'usager.

Le sous-module Validation reçoit les arguments du sous-module Contrôle, puis vérifie pour chaque couple de services la possibilité de leur cohabitation. En effet, bien que supportés de façon indépendante par l'environnement, les services ne sont pas toujours supportés de façon conjointe. Cette vérification permet, dans le cas où l'usager désire tester plusieurs services simultanément, d'éviter de construire des tests, qui finalement, donneraient des résultats contradictoires et non cohérents.

Le sous-module Lien reçoit les arguments validés par le Contrôleur, puis procède à une recherche dans la base de données des composantes de cas de test afin de déterminer pour chaque service sous test, le routage, le trafic, et la validation nécessaires au script de test. La base de données est un fichier hiérarchisé contenant pour chaque service des cas de test, et dans chaque cas de test, les références aux composantes de trafic, validation, routage, etc. Les composantes sont stockées à leur tour dans des fichiers distincts.

Les noms des composantes retrouvées sont mémorisés dans des vecteurs qui sont retournés au sous-module Contrôleur.

Service	**Cas-Test**	**Trafic**	**Routage**	**CLI**	**Validation**
S_1	$CT_{11}...CT_{1X}$	$T_{11}...T_{1X}$	$R_{11}...R_{1X}$	$C_{11}...C_{1X}$	$V_{11}...V_{1X}$
S_2	$CT_{21}...CT_{2Y}$	$T_{21}...T_{2Y}$	$R_{21}...R_{2Y}$	$C_{21}...C_{2Y}$	$V_{21}...V_{2Y}$

Le sous-module 'Organisation' organise l'information relative aux cas de test. Les composantes de trafic, routage, etc, sont regroupées par service ou groupe de services. Cette information est retournée au sous-module 'Contrôle' qui se

charge de l'enregistrer dans 4 fichiers sous format XML : trafic, routage, CLI et validation (voir figure suivante).

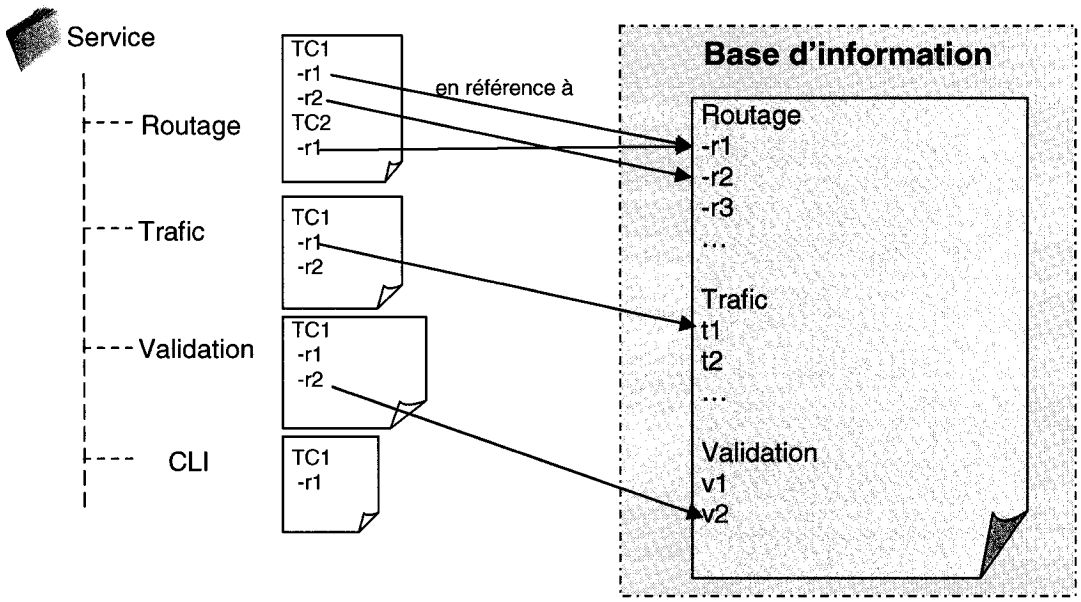


Figure 16. Fichiers générés par le Sélecteur

Après la création des fichiers de composantes des cas de test, le module Sélecteur retourne un message au module de Gestion et de Contrôle du ScriptMaker, lui indiquant le succès ou l'échec de l'opération. Divers témoins ont été insérés dans divers sous modules afin d'indiquer, avec la plus grande précision possible, les causes d'un échec.

Lorsque l'opération de sélection connaît un succès, le module de Gestion reçoit, sous forme d'un vecteur, la liste des composantes relative aux services et tests sélectionnés.

Finalement le module de Gestion et Contrôle de données, transmet les informations sur les composantes des tests au module de Gestion de structures,

afin que ce dernier puisse préparer la structure du Script de test qui s'adapte à ces modules.

4.5.2 Module de Gestion de structures

Ce module concerne la gestion des structures qui permettront de construire le script de test final. En effet, une fois les composantes de cas de test sélectionnées et organisées, il faut procéder à leur insertion dans la structure de script de test.

Le module de Gestion de structures comporte essentiellement trois sous-modules : repérage de script de test en fonction des tests sélectionnés, validation des données de chacune des composantes et finalement l'insertion.

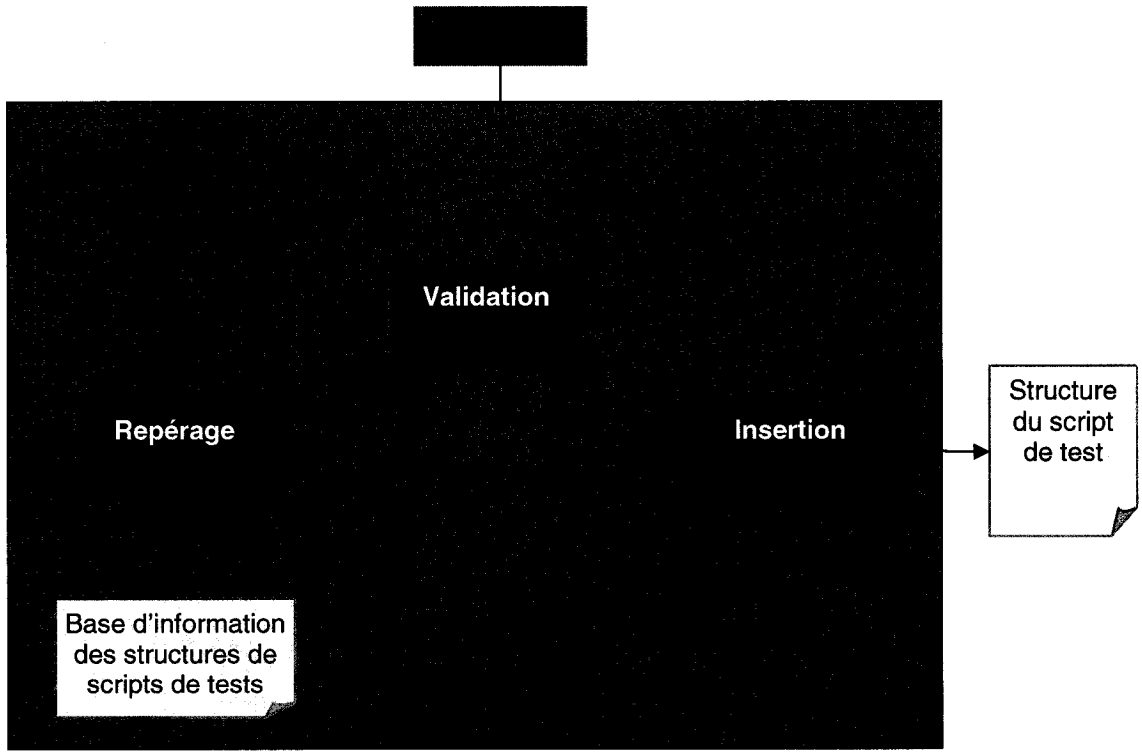


Figure 17. Architecture du module Gestion de structures

Le sous-module Repérage reçoit en arguments un vecteur contenant les divers services et composantes des cas de test. Ce vecteur est déterminant, car selon les types de composantes sélectionnées, un fichier correspondant à un Script de test est choisi dans la base d'information des structures de Script de tests.

Le sous-module Validation consiste à comparer les procédures globales aux divers paramètres des composantes de cas de test. En effet, pour chaque procédure globale, il y a un ou plusieurs cas de test qui lui font appel, cela signifie, l'existence dans le cas de test d'un appel de fonction contenant des paramètres. Une fois ces procédures validées, elles sont transmises au sous module d'insertion.

Le sous-module d'Insertion recherche le libellé de chaque procédure et de chaque composante de test dans la structure du script de test repéré, puis insère une à une les composantes sélectionnées. Finalement le fichier structuré du Script de test est enregistré et le nom est retourné au module de Gestion et de Contrôle de données.

4.5.3 Module d'Instanciation des structures

Ce module reçoit en argument le nom du Script de test structuré, le nom des services configurés et l'environnement de test. Son rôle est de parcourir les diverses commandes CLI existantes dans le fichier Script et de valider les données (paramètres, valeurs, etc). La validation des paramètres du CLI se fait à travers le module Meta-CLI.

Une fois la validation terminée, ce module génère deux fichiers relatifs au Script de test : le premier sous format XML, est réutilisable dans le cas où l'on désire apporter quelques changements au niveau de l'environnement, ou sur certaines valeurs, puis le second fichier sous format texte, sert à lancer et exécuter les tests sur le banc d'essai.

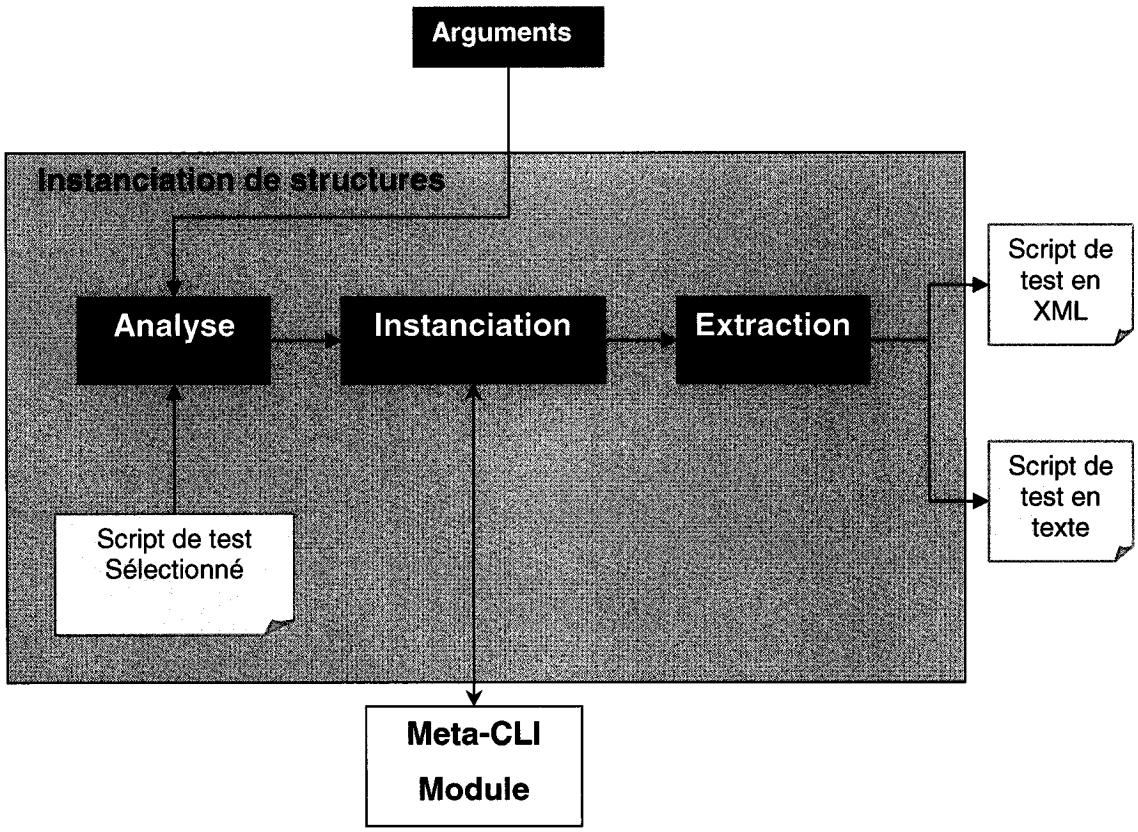


Figure 18. Architecture du module d'Instanciation de structures

Le sous-module Analyse reçoit sous forme d'un vecteur, le nom du Script de test, les services et l'environnement de test. Il parcourt le fichier de Script de test et relève toutes les commandes CLI existantes (on reconnaît une commande CLI par son libellé). Ces commandes sont enregistrées dans une table, et leurs emplacements initiaux dans le script de test sont remplacés par des références. Finalement la table est transmise au sous module d'instanciation.

Le sous-module d'Instanciation transmet au module Meta-CLI les arguments de service et d'environnement, puis chaque commande CLI pour la validation des paramètres et des valeurs. Dans le cas, ou une commande CLI doit comporter un paramètre supplémentaire (inexistant dans la table), ce dernier se verra attribuer une valeur par défaut inscrite dans l'arbre du Meta-CLI.

Une fois les paramètres de la table validés, ils sont insérés dans le script de test. Cette instanciation, se fait en suivant les références et les libellés dans la structure du script de test.

Finalement le sous-module Extraction charge le fichier Script sous format XML et retire tous les libellés pour créer un fichier sous format texte qui sera exécuté sur l'environnement de test sélectionné. Le Module de Gestion et de contrôle interagit avec l'usager pour donner un nom spécifique aux fichiers générés par ce sous-module.

4.5.4 Module Meta-CLI

Le module Meta-CLI est responsable de la validation des services et de la validation des paramètres et valeurs de chaque commande CLI. Les concepts et détails sur le fonctionnement du Meta-CLI ont été largement abordés dans la première section de ce chapitre.

4.5.5 Scénarios d'utilisation du ScriptMaker

À travers la description de l'architecture du ScriptMaker, nous avons vu comment on arrive à générer un script de test à partir du choix de l'usager (service, environnement, etc.) et des composantes de tests. Il existe d'autres situations ou cas d'utilisations qui n'ont pas été abordés, mais néanmoins, ont été implémentés dans l'outil ScriptMaker.

Nous énonçons ci-dessous quelques exemples de scenarios de cas de test faisant partie des fonctionnalités du ScriptMaker.

4.5.5.1 Ajout de cas de test à un script de test

Ce scénario consiste à ajouter un ou plusieurs cas de test à un script de test existant (figure suivante). L'usager introduit le nom du script de test, l'outil

ScriptMaker retrouve dans sa base d'information les cas de test disponibles, puis l'usager sélectionne dans une liste de choix les cas de test qu'il désire ajouter au Script.

Le contenu du ou des cas de test sélectionnés est validé de façon à conserver une cohérence avec l'environnement de test. Le ScriptMaker génère finalement le nouveau Script de test sous les deux formats déjà décrits précédemment.

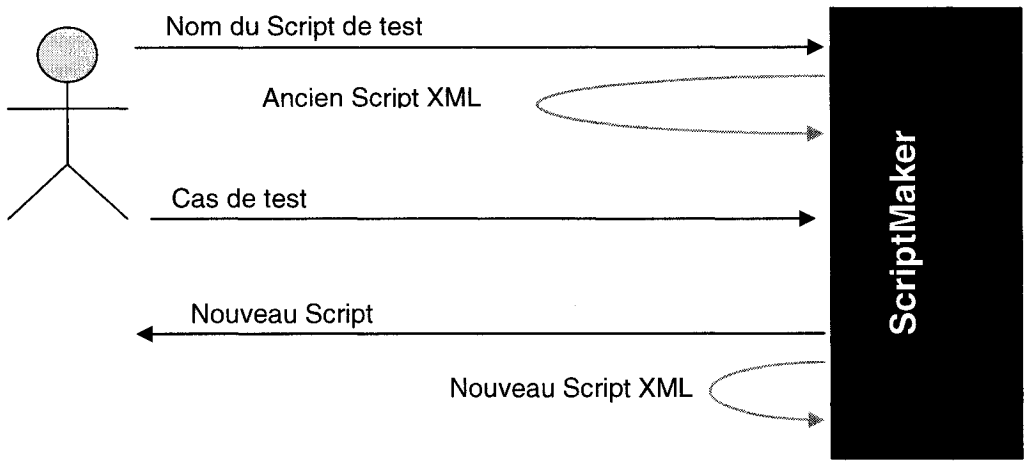


Figure 19. Scénario d'ajout de cas de test au script de test

Ce scénario est similaire à celui de la création d'un script de test décrit dans l'architecture; cependant, le traitement est moins lourd et seulement quelques sous-modules sont concernés par cet ajout.

4.5.5.2 Modification d'une composante de cas de test

Qu'il s'agisse de trafic, de routage, ou de validation, le ScriptMaker permet la modification des cas de test en modifiant une de leurs composantes. L'usager

doit sélectionner le nom du script de test qu'il désire modifier, puis une liste de cas de test et leurs composantes respectives.

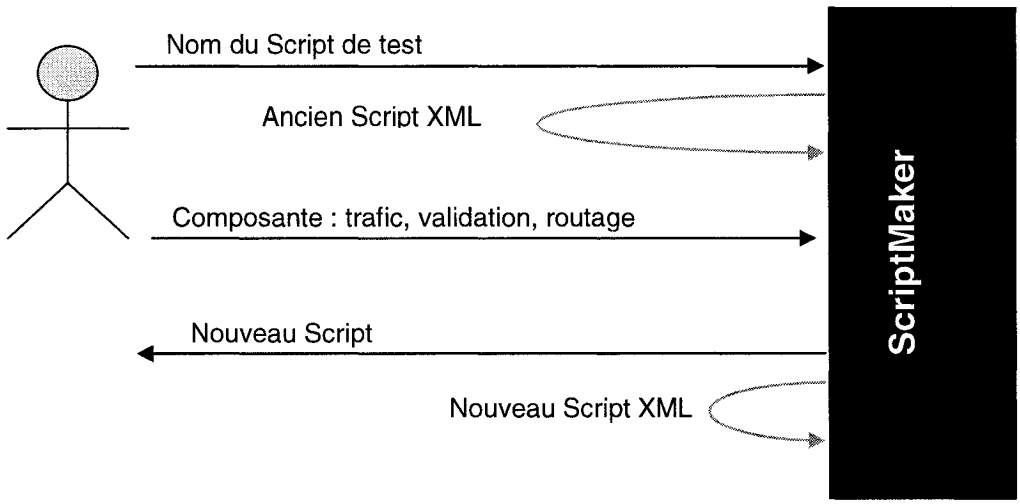


Figure 20. Scénario de modification d'une partie du cas de test

Une fois sélectionnées, les composantes à modifier sont validées dans le but de créer une cohérence avec l'environnement de test et les autres composantes du cas de test. Ainsi, s'il s'agit d'une composante de trafic, il ne faut pas avoir dans un même cas de test deux types de trafic, ou deux types de routage qui sont mutuellement exclusifs.

La composante validation peut avoir plusieurs formes; mais encore là, l'usager doit maîtriser le contenu. Si une composante de validation est choisie arbitrairement, elle peut s'insérer dans le script de test, mais à la fin du processus de test, les résultats seront non conformes à ce qui est escompté.

4.5.5.3 Modification de l'environnement de test

Ce scénario est très utilisé, car les interfaces et les versions d'OS sont des éléments en constante évolution, et le besoin de tester les mêmes services sur

des nouvelles versions d'OS ou de type de NIC, est réel. Pendant les phases de test de régression, la réutilisation des scripts de test avec une modification d'un élément d'environnement. Elle constitue donc une nécessité et un défi.

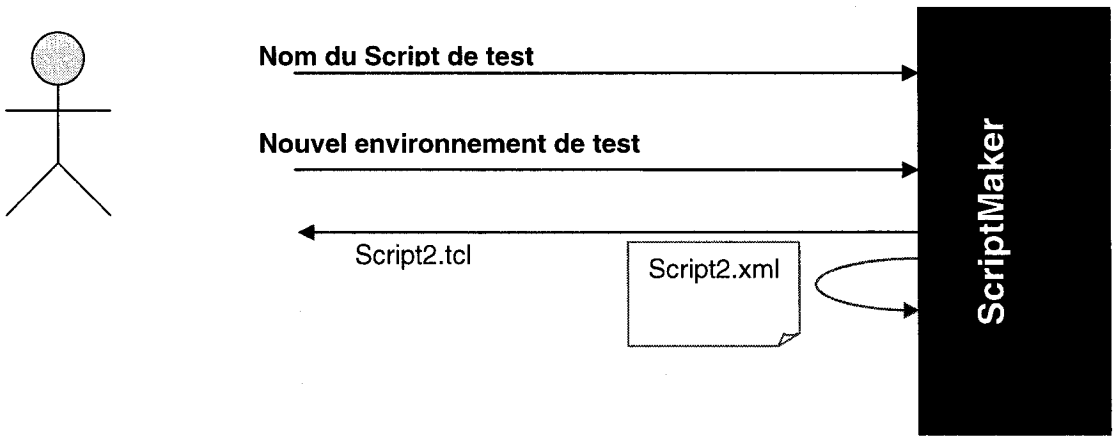


Figure 21. Scénario de modification de l'environnement de test

Le ScriptMaker permet donc de réutiliser les cas de test existants, ou les scripts de tests afin d'en modifier le contenu conformément au nouvel environnement de test.

Les modules concernés par ce changement sont essentiellement le Meta-CLI et celui de l'instanciation des structures. En effet, lors du changement d'environnement, les cas de test du Script de test choisi, ne subiront pas de changement dans leurs objectifs, mais seulement dans la validité des valeurs et paramètres de leurs composantes. Ainsi chaque configuration ou routage ou trafic, se verra valider ses paramètres à travers l'arbre du Meta-CLI. Les nouvelles commandes, paramètres ou valeurs sont instanciées dans un nouveau Script de test.

4.6 Résultats découlant du développement de l'outil ScriptMaker

L'outil ScriptMaker a été conçu sur la base d'un banc de test constitué de 3 routeurs et un générateur de trafic de type IXIA. Le test ne s'effectue que sur une interface (NIC) d'un routeur désigné à cet effet; ceci implique que les interfaces ne peuvent être testées que de façon indépendante.

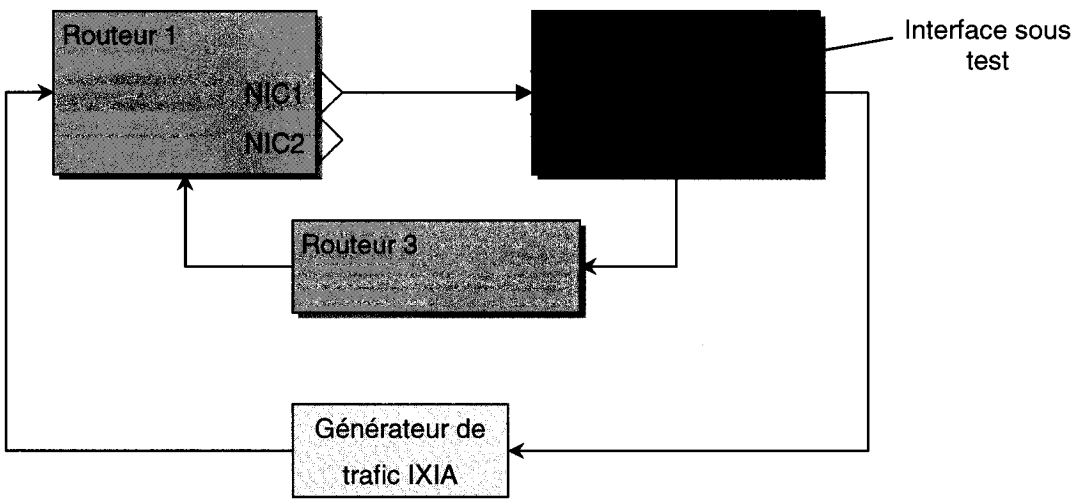


Figure 22. Banc d'essai pour le ScriptMaker

Le ScriptMaker a permis la génération de plusieurs Scripts de test, la plupart ont été élaborés pour des fonctionnalités de services indépendamment de l'existence d'autres services.

Malheureusement, pendant l'implémentation de l'outil ScriptMaker, le banc d'essai a été retiré par la compagnie, pour être remplacé par un autre plus performant. Cette décision nous a obligés à reconsidérer nos objectifs et à vérifier la validité de l'outil par des moyens plus simples, comme la comparaison de fichiers.

Ainsi, nous avons pris quelques exemples de Scripts de tests réalisés manuellement par les testeurs de la compagnie, construit les composantes nécessaires à ces scripts de tests et finalement, reconstitué les scénarios pour chaque service et chaque test. Nous avons également varié certains paramètres d'environnement et vérifié que le script de test correspond bien à celui généré manuellement. Pour comparer les Scripts de test générés par l'outil ScriptMaker avec ceux créés manuellement, nous avons utilisé l'outil «Diff».

Lorsqu'on choisissait le même environnement que dans le cas manuel, on retrouvait le même script de test, par contre lorsqu'on modifiait l'environnement de test (type de NIC, version OS, etc), on observait des différences dans les commandes CLI de chaque service, parfois dans les paramètres de trafic. Cette approche nous indiquait d'une part, que le script de test réalisé avec l'assistance de ScriptMaker correspondait parfaitement à celui qui a avait été réalisé de façon manuelle, et d'autre part, les changements apportés au point de vue environnement donnaient lieu à des variations cohérentes au niveau de la configuration et du trafic.

Finalement, nous avons regroupé plusieurs services dans un même script de test, mais la validation du résultat était moins évidente à réaliser. En effet, pour construire un script de test pour plusieurs services en même temps, plusieurs facteurs doivent être pris en considération et les composantes de test de chaque service doivent être analysées pour sortir un trafic commun, un routage commun et une validation commune à tous les services sous test. Comme l'outil ScriptMaker ne contrôle pas tous les aspects et liens entre les différentes composantes et services dans un test, il reproduit les cas de test de chaque service, tout en gardant dans la configuration les commandes CLI associées à tous les services. On peut ainsi valider fonctionnalité par fonctionnalité, mais avec tous les services configurés dans l'interface sous test.

4.7 Analyse et discussions

L'outil ScriptMaker a été développé essentiellement dans le but de minimiser les coûts associés aux tests. Cette diminution de coûts est directement proportionnelle à la capacité d'adaptation de l'outil à différentes situations de test et à sa puissance de réutilisation des tests déjà réalisés. Cependant, le fait de pouvoir tester plusieurs services de façon simultanée est un facteur non négligeable dans la réduction des coûts, et de façon indirecte, peut mener à la détection des interactions entre les services. L'interaction de service est un souci majeur pour tout testeur désireux de livrer un produit exempt d'erreurs.

La description de l'architecture de l'outil ScriptMaker montre clairement comment les tests peuvent être réutilisés par le biais de la reconstitution de leurs composantes.

Par ailleurs, pour tester plusieurs services, le choix des composantes peut être réalisé et leur insertion dans une structure de Script de test est possible, mais certains aspects échappent au contrôle du processus de construction du script de test. Par exemple, si deux services sont tolérés par l'environnement, et leur composantes de test ne sont pas contradictoires, ScriptMaker ne peut pas savoir si ces services sont mutuellement exclusifs au point de vue fonctionnel. Si un service permet au trafic de passer et un autre le lui interdit, il n'est pas logique de les faire cohabiter dans une même configuration; sauf si c'est un cas de test déterministe dédié à cet effet.

Ma contribution dans le cadre de ScriptMaker se situe dans l'analyse et conception des modules Sélection, Gestion et Contrôle de données, et finalement l'Instanciation de Structures. Le module le plus important étant celui de la Sélection, car il représente la base du processus menant à la réutilisation des composantes de script de test.

4.7.1 Rôle du module Sélecteur dans le ScriptMaker

Le module Sélecteur est une composante clé dans l'architecture du ScriptMaker, il est responsable de la détection et de la préparation de la « matière première » nécessaire à la construction du script de test.

Parmi les possibilités qu'offre le ScriptMaker, citons le test conjoint de services multiples. Par la sélection de plusieurs composantes de tests relatives à des services distincts, il est possible de générer un script dont le but est de tester ces services de façon simultanée, efficace et fiable.

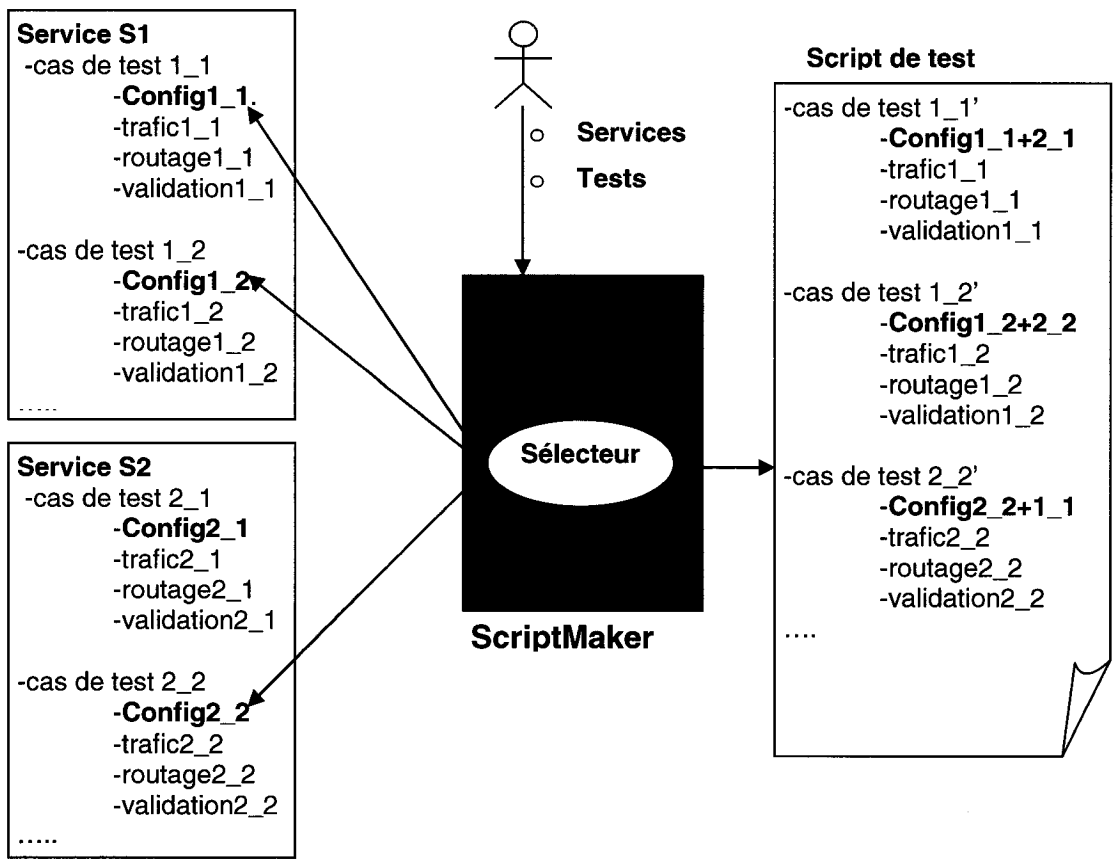


Figure 23. Regroupement de plusieurs tests dans un script de test

La figure précédente montre comment à partir des composantes de cas de test relatives à des services distincts, il est possible de construire un script de test conjoint (voir exemple de la figure suivante). Le sélecteur reçoit les arguments décrivant le choix de l'usager (services, cas de test, environnement), puis retrouve pour chaque service et cas de test, les composantes de configuration, le trafic, le routage et la validation à considérer.

Si l'usager désire réutiliser uniquement les cas de test1_1 et 1_2 de S1 et le cas de test2_2 de S2; alors le script de test doit contenir les trois cas de test 1_1, 1_2, et 2_2. Il reste à déterminer quelle instance de service (Config) doit figurer dans chaque cas de test. Dans cet exemple, nous avons choisi les couples de config. (1_1,2_1), (1_2, 2_2), et (1_1, 2_2), de cette façon, le nombre de cas de test dans le script correspond à celui choisi par l'usager.

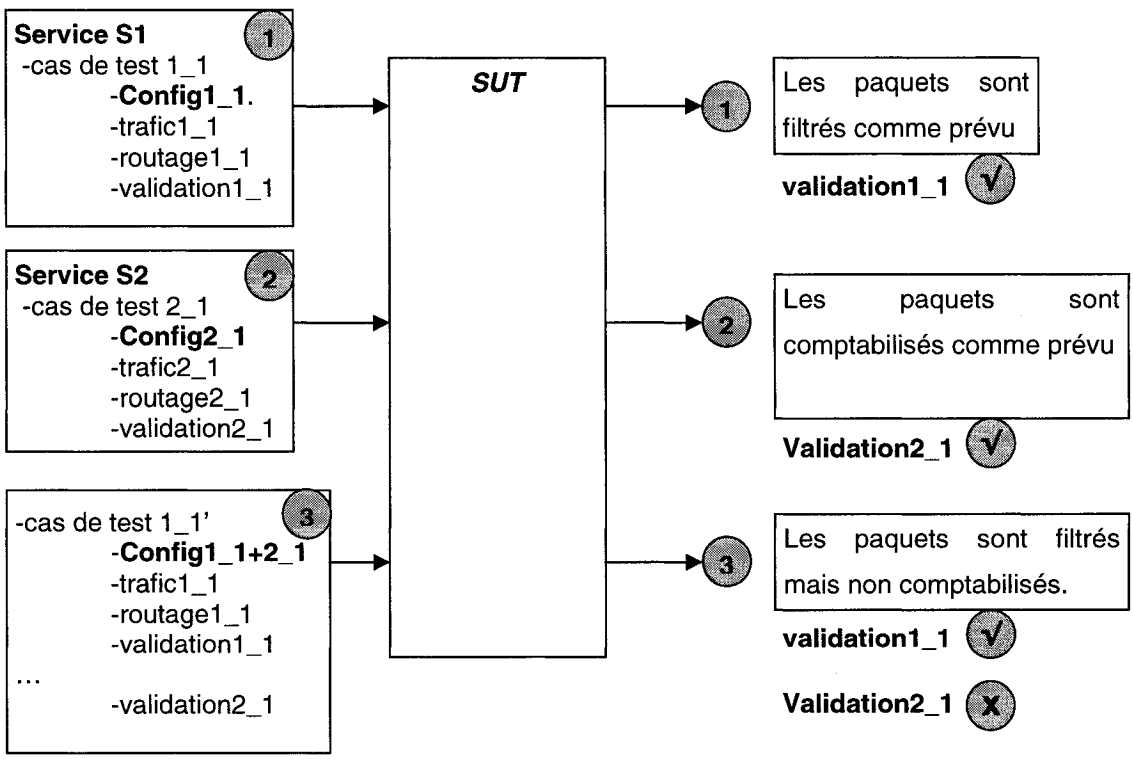


Figure 24. Exemple d'interaction de service

Le script de test réalisé peut nous amener à détecter une interaction de service, car la présence conjointe de plusieurs services sur le routeur peut provoquer une réaction qu'on ne peut déceler autrement. Par exemple si S1 est un service qui filtre les paquets provenant d'un réseau et S2 un service comptabilisant les paquets (avec S1 prioritaire sur S2), alors tous les paquets filtrés ne seront jamais comptabilisés. Cette interaction entre S1 et S2 n'est possible à déceler que lorsque S1 et S2 sont configurés à travers le même cas de test (3).

4.7.2 Avantages et lacunes de l'outil ScriptMaker

Chaque utilisateur peut se permettre d'effectuer des modifications dans son environnement de test, et demander à l'outil ScriptMaker de générer les nouveaux scripts de test. Ce outil offre donc la possibilité de générer des scripts de test et également de réutiliser les scripts existants. Cette réutilisation engendre un gain important dans l'effort de production des tests, et par conséquent minimise les coûts associés aux tests.

Le plan de test est établi avant le lancement des tests. Dans le cas du ScriptMaker, cela n'est pas toujours respecté. Comme l'outil permet d'exécuter plus de tests, et d'essayer plus de possibilités, le plan de test initial devient facilement obsolète; il faut alors procéder à sa mise à jour de façon manuelle.

L'outil ScriptMaker ne gère pas les changements topologiques, si par exemple, un routeur ayant le même environnement passe d'une position «Customer Edge» vers une position «Provider Edge», son comportement peut changer et les éléments de tests doivent être reconsidérés en conséquence; chose qui n'est pas faisable par le ScriptMaker. Finalement, ScriptMaker ne fait pas d'association entre les paramètres et les valeurs des différentes composantes d'un même cas de test.

4.8 En résumé

Pour développer un outil qui assiste la génération de scripts de tests aux fins de valider des services sur un routeur, nous sommes passés par deux phases essentielles : l'abstraction de l'information de test par rapport à l'environnement, puis la structuration des scripts de tests et les diverses composantes qui les forment. L'abstraction de la configuration est possible grâce au concept de Meta-CLI (Chapitre 4.1), qui selon sa structure hiérarchique, donne à chaque paramètre du CLI une description pouvant identifier les valeurs possibles, les versions d'OS qui le supportent, et le type d'équipement dans lequel ces paramètres peuvent être configurés. La deuxième phase, a consisté à organiser sous forme de petites structures toutes les composantes du script de test. Ainsi, une commande CLI devient une succession de paramètres, une méthode devient une succession de paramètres et de sous-paramètres, un trafic devient un ensemble de paramètres, etc. Il devient facile d'identifier l'emplacement de toutes les composantes du Script de test et pouvoir ainsi modifier leur contenu, ou ajouter d'autres composantes.

Le ScriptMaker est un outil qui offre plusieurs fonctionnalités facilitant la création, la modification et la réutilisation de scripts de tests. Les différents modules qui le composent ont une grande cohésion et une faible interaction, ils utilisent essentiellement des fichiers XML pour échanger l'information. La force de l'outil réside essentiellement dans la capacité de réutilisation des tests existants et ce, quelque soit l'environnement. Le ScriptMaker peut être automatisé si les différentes dépendances entre les composantes de tests sont bien contrôlées. Cela donnerait à l'outil la possibilité de détecter d'éventuelles interactions entre les services (voir chapitre suivant). Soulignons cependant que pour les raisons pratiques énoncées plus tôt, ScriptMaker n'a pas été utilisé à cette fin.

Chapitre 5 Combinaisons de cas de test

Pendant les tests de régression, il y a une monopolisation de ressources matérielles, logicielles et humaines, afin de reproduire à petite échelle des scénarios réels. Cette monopolisation engendre forcément une augmentation des coûts relatifs aux tests. Pour minimiser certains coûts, les ingénieurs ont mis sur pied un banc de test regroupant plusieurs NIC et supportant de multiples services. Ainsi, il devient possible d'effectuer des tests parallèles sur plusieurs équipements.

Cependant, le test de plusieurs services configurés sur plusieurs interfaces reste un défi de taille. Il est facile d'activer plusieurs services sur plusieurs interfaces, mais, il reste difficile d'effectuer des tests sur plusieurs services de façon simultanée.

L'intérêt premier d'effectuer des tests sur plusieurs services en même temps est de déceler d'éventuels comportements -non désirés- qui pourraient se produire lorsque ces tests sont appliqués au système sous test. Lorsque ces comportements sont observés, on parle alors d'une interaction de service.

Ce chapitre décrit de façon sommaire la manière dont se produit une interaction de service. Il introduit le concept de réutilisation des cas de test, expose l'idée de combiner les cas de test et les règles qui gèrent le processus de combinaison, et finalement présente les résultats de tests appliqués au banc d'essai. Ces résultats incluent les tests indépendants relatifs à des services distincts, et les tests combinés s'appliquant à plusieurs services. Une reconstitution d'un scénario d'interaction de service a été réalisée dans le but de montrer l'intérêt de combiner des cas de test.

5.1 Détection de l'interaction de Service

L'interaction de service est plus simple à déceler quand on dispose d'une spécification exhaustive du système sous test, on peut par exemple construire un diagramme d'états et voir à quel moment certaines fonctionnalités entrent en conflit de ressources. Cette approche, permet de déceler les interactions de service pendant l'étape d'analyse et de conception, même si cela n'implique pas que le système va fonctionner correctement une fois placé dans son environnement physique. Par ailleurs, la spécification des systèmes sous forme de diagramme d'états n'est pas toujours possible ou disponible, ce qui nous amène à essayer de détecter d'éventuelles interactions de services pendant les phases de test de régression.

Un système peut changer de comportement sous l'effet d'un test et provoquer ainsi un disfonctionnement de certaines fonctionnalités. Généralement, cela est dû à un manque de ressources (matérielles ou logicielles). Par exemple, dans un système où il y a une priorité dans l'utilisation des ressources, une fonctionnalité peut se retrouver pénalisée par une autre fonctionnalité. L'exemple qui suit donne un aperçu du type d'interaction de service dont il sera question dans la suite de ce chapitre.

- Soit A, B deux services d'un système, R_A une ressource nécessaire utilisée par A et S_i l'état du système à un instant i.
- Soit T_{ij} un test faisant transiter le système de la situation S_i vers la situation S_j.

- On suppose que A est prioritaire sur B.
- √ : service fonctionnel
- X : service non fonctionnel

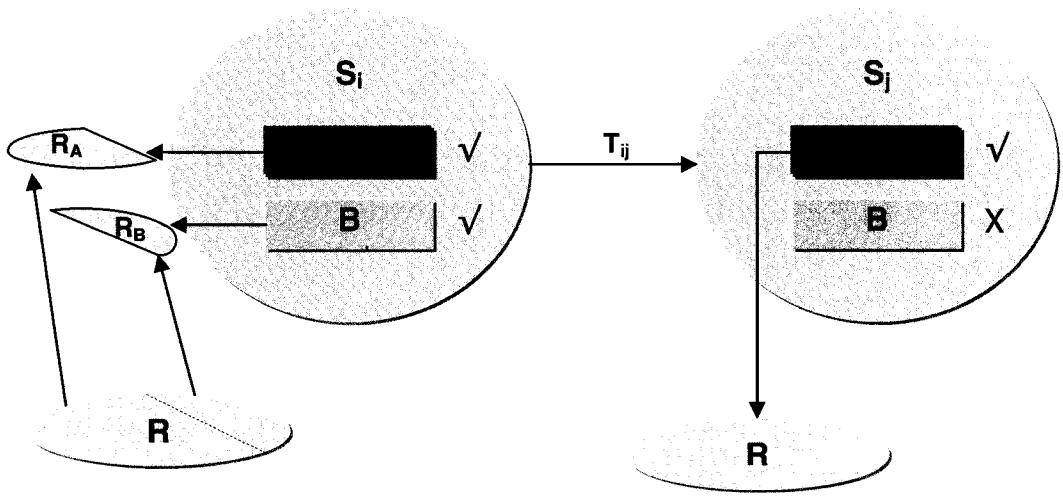


Figure 25. Répartition des ressources entre deux services

Dans 'Si', le service A requiert seulement une partie de la ressource R, ce qui laisse la possibilité au service B d'en utiliser une partie également. Cette répartition de la ressource R permet aux deux services A et B de fonctionner en même temps.

Le test T_{ij} fait transiter le système de 'S_i' vers 'S_j'. Ce changement implique que le service A requiert la totalité de la ressource R, et comme A est prioritaire sur B, il obtient toute la ressource (R_A = R). Ce changement du système entraîne évidemment un manque de ressource pour le service B et par conséquent provoque son disfonctionnement, on parle alors d'une interaction entre les services A et B.

5.2 Réutilisation des cas de test

Dans un plan de test on retrouve des contraintes liées à l'environnement de test (ex : banc d'essai, APIs, etc.) et au système lui-même (ex : fonctionnalité, performance, endurance, etc.). Les cas de test ou vecteurs de tests sont une implémentation du plan de test, il est donc possible de les réutiliser.

Lorsqu'on modifie un système ou on lui ajoute des fonctionnalités, non seulement il est impératif de lui faire subir les tests concernant ces modifications, mais aussi vérifier ce qui a déjà été testé auparavant. Ceci engendre inévitablement une hausse des coûts de développement du système.

Pour réduire le temps de test et par conséquent le coût qui lui est associé, on peut envisager de réutiliser les tests existants et réaliser des tests simultanés sur plusieurs fonctionnalités. Pour réaliser des tests simultanés, il faut combiner de simples cas de test relatifs aux fonctionnalités du système sous test (voir figure 26). Ce concept est décrit plus en détail dans les sections suivantes et les résultats sont discutés en fin de ce chapitre.

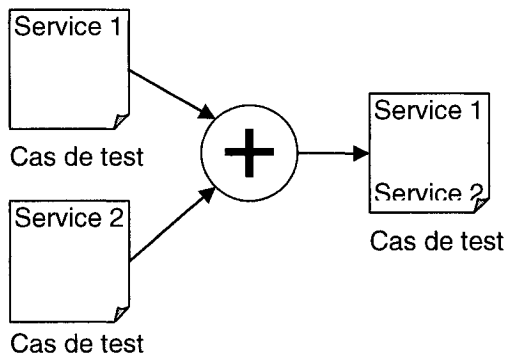


Figure 26. Combinaison de cas de test relatifs à des services distincts

La combinaison des cas de test relatifs à des services distincts, requiert une structure générique de test. Nous avons conçu cette structure qui nous permet de comparer des instances de tests et possiblement de les combiner.

5.2.1 Cas de test générique

La structure générique d'un cas de test est composée essentiellement de commandes et de paramètres. Une commande peut-être vue comme une

procédure et les paramètres comme des arguments ou variables. On peut ainsi construire un arbre représentant le cas de test et réutiliser ses instances. Cette structure donne des attributs à chaque commande ou paramètre, et par conséquent, la réutilisation n'est pas basée seulement sur la valeur des paramètres, mais également sur leur type.

La structure générique d'un cas de test est représentée dans la figure suivante :

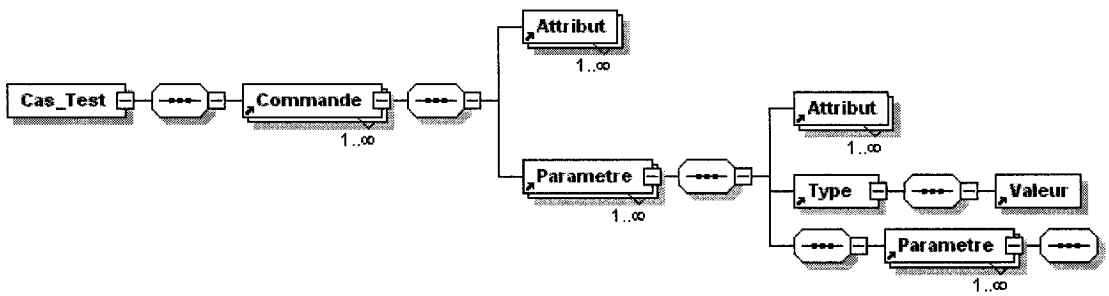


Figure 27. Structure générique de cas de test

Les attributs nous renseignent sur l'importance et les limites qui s'appliquent à chaque paramètre. Ainsi, deux paramètres appartenant à deux cas de test distincts peuvent être comparés sur la base de ces attributs et permettre leur combinaison lorsque cela est possible.

5.2.2 Combinaison des cas de test

À partir de la structure de test précédente, il est évident qu'on puisse générer des instances de cas de test pour différents services ou fonctionnalités. Au moment de la création d'un cas de test, il faut spécifier pour chaque paramètre ses attributs et ses contraintes (valeurs tolérées, exclusivité, etc.). La valeur par défaut d'un paramètre est considérée lorsqu'aucune spécification n'est donnée pour ses attributs.

Les paramètres ne doivent pas concerner des fonctionnalités contradictoires et ne doivent pas s'annuler mutuellement. Pour éviter ce type de situation, il faut établir des règles qui permettent de comparer les paramètres et de fixer les règles de décision pouvant amener (ou pas) à une combinaison de cas de test.

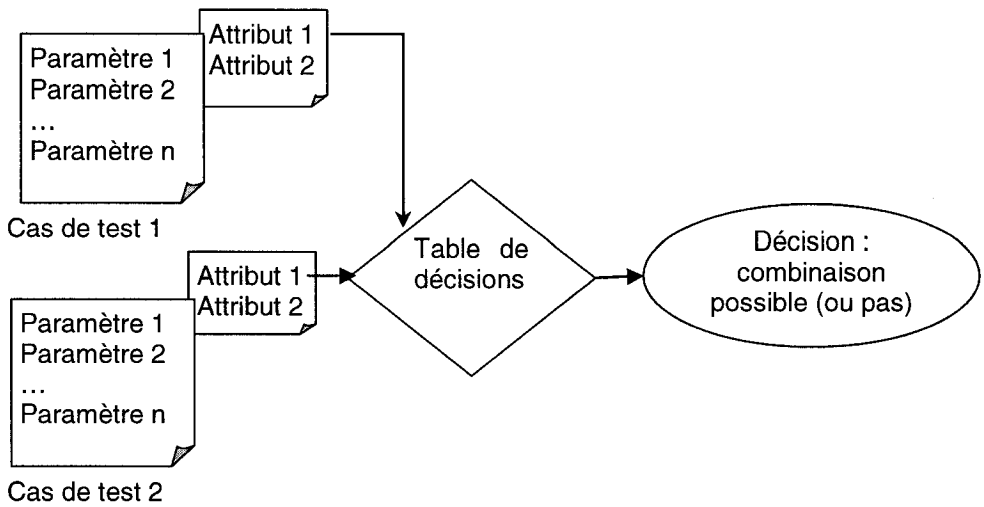


Figure 28. Comparaison de cas de test par type d'attribut

Chaque cas de test possède des commandes et chaque commande est composée de un ou plusieurs paramètres. Lorsque deux cas de test sont comparés, les attributs de même type sont comparés, suivis des valeurs de chacun des paramètres.

L'algorithme qui suit, décrit le processus permettant de comparer deux cas de test, et la possibilité de les combiner. Cette possibilité est tributaire des attributs et des valeurs de chaque paramètre.

Dans la prochaine section, on discutera des requis pour décider si deux paramètres appartenant à deux cas de test peuvent être combinés ou pas.

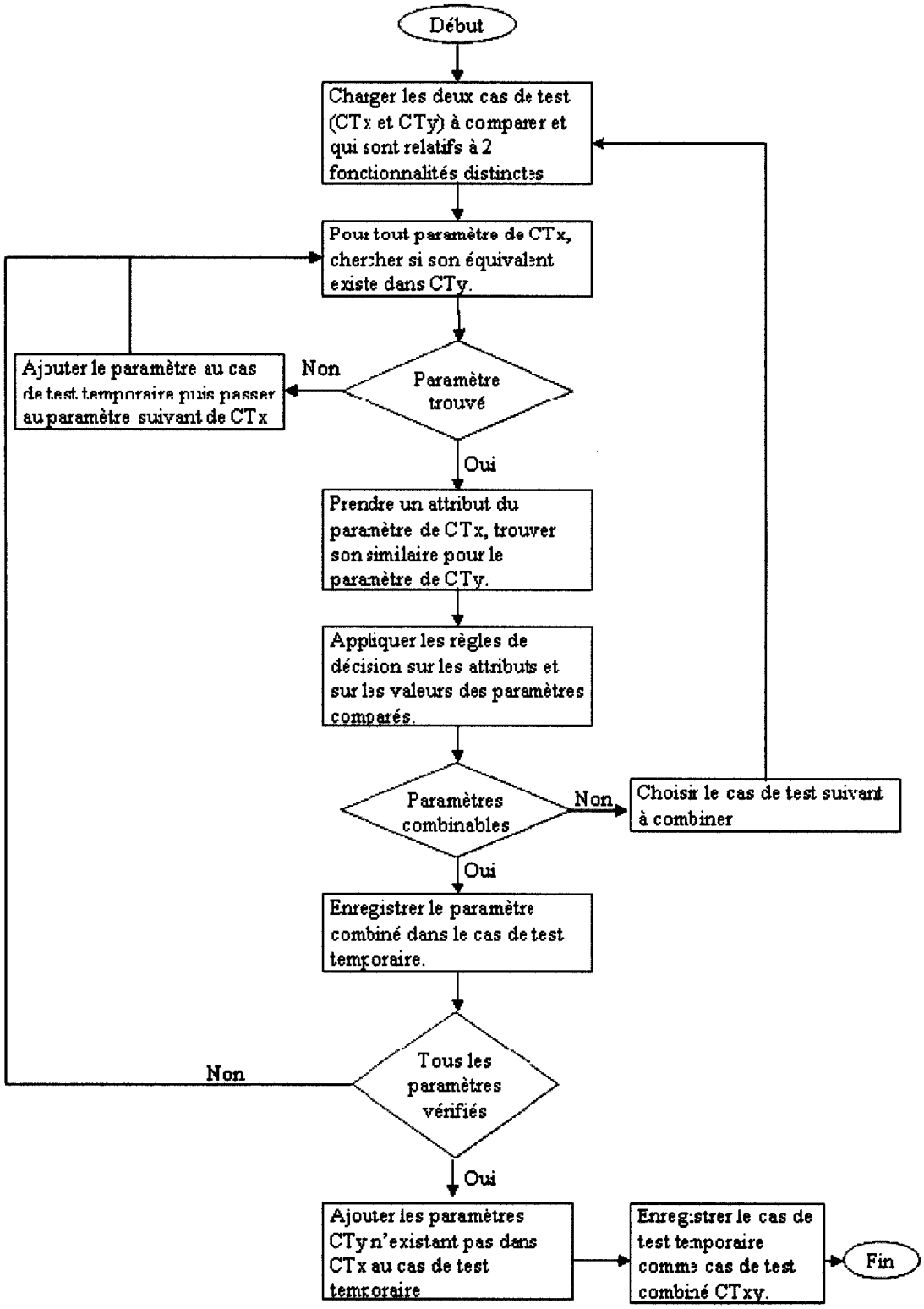


Figure 29. Algorithme de combinaison de cas de test

5.2.3 Prise de décision

La prise de décision est basée sur deux aspects importants : un aspect structurel et un autre comportemental. L'exemple qui suit illustre le cas d'une comparaison basée sur l'attribut qui a un aspect structurel.

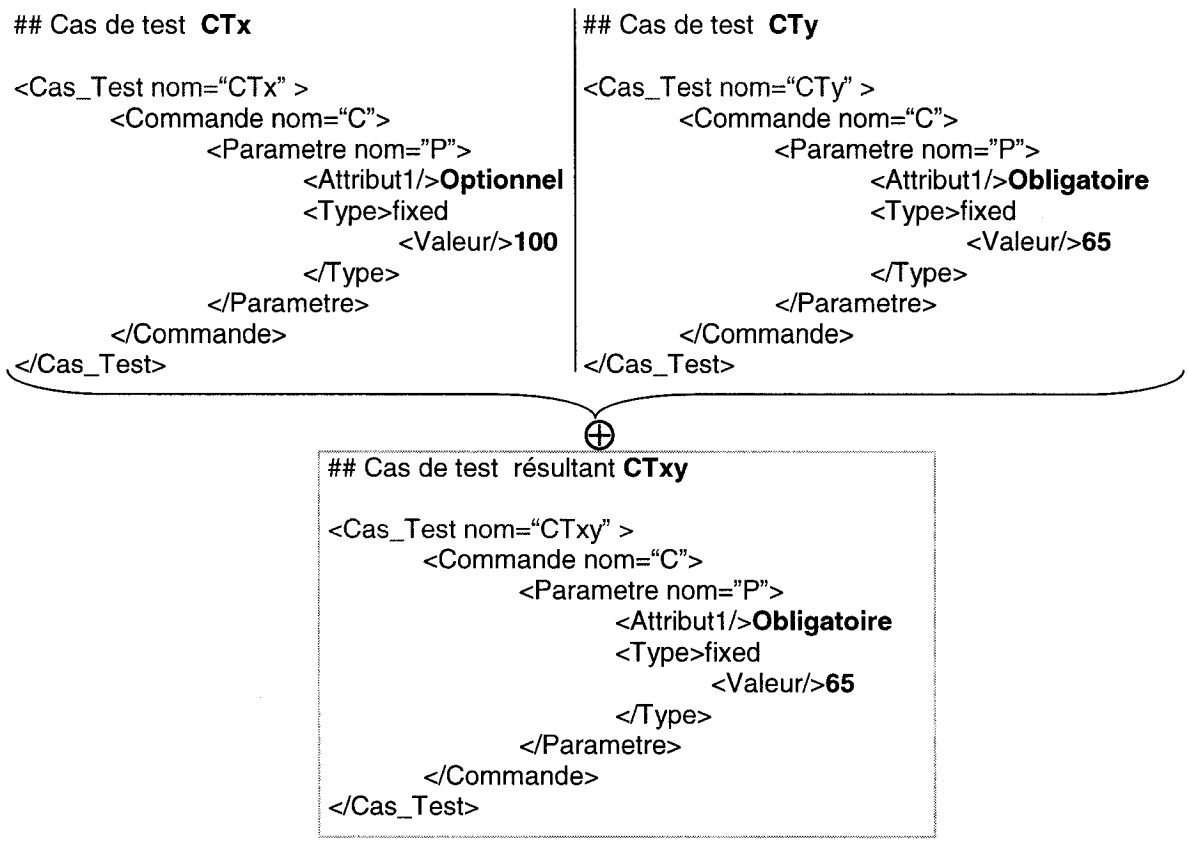


Figure 30. Comparaison structurelle

Le terme '*optionnel*' signifie que le paramètre P du cas de test CTx prend sa valeur par défaut. Le terme '*obligatoire*' signifie que le paramètre P du cas de test CTy a une valeur bien précise. Dans cet exemple, le paramètre combiné P du cas de test CTxy aura la valeur 65 de P de CTy (obligatoire). Lorsque les deux paramètres à combiner ont un attribut '*obligatoire*', il faut passer par des règles de décision. Celles-ci seront détaillées dans la section suivante.

Afin d'appliquer les règles de décision, les paramètres sont comparés selon un deuxième attribut. Celui-ci est de type comportemental.

L'attribut comportemental d'un paramètre consiste à donner à celui-ci des restrictions de telle façon à ce qu'on puisse le comparer avec d'autres paramètres. L'attribut peut être une valeur minimale, un intervalle de valeurs, une exclusion, etc; autant de critères pouvant être considérés pour aboutir à une décision de combinaison finale.

En supposant deux paramètres dont l'attribut structurel est '*obligatoire*', voici quelques cas de figures possibles :

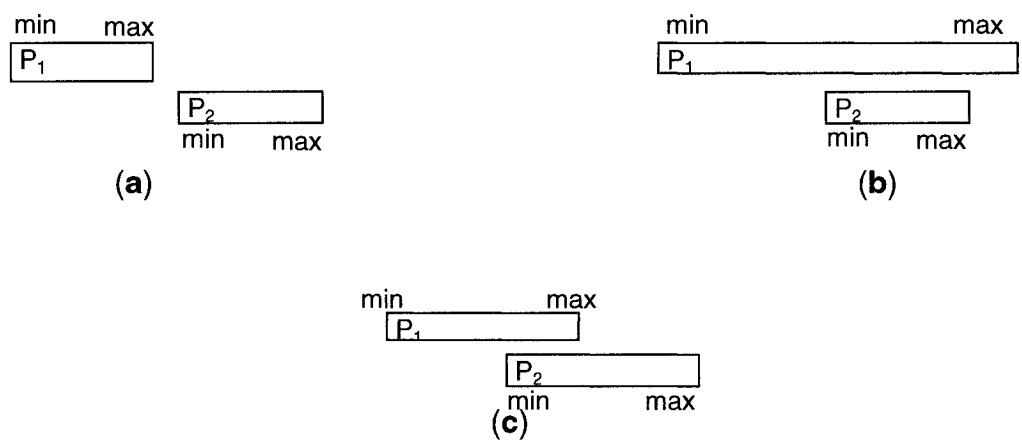


Figure 31. Exemple de combinaison de paramètres de type numérique

(a) Les deux cas de test ne peuvent être combinés, car P1 et P2 ont des valeurs disjointes.

(b) Il est possible de combiner les deux paramètres P1 et P2, le paramètre résultant sera '*obligatoire*' et aura les valeurs de P2.

(c) Il est également possible de combiner P1 et P2. Le paramètre résultant aura un attribut structurel 'obligatoire', et des valeurs se situant dans l'intervalle entre min de P2 et max de P1.

La section qui suit, décrit les règles de décision nécessaires pour les cas de test s'appliquant à un routeur configurable.

5.3 Cas d'un routeur

La présente section et les sections subséquentes, représentent un cas pratique réalisé en laboratoire sur des routeurs. Ce cas nous permet de concrétiser les concepts énoncés précédemment, et d'illustrer leurs intérêts à la lumière des données recensées et des résultats obtenus.

Globalement, tout service possède déjà des cas de test conçus pendant la phase de développement. Lorsque deux ou plusieurs services sont configurés sur la même interface d'un routeur, on se doit de tester l'interaction entre eux. À cette fin, il est possible de combiner les tests relatifs à chaque service engendrant ainsi la création de nouveaux cas de test pouvant être appliqués simultanément aux services configurés. La combinaison des cas de test permet également de minimiser le nombre de tests et par conséquent le temps qui lui est associé.

Actuellement, il n'existe pas de méthode systématique permettant de tester de façon simultanée plusieurs services sur une même carte d'un routeur, et de vérifier d'éventuelles interactions de services non désirées.

On se propose donc d'utiliser le concept énoncé dans la section précédente pour combiner les cas de test en utilisant leurs paramètres de configuration, de trafic et de validation. Cette technique permettra de combiner les cas de test tout en conservant les objectifs des tests pour chaque service. Le but ultime de cette technique sera de tester des interactions -non désirées- entre ces services.

5.3.1 Hypothèses

Pour que la combinaison des cas de test soit possible, il faut au préalable s'assurer que :

- La coexistence des services soit possible dans un même environnement de test, ce qui implique, que les services sont supportés par l'OS et par l'équipement.
- Les services ne soient pas contradictoires.
- Les services soient configurés sur les interfaces
- Les services configurés et les cas de test qui leurs sont associés, soient valides.

5.3.2 Règles de combinaison

Même si tous les paramètres présents dans un cas de test ont un rôle à jouer dans le test, leur importance reste néanmoins variable. Par exemple, si un test consiste à filtrer les paquets selon leur adresse IP, le cas de test définira un trafic dont les paquets peuvent avoir des tailles variables, mais dont l'adresse IP est bien précise. Le paramètre IP est donc '*obligatoire*' pour le test, mais celui de la taille des paquets est '*optionnel*' ou par défaut.

En utilisant les règles structurelles, on identifie les paramètres 'essentiels' pour le test d'un service en particulier. Une fois ces paramètres identifiés, on applique les règles comportementales qui nous renseignent sur les décisions à prendre lorsque l'attribut est '*obligatoire*' pour les deux paramètres à combiner.

5.3.2.1 Règles structurelles

Ces règles sont utiles dans la mesure où on veut alléger le poids de la comparaison. Trois situations se présentent :

- Lorsque deux paramètres P1 et P2 ont l'attribut '*optionnel*', le résultat de la combinaison sera une valeur par défaut.
- Si P1 a un attribut '*optionnel*' et P2 a un attribut '*obligatoire*', alors le résultat combiné aura la valeur et l'attribut du paramètre P2.
- Lorsque P1 et P2 ont un attribut '*obligatoire*', il faut procéder à une vérification en appliquant les règles comportementales.

5.3.2.2 Règles comportementales

Les règles comportementales sont basées sur les attributs de type comportement de chaque paramètre. Ces attributs sont comme suit :

- **Min** : Cet attribut signifie que le paramètre peut avoir comme valeur minimale celle indiquée dans le cas de test.
- **Max** : Cet attribut signifie que le paramètre peut avoir comme valeur maximale celle indiquée dans le cas de test.
- **Exclusif** : cet attribut indique que le paramètre a une valeur exclusive qui ne peut être combinée avec une autre qui ne joue pas le même rôle.
- **NonMixable** : la valeur du paramètre dans le cas de test est fixe.
- **Concatène** : Cet attribut est utile surtout pour des commandes qui peuvent s'ajouter dans un même cas de test.
- **Plage**: cet attribut indique le domaine de validité des valeurs (min, max) du paramètre.

Les règles comportementales qui s'appliqueront à des paramètres ayant ces attributs, sont résumées dans le tableau qui suit.

Tableau 1. Exemple de table de décision pour la combinaison de cas de test

		Règle structurelle		Règle Comportementale					
	Attributs de V2 / **Attributs de V1**	Optionnel	Obligatoire	Min	Max	Exclusif	NonMixable	Concatène	Plage(n,m)
Règle structurelle	Optionnel	V1 ou V2	V2	V2	V2	V2	V2	V2	V2
	Obligatoire	V1	Comportement						
	Min	V1		Max(v1,v2)	(v1,∞) ∩ (0,v2)	N/A	V2 si v2>v1	N/A	(v1,∞) ∩ (n,m)
	Max	V1		(0,v1) ∩ (v2,∞)	Min(v1,v2)	N/A	V2 si v2<v1	N/A	(0,v1) ∩ (n,m)
	Exclusif	V1		N/A	N/A	V1 ou V2 si v1⇔v2	N/A	N/A	N/A
	NonMixable	V1		V1 si v1>v2	V1 si v1<v2	N/A	V1=V2	N/A	V1 si v1 ∈ (n,m)
	Concatène	V1		N/A	N/A	N/A	N/A	V1 suivi de V2	N/A
	Plage(n,m)	V1		(v2,∞) ∩ (n,m)	(0,v1) ∩ (n,m)	N/A	V2 si v2 ∈ (n,m)	N/A	V1(n,m) ∩ V2(n,m)

V1 : valeur dans le test
v1 : valeur selon l'attribut

Le tableau précédent représente la grille de décision qui a été utilisée pour réaliser les tests sur le band d'essai. Il regroupe les règles structurelles et les règles comportementales s'appliquant aux commandes et paramètres des cas de test. Pour mieux comprendre le fonctionnement de cette table, voici quelques exemples qui illustrent l'utilisation des règles.

Exemple1 : Le paramètre <adresse IP1> peut avoir une valeur quelconque dans un test qui valide la bande passante, par conséquent on peut lui affecter un attribut '*optionnel*'. Un autre cas de test basé sur le filtrage IP aura également un paramètre <adresse IP2> dont l'attribut est '*obligatoire*'. La combinaison des paramètres des deux cas de test donnera un paramètre dont l'attribut est '*obligatoire*' et dont la valeur est IP2.

Exemple2 : Soit V1 un paramètre correspondant à la taille d'un paquet et dont l'attribut '*min*' indique la valeur minimale du paramètre tolérée dans le test. Soit V1 ('*min*') = 512Octets, et soit V2 ('*min*') = 256Octets. Le résultat de la combinaison de V1 et V2 est donc un paramètre V3 dont l'attribut sera '*min*' et dont la valeur est 512Octets.

Exemple3 : Supposons un paramètre V1 qui correspond à la tolérance de perte de paquets dans le routeur et dont l'attribut est '*max*'. Celui-ci indique la valeur maximale du paramètre tolérée dans le test. Soit V1 ('*max*') = 2%, et soit V2 ('*max*') = 5%. Le résultat de la combinaison de V1 et V2 est donc un paramètre V3 dont l'attribut sera '*max*' et dont la valeur est 2%.

Exemple4 : soit un paramètre de validation dont la valeur est '*bloqué*'. Ce paramètre permet de vérifier que les paquets n'ont pas traversé le routeur. Ce paramètre est exclusif, il ne peut être combiné qu'à son semblable. Par exemple, dans notre banc d'essai, on ne peut valider dans un même cas de test les paquets bloqués et ceux qui sont passés.

Exemple5 : si deux commandes ou paramètres ont comme attribut 'concatène', elles peuvent être groupées dans le même cas de test. Par exemple, une commande qui affiche le nombre de paquets filtrés et une autre qui affiche la taille moyenne des paquets.

L'exemple concernant l'attribut '*Plage(n,m)*' est décrit dans la section 5.2.3 figure 31.

L'application des règles structurelles et comportementales à travers la table de décision se fait toujours dans cet ordre. Ainsi, si une commande a un attribut structurel 'optionnel', alors tous les paramètres de la commande le sont également. Si au moins un paramètre d'une commande est obligatoire, alors la commande l'est aussi. Cette façon de faire, permet d'alléger le traitement des cas de test lors de la comparaison, c'est-à-dire, avant la combinaison.

5.4 Description du fonctionnement du banc d'essai

Pour réaliser les tests, nous avons utilisé un banc d'essai composé de deux routeurs modulaire et un générateur de trafic. Ce dernier génère du trafic IP avec un entête MPLS, puis l'injecte au premier routeur; celui-ci enlève l'entête MPLS et achemine les paquets IP au second routeur en générant des flux selon les données des cas de test.

La description du banc d'essai est importante, car elle nous permettra de mieux comprendre certains résultats et la façon de les interpréter.

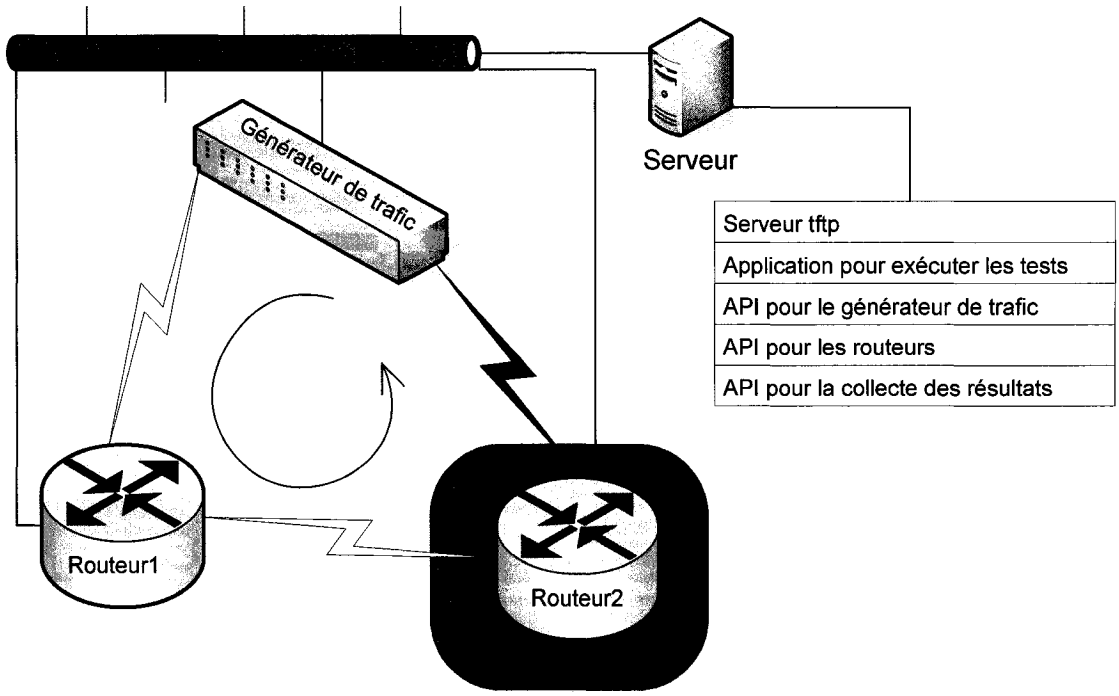


Figure 32. Aperçu du banc de test et son environnement

Les testeurs communiquent avec le banc d'essai moyennant une connexion Ethernet qui relie un serveur d'applications aux routeurs et au générateur de trafic. Le serveur comporte :

- Un serveur TFTP pour stocker les images OS et les configurations de base des routeurs
- L'application qui exécute les tests, les fichiers de topologies et de cas de test, et les résultats des tests.
- Une API pour permettre à l'application qui exécute les tests de communiquer avec le générateur de trafic.
- Une API pour ouvrir des sessions sur le routeur et exécuter des tests.
- Une API pour collecter les résultats de tests, les analyser puis les valider.

Les deux routeurs possèdent les mêmes NIC qui sont connectées physiquement les unes aux autres. Chaque NIC possède plusieurs interfaces physiques et plusieurs sous-interfaces logiques.

L'application permettant d'exécuter les tests, offre à l'usager la possibilité d'activer une ou plusieurs interfaces afin de construire le fichier de topologie. Ce fichier sert à construire les tables de routage du routeur sous test.

5.5 Processus d'exécution des tests

Ce processus passe par deux phases : la mise au point du banc d'essai et l'exécution des tests. La mise au point du banc d'essai consiste à effacer l'image de l'OS ainsi que la configuration du routeur, puis recharger l'OS et la configuration choisis par l'usager. Pendant cette phase, plusieurs opérations de vérification matérielles et logicielles sont effectuées.

Avant d'exécuter les tests, l'usager doit sélectionner les interfaces qui constitueront la topologie. Le lancement des tests consiste à générer des flux de paquets selon les données du trafic, à valider les paquets sortants et finalement à récupérer tous les résultats des tests dans des fichiers logs.

Les résultats des tests impliquent plusieurs cartes du routeur simultanément, par conséquent, une analyse détaillée du fichier log s'impose. Ce fichier peut nous renseigner sur les cartes qui causent les échecs des tests par exemple.

5.6 Combiner des tests associés à différents services

Plusieurs services possédant chacun un ensemble de cas de test peuvent être testés conjointement en combinant leurs tests. Cette opération permettra de minimiser le nombre de cas de test, le temps d'exécution des tests et par

conséquent le temps de monopolisation du banc d'essai, et finalement, la possibilité de détecter une éventuelle interaction de service non désirée.

Pour réaliser les tests, plusieurs configurations ont été considérées, contenant chacune plusieurs services. Les services sont configurés sur les interfaces choisies selon le fichier correspondant à la topologie. Les tests valides de chaque service sont exécutés et leurs résultats notés pour des fins de comparaison une fois la combinaison de tests réalisée.

Tableau 2. Les résultats prévisibles après combinaison des tests

Résultats de tests		
Avant combinaison		Après combinaison
Service1	Service2	Service (1 + 2)
Passé	Échoué	Échoué
Passé	Passé	Passé
Échoué	Passé	Échoué
Échoué	Échoué	Échoué

Le tableau ci-dessus permet de prédire les résultats des tests combinés à partir de ceux des tests singuliers. Lorsque ces dits résultats ne coïncident pas avec ceux du tableau, il faut prospecter pour déterminer s'il n'y pas cas d'interaction entre les services sous test.

5.7 Résultats des tests : analyse et interprétation

Cette section contient tous les résultats obtenus pendant les simulations sur le banc de test. Quatre configurations ont été utilisées durant les essais. La première nous permettait de mesurer le temps d'exécution des tests, et de valider certains cas de test. La deuxième configuration avait un service de plus que la première et possédait plus de cas de test. Une topologie différente a été

utilisée dans ce cas. Les cas de test des deux premières configurations ont été réutilisés afin de réaliser des combinaisons permettant de tester plusieurs services sur plusieurs cartes de façon simultanée. La troisième et la quatrième configuration ont été réalisées dans le but de montrer comment on peut aboutir à une interaction de service à partir d'une combinaison de cas de test.

Les cas de test sont décrits en annexe (annexe 3).

5.7.1 Résultats de tests sur Config1

Tableau 3. Tests singuliers sur la configuration Config1

Cas de test	Suite de tests	Durée du test	Résultats du test
Test1	TestSuite1	15mn	Passed
Test2	TestSuite2	17mn	Failed
Test2	TestSuite3	40mn	Failed
Test3			Passed
Test4			Failed
Test5			Failed
Test6			Passed
Test7			Passed
Test8			Failed
Test9	TestSuite4	60mn	Passed
Test10	TestSuite5	17mn	Passed
Test11	TestSuite6	15mn	Failed
Test1	TestSuite7	17mn	Passed
Test12			Passed
Test13			Passed
Test14			Passed

Les résultats dressés sur le tableau ci-dessus ont été réalisés grâce aux cas de test relatifs à chaque service de la configuration Config1. On peut observer qu'un cas de test peut durer en moyenne 17mn, et pour une suite de test, la durée moyenne par test est de 5mn. Cette différence est due au délai important de la mise au point du banc d'essai.

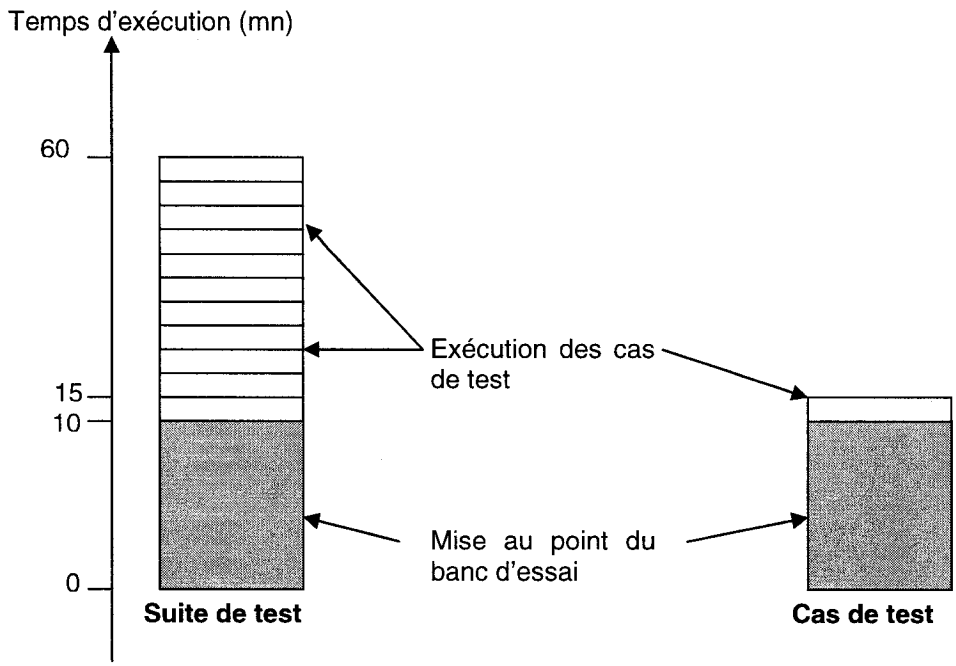


Figure 33. Temps d'exécution de cas de test et de suite de test

Dans le schéma ci-dessus on entend par suite de test, un ensemble de cas de test exécutés les uns après les autres dans un même environnement et avec un même système d'exploitation OS.

Il ne faut pas confondre suite de test et test combiné. En effet, ce dernier représente un cas de test pouvant tester plusieurs fonctionnalités simultanément dans un même cas de test.

Par ailleurs, les résultats ont parfois échoué (failed), cet échec n'est pas recensé sur toutes les interfaces et sous-interfaces Une investigation des fichiers contenant la trace du test (log file), a permis de relever les interfaces ayant un comportement non conforme à ce qui est prévu par la validation du test et qui sont à la base de ce résultat.

D'un autre côté, on remarque que les mêmes cas de test donnent les mêmes résultats pour différentes suites de tests (ex : Test2 pour TestSuite2, TestSuite3), le fichier log indique également que ce sont les mêmes interfaces qui causent un échec. Ce dernier s'explique par l'un des phénomènes suivants :

1. Des versions d'OS sont moins stables que d'autres.
2. Certaines NIC ou interfaces sont instables sous certaines versions d'OS.
3. L'application qui permet d'exécuter les tests ne prévoit pas un délai entre le moment où le trafic est généré et celui où l'on commence à comptabiliser les paquets sortants.

Ce dernier point est important, car un routeur a besoin (comme tant d'autres équipements) d'un délai initial de fonctionnement pour atteindre son régime « normal » de fonctionnement. Ainsi, pendant ce délai, certains paquets sont perdus ou d'autres ne sont pas filtrés.

Afin de nous assurer de la stabilité des résultats, nous avons exécuté à plusieurs reprises les mêmes suites de test

5.7.2 Résultats de tests pour Config2

Sachant que certaines interfaces ont causés des échecs au cours des simulations sur la config1, ces interfaces ont été désactivées dans la topologie et de nouveaux tests ont été exécutés dans la config2. Celle-ci comporte des fonctionnalités supplémentaires par rapport à la config1.

Tableau 4. Tests singuliers sur la configuration Config2

Cas de test	Suite de tests	Durée du test	Resultats du test
Test1	TestSuite1	15mn	Passed
Test15	TestSuite8	60mn	Failed
Test16			Passed
Test17			Passed
Test18			Passed
Test19			Passed
Test20			Passed
Test21			Failed
Test22			Failed
Test23			Failed
Test24			Failed
Test25			Failed
Test26			Failed
Test4			Passed
Test3			Passed
Test27	TestSuite9	18mn	Passed
Test30			Failed
Test2	TestSuite10	31mn	Passed
Test3			
Test4			
Test5			
Test6			
Test7			
Test9	TestSuite4	60mn	Passed

Les résultats obtenus confirment que le temps nécessaire pour exécuter un cas de test est réduit lorsqu'on utilise une suite de test. Nous vérifierons par la suite que ce délai sera réduit encore plus en combinant les cas de test. Le cas de test Test9 constitue une exception. Il consiste à fragmenter les paquets et vérifier leur passage à travers le routeur sous test. La fragmentation des paquets IP explique la longue durée (60mn) pour un seul cas de test, ce qui dépasse largement celle des autres tests (moyenne 17mn).

5.7.3 Combinaison des tests pour Config1

La combinaison des cas de test a été réalisée telle que décrite dans la section 5.2.2 de ce chapitre. Les résultats d'exécution de ces cas de test combinés sont regroupés dans le tableau qui suit :

Tableau 5. Tests combinés sur la configuration Config1

Test Combiné	Cas de Test 1	Cas de Test 2	Durée du test	Résultats du test
PC1Test1_1	Test1	N/A	N/A	N/A
PC1Test2_1	Test2	Test4	16mn	Failed
PC1Test2_2		Test11	17mn	Failed
PC1Test2_3		Test10	17mn	Failed
PC1Test2_3		Test6	16mn	Failed
PC1Test3_1	Test3	Test10	17mn	Passed
PC1Test3_2		Test11	16mn	Failed
PC1Test4_1	Test8	Test10	15mn	Failed

Selon les règles de décision énoncées dans le Tableau1, les attributs des paramètres du Test1 ne permettent pas de le combiner avec d'autres tests.

Par contre les cas de test Test2, Test3 et Test8 ont pu être combinés avec ceux présentés dans le Tableau 5. La combinaison de ces cas de test respecte parfaitement les exigences énoncées dans le Tableau 2. (section 5.6).

À titre d'exemple:

Avant combinaison	Après Combinaison
Test3 'passed', Test10 'passed'	PC1Test3_1 'passed'
Test3 'passed', Test11 'failed'	PC1Test3_2 'failed'
Test2 'failed', Test11 'failed'	PC1Test2_2 'failed'

Pour chaque résultat obtenu, les traces (fichier log) ont été inspectées pour déterminer les causes des échecs, et ainsi, s'assurer que les interfaces recensées précédemment sont les mêmes qui sont à l'origine de ces échecs.

Par ailleurs, pour tester la configuration Config1 nous disposions de 14 cas de test, ce qui nous laisse (n*(n-1)/2) = 91 possibilités de cas de test combinables. Or, il n'a été possible de combiner que 7 d'entre eux. Le rapport donne 7/91 = 7.7%. On peut donc conclure que dans ce cas précis, la réduction du temps de conception et de réalisation de tests combinés est d'environ 8%. Ce faible pourcentage est dû essentiellement aux contraintes du banc d'essai. Par exemple, il n'est pas possible de valider deux aspects opposés, ni de générer deux différents types de trafic, etc.

On peut dire que pour la configuration Config1 nous avons pu réduire le temps de test simultané de plusieurs fonctionnalités d'environ 8%. Cette réduction est

significative lorsqu'on connaît les coûts réels de ces tests (salaires d'ingénieurs concepteurs, de développeurs, de techniciens, de l'équipement, etc.).

Bien qu'à travers ces cas de test combinés aucune interaction de service n'ait pu être détectée, il est possible de construire des cas de test qui, en les combinant peuvent nous faire aboutir à une interaction de service non désirée. Une simulation d'un exemple connu d'interaction de service a été réalisée grâce à cette technique (voir section 5.7.6).

5.7.4 Combinaison des tests pour Config2

Les cas de test combinés ont été exécutés sur la topologie utilisée sur la configuration 'Config2'. Les résultats de ces tests sont détaillés dans le tableau en annexe 2. D'après ce tableau, on remarque que le délai d'exécution d'un test combiné est sensiblement égal à celui des cas de test unitaires, ceci implique une réduction du temps de lancement des tests de 50%.

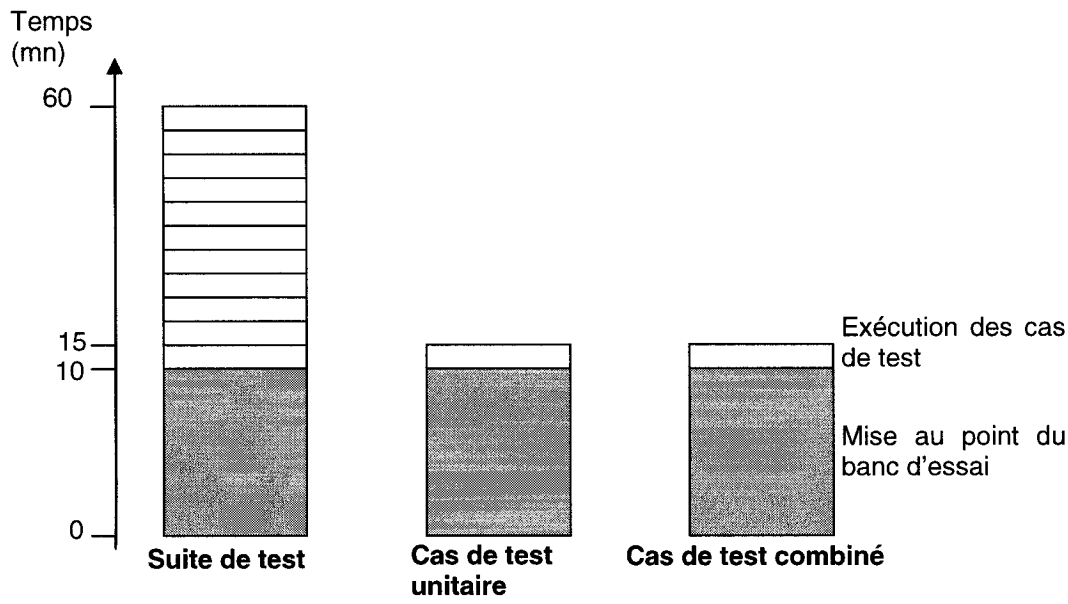


Figure 34. Temps d'exécution pour divers cas de test

Le cas de test combiné comporte réellement deux cas de test. Si on suit la logique d'une suite de test, on devrait avoir au minimum un temps d'exécution de 20mn, or les simulations révèlent que le délai d'exécution d'un test combiné est sensiblement égal à celui d'un simple cas de test.

Par ailleurs, pour la configuration Config2 nous disposions de 24 cas de test, donc 276 possibilités de cas de test à combiner. Le tableau en annexe 2 indique seulement 23 cas de test combinés. Il est ainsi possible de réduire le nombre de cas de test pouvant valider plusieurs fonctionnalités de façon simultanée, d'environ 23/276 = 8.3%. . En d'autre termes, on ne combine pas les tests, il faudra concevoir et développer 276 nouveaux cas de test, autrement, cette technique nous permettra de réduire ce nombre de 23. Ce faible pourcentage est également dû aux contraintes du banc d'essai.

Les résultats obtenus (~8% de réduction de l'effort de test) sont très significatifs pour une entreprise dont les coûts associés aux tests dépassent les 65% du coût de développement. Cependant, ce pourcentage pourrait être amélioré si on ne considère que certains cas de tests à combiner. Il y a toujours des services qui sont plus « sensibles » que d'autres et sur lesquelles on peut se concentrer afin d'appliquer cette méthodologie.

Par ailleurs, le temps que prend un test à s'exécuter est de moitié inférieur sachant qu'on peut vérifier deux services à la fois au lieu d'un. Cette réduction de temps d'exécution aura pour effet de diminuer le temps d'utilisation du banc d'essai, et par conséquent réduire les budgets qui lui sont alloués.

5.7.5 Reconstitution d'un scénario d'interaction de Service

Dans cette section, nous reconstituons un scénario d'interaction de service déjà connu pour montrer la pertinence de la combinaison des cas de test et son impact dans la détection de cette interaction.

5.7.5.1 Cas connu d'Interaction

Sur une NIC d'un type particulier, lorsque le nombre d'ACLs dépasse un certain nombre *n*, le service permettant de comptabiliser les paquets est désactivé: ce phénomène est dû à un manque de ressources matérielles.

Tableau 6. Le nombre d'ACL < n

Cas de test	Suite de tests	Durée du test	Resultats du test
Test28	TestSuite13	16mn	Passed
Test15			Passed

Le tableau ci-dessus indique que les cas de test Test28 et Test15 ont réussi. Ainsi, le résultat correspondant aux paquets échantillonnés par le service qui comptabilise les paquets, est égal à celui prévu par la validation, et les paquets dont les adresses IP permises par les cas de test, sont tous passés.

Des tests négatifs ont été élaborés également dans le but de conforter les précédents résultats. Par exemple, si on permet le passage de paquets ayant une adresse IP particulière, il faut s'assurer que les autres paquets n'ayant pas cette adresse, ne passent pas.

Ainsi, lorsqu'on choisit de tester le service qui comptabilise les paquets uniquement, le résultat est toujours un succès. Par contre, quand on combine son cas de test avec celui qui correspond au service ACL dont le nombre est supérieur à n, on s'aperçoit que le premier service ne fonctionne plus. Cette interaction entre ces deux services est définitivement due à la combinaison des deux cas de test, et plus précisément au nombre d'entrées ACL.

Tableau 7. Le nombre d'ACL > n

Cas de test	Suite de tests	Durée du test	Resultats du test
Test29 + Test28	TestSuite14	21mn	Failed
Test29			Passed

La Configuration *Config2* tend vers *Config3* sous l'effet du cas de test *Test29*

Sachant qu'au début de chaque test c'est la configuration Config2 qui est chargée, il est impossible de faire échouer le service qui comptabilise les paquets sans l'action du Test29.

Par ce cas concret, il est clair que la combinaison des cas de test peut nous amener vers une interaction de service non souhaitée. Comme cela a été dit précédemment, l'interaction de service est généralement une conséquence d'un manque de ressources plutôt que d'une contradiction entre les fonctionnalités des services.

5.7.6 Autre scénario d'interaction de service

Lorsque le service X est configuré sur des sous interfaces d'une NIC, l'interface en question ne laisse plus passer de trafic, tous les paquets sont rejetés. Ceci nous amène à un disfonctionnement de tous les services présents sur la dite interface.

Tableau 8. résultats des tests pour la config4.

Cas de test	Suite de tests	Durée du test	Resultats du test
Test2	TestSuite3	52mn	Failed

Test3			Failed
Test4			Failed
Test5			Failed
Test6			Failed
Test7			Failed
Test8			Failed

La configuration Config4 est comparable à Config1, la seule différence réside dans le fait que les sous-interfaces ont été configurées avec le service X.

Les résultats des cas de test 3, 6 et 7 ont réussi dans le cas de Config1, pourtant ils ont tous échoués dans la config4. Ce changement a été possible grâce au test combiné. Ce dernier a configuré le service X sur la sous-interface de la NIC qui a fait échouer tous les autres services.

Si les sous-interfaces de la NIC sont désactivées dans la topologie, la combinaison des cas de test n'aura aucun effet sur les résultats de la Config1.

L'interaction de service peut exister entre n'importe quels types de services. Cependant, il y a une exception à la règle. Lorsqu'il s'agit de services dont les fonctionnalités sont contradictoires, l'interaction est déjà visible, on doit donc se garder de les placer dans la même configuration.

Chapitre 6 Conclusion

Actuellement, les tests continuent de représenter une majeure partie de l'effort de développement d'un produit. Le moyen le plus évident pour minimiser cet effort, est la réutilisation de ce qui a déjà été créé. Cette réutilisation n'est pas sans conditions. Un ensemble de règles de validation doivent être accomplies afin de garantir une qualité de test. Les entreprises dépensent des sommes importantes pour améliorer leurs procédés de fabrication, mais cette amélioration n'est rien d'autre que le fruit d'une réflexion basée sur les résultats des tests.

Les méthodologies de tests adoptées dans le domaine matériel ou logiciel représentent une bonne base pour les tests des routeurs. Sachant que ces derniers sont similaires à des systèmes embarqués, et que leur fonctionnement ne peut être représenté par un diagramme d'état, il faut reprendre les méthodologies existantes et les améliorer en procédant à la structuration des éléments de test et à l'abstraction de l'information. Ainsi l'état du routeur peut être interprété comme étant une configuration ayant un certains nombre de commandes CLI, avec des paramètres et des valeurs. Bien que la configuration d'un routeur représente un ensemble de commandes cohérentes entre elles, il est toujours possible de distinguer les services qui sont activés sur le routeur et les fonctionnalités qui leurs sont associées.

L'environnement d'un routeur reste un facteur déterminant pour la configuration. En effet, sans connaissance préalable de la version OS ou du type des NIC du routeur, il est presque impossible de réaliser une configuration cohérente. Le concept du Meta-CLI offre la possibilité de générer une configuration en faisant abstraction de son environnement, de ses paramètres et valeurs. Cette abstraction permet de réutiliser la configuration sans se soucier de l'environnement dans lequel elle a été réalisée initialement.

Les scripts de test sont généralement écrits de façon séquentielle sans une véritable structure, il en résulte une difficulté de lecture, de compréhension et d'interprétation. À travers sa composition modulaire, l'outil ScriptMaker contribue à l'optimisation de l'effort de création de tests, et fournit les instruments nécessaires pour réutiliser toute information relative aux scripts de tests. Cependant, il comporte également des lacunes, telles que, le contrôle de l'exclusion entre les services, la contradiction des validations dans les cas de test, la validation de l'information de test en fonction de la topologie, etc. autant de points intéressants qu'il faudra prospecter et qui mèneront, sans aucun doute, à des travaux de recherche prenants.

Parmi les stratégies adoptées par l'entreprise pour minimiser ses coûts de tests, il y a la création de bancs d'essais capables d'exécuter des tests 'parallèles' sur plusieurs services configurés sur divers équipements, et ce de façon simultanée. La possibilité de tester plusieurs services conjointement, s'appuie sur la capacité de combiner les cas de test de ces différents services. La technique proposée dans le cadre de ce mémoire, a permis de réaliser un cadre dans lequel, les cas de test de différents services peuvent être combinés. Cette approche a aussi ses limites : il est indispensable de prendre un à un les paramètres de chaque cas de test, et leur assigner des attributs pouvant les identifier et les distinguer des autres. Cette assignation représente une tâche assez ardue, mais une fois réalisée, elle peut être utilisée indéfiniment.

La technique de combinaison des cas de test devrait pouvoir gérer également les relations entre la configuration, la topologie, le trafic, le routage, et les cas de test. Cette approche fait actuellement l'objet d'une recherche au sein du laboratoire téléinformatique de l' UQAM.

Les problèmes rencontrés lors de la réalisation des projets sont divers. À titre d'exemple, on peut citer : la difficulté de compréhension des diverses APIs et procédures d'accès et d'exécution de tests déjà en place, la difficulté de se familiariser avec l'équipement et logiciels utilisés par les ingénieurs de l'entreprise, les pannes matérielles et logicielles non connues de tous, ainsi que diverses modifications du banc d'essai qui ne nous ont pas été signalées.

Au cours de la réalisation des travaux de recherche qui ont mené à la rédaction de ce mémoire, plusieurs autres pistes ont été explorées. Il s'agit essentiellement de stratégies de détection de bogues et l'utilisation de BFMs afin d'abstraire les transactions entre les usagers et l'application qui génère les tests. La détection de bogues se base sur le principe de Pareto (80% des erreurs se trouvent dans 20% d'un système). La base d'information de l'entreprise Systems contient tous les bogues déjà détectés, il est donc possible de concevoir un dispositif permettant d'en découvrir de nouveaux. Par exemple, en utilisant des stratégies de tests pseudo-déterministes ou aléatoires. Ces idées de recherche n'ont pas été retenues parce qu'elles ne présentaient pas un potentiel de réalisabilité acceptable dans le cadre du projet et selon les priorités des requis.

Les résultats de recherche exposés dans le chapitre 5, représentent une compilation des données récoltées dans les fichiers comportant les traces des tests. L'effort d'analyse de ces fichiers est colossal. Par conséquent, il serait pertinent de produire des outils de 'parsing' afin de réduire le temps d'analyse et ainsi le coût relatif au test.

Références

[1] Amyot,D.; Logrippo,L.; "Directions in feature interaction research" School of Information Technology and Engineering, University of Ottawa.

[2] ARDITI, L.; CLAVÉ, G. " A Semi-Formal Methodology For The Functional alidation of An Industrial DSP System ". Proceeding of ISCAS2000, IEEE.

[3] Bartley,M.G.; Galpin,D.; Blackmore,T.; "A comparison of three verification techniques: directed testing, pseudo-random testing and property checking"- DAC 2002.

[4] BEIZER, B, "Software Testing Techniques ", 2nd ed. New York: Van Nostrand Reinhold - John Wiley & Sons, Inc -1990.

[5] BEIZER, B. 1995. Black-Box Testing: " Techniques for Functional Testing of Softwares and Systems ", John Wiley & Sons, Inc -1995.

[6] Chiueh,T.; Pradhan,P. "High-Performance IP Routing Table Lookup Using CPU Caching"- Computer Science Department State University of N.Y - 1999.

[7] Daveau,J.M.; Marchioro,G.; Amine Jerraya, A.; "Hardware/software co-design of an ATM network interface card: a case study " – IEEE - 1998.

[8] Deca,R.; Cherkaoui,O.; Puche,D.; "Configuration Model for Network Service Management" Proc. of the 6th IFIP/IEEE International Conference on Network Control and Engineering (Net-Con2003), Palma de Mallorca, Spain, November, 2004.

[9] Deca,R.; Mahrez,O.; Cherkaoui,O.; Savaria,Y.; Slone,D.; "Contributions to Automated Testing of Network Service Interactions" Notere - 2005.

[10] El-Darieby, M.; Rolia, J.; Petriu, D.C.; "Performance modeling for virtual network based service provisioning" - IEEE/IFIP International Symposium 2001

[11] Gottlieb,Y.; Peterson,L..- "A comparative study of extensible routers";Open Architectures and Network Programming Proceedings, 2002 IEEE28-29 June 2002

[12] Hang-Sheng,W.; Li-Shiuan,P.; Sharad,M - " A Power Model for Routers: Modeling Alpha 21364 and InfiniBand Routers " - (2002)

[13] Heimdahl, M.P.E.; George, D.; " Test-suite reduction for model based tests: effects on test quality and implications for testing " Automated Software Engineering, 2004.

[14] Hood, C.S.; Chuanyi Ji; " Intelligent agents for proactive fault detection "- Internet Computing, IEEE Volume 2, Issue 2, March-April 1998

[15] Keck, D.O.; Kuehn, P.J.; " The feature and service interaction problem in telecommunications systems: a survey " Software Engineering, IEEE - 1998

[16] Keshav,S.; Sharma,R.; "Issues and trends in router design" IEEE - 1998

[17] Kohler,E.; Morris,R.; Chen,B.; Jannotti,J.; Frans Kaashoek,M. " The click modular router " MIT - 2000

[18] Larson,R.E.; Low,C.S.; Rodriguez,P.; "Routing", Coriolis - 2000.

[19] Lee, D.C.; Harper, S.J.; Athanas, P.M.; Midkiff, S.F.; " A stream-based reconfigurable router prototype " - Communications, 1999

[20] Leung, H.K.N.; "Test tools for the Year 2000 challenges" - Euromicro Conference, 1998.

[21] Li, H.C.; Krishnamurthi, S.; Fisler, K.; " Interfaces for modular feature verification " Automated Software Engineering, 2002. Proceedings. ASE 2002. IEEE

[22] Michael, J.B.; Bossuyt, B.J.; Snyder, B.B.; " Metrics for measuring the effectiveness of software-testing tools " - Software Reliability Engineering, 2002.

[23] Newhall,B.; Lenoski,D.; "High-end Routers & Modern Supercomputers" Routing Technology Group NORDUnet 2003, Reykjavik - August 2003

[24] Puente,V.; Gregorio,J.A.; Beivide,R.; Izu,C.- "On the design of a high-performance adaptive router for CC-NUMA multiprocessors"; Parallel and Distributed Systems, IEEE Transactions onVolume 14, Issue 5, May 2003 Page(s)

[25] Rexford,J.; Feng,W.; Dolter,J.; Shin,K.G.; "PP-MESS-SIM : a flexible and extensible simulator of evaluating multicomputer networks " IEEE - 1997

[26] Rexford,J.; Hall Kang,J.; Shin,G. " A router architecture for real-time point to point networks " ACM - 1996

[27] Saraph, P.; Last, M.; Kandel, A.; " Test case generation and reduction by automated input-output analysis " -Systems, Man and Cybernetics, 2003. IEEE

[28] Wakahara, Y.; Fujioka, M.; Kukuta, H.; Yagi, H.; Sakai, S.I.; "A method for detecting service interactions " - Communications Magazine, IEEE - 2000

[29] White, L.; Leung, H.K.N.; " On the edge. Regression testability " - IEEE, April 1992

[30] White, L.J.; Narayanswamy, V.; Friedman, T.; Kirschenbaum, M.; Piwowarski, P.; Oha, M.; " Test Manager: A regression testing tool Software Maintenance " - 1993.

[31] www.ipv6.or.kr/wg/test/Interim/1-004.pdf

[32] www.juniper.net

[33] www.lightreading.com/document.asp?doc_id=4009

[34] www.spirentcom.com/analysis/technology.cfm?WS=7&SS=156&D=16

[35] www3.ietf.org/proceedings/03jul/161.htm

[36] Zeng,H.; Zhou,X.; Song,.; "On Testing IP Routers". Parallel and Distributed Computing, Applications and Technologies, 2003. PDCAT'2003.

Annexes

Annexe 1. Sommaire des Classes du package Selector

Package Selector

Class Summary	
Feature	Cette classe fournit et met à jour l'information relative au service et son environnement
Parameter	Pour identifier et valider tous les paramètres utiles pour le script de test.
Selector	La classe principale du module. Elle gère et synchronise les échanges entre les différents modules du package Selector.
TcSelector	Cette classe se charge de réaliser le choix des composantes qui serviront à construire le script de test.
TestFunctions	Traitement de données relatives à la configuration et aux cas de test.
TrafficSelector	Cette classe choisit le trafic en fonction des services et de l'environnement sous test.
Validation	Valide la cohabitation entre les services, ainsi que la validation des cas arguments reçu par le module Selector.
Link	Cette classe permet de faire le lien entre les services, l'environnement, et les éléments de base du script de test (trafic, routage, validation, etc.)
XmlFunctions	Cette classe regroupe toutes les fonctions qui organisent la structuration et l'organisation des composantes sélectionnées par le module Selector.

Annexe 2. Tests combinés pour Config2

Tableau 9. Tests combinés sur la configuration Config2

Configuration: Config2

Test Combiné	Cas de Test 1	Cas de Test 2	Durée du test	Résultats du test
PC2Test2_1	Test2	Test4	16mn	Passed
PC2Test2_2		Test11	17mn	Passed
PC2Test2_3		Test10	17mn	Passed
PC2Test2_3		Test6	16mn	Passed
PC2Test3_1	Test3	Test10	17mn	Passed
PC2Test3_2		Test11	16mn	Passed
PC2Test4_1	Test8	Test10	15mn	Passed
PC2Test5_1	Test15	Test27	18mn	Passed
PC2Test5_2		Test2	17mn	Passed
PC2Test5_3		Test1	15mn	Passed
PC2Test5_4		Test3	17mn	Passed
PC2Test5_5		Test4	17mn	Passed

PC2Test5_6		Test10	16mn	Passed
PC2Test5_7		Test11	16mn	Passed
PC2Test5_8		Test6	15mn	Passed
PC2Test5_8		Test7	16mn	Passed
PC2Test6_1	Test21	Test27	17mn	Failed
PC2Test7_1	Test16	Test3	21mn	Passed
PC2Test7_2		Test6	18mn	Passed
PC2Test7_3		Test7	18mn	Passed
PC2Test8_1	Test20	Test4	17mn	Passed
PC2Test8_2		Test6	16mn	Passed
PC2Test8_3		Test7	16mn	Passed

Annexe 3. Description des cas de test utilisés

Tableau 10. Tableau descriptif des tests

Test ID	Description
Test1	Vérifie le trafic dirigé vers une interface donnée
Test2	Simple compteur de paquets IP
Test3	Valeur minimale pour les ports source et destination du protocole TCP
Test4	Valeur maximale pour les ports source et destination du protocole UDP
Test5	Variation de la taille des paquets IP
Test6	TTL est fixé à 2, les paquets sont blogués
Test7	TTL est fixé à 3, les paquets passent
Test8	Test le service qui retrace les paquets lors d'une attaque pour certaines adresses
Test9	Test le passage des paquets fragmentés
Test10	Affiche les résultats des commandes
Test11	Compare les compteurs avant et après le passage du trafic
Test12	Vérifie le trafic dirigé vers certaines interfaces en utilisant un ttl élevé
Test13	Vérifie le trafic dirigé vers une interface Ingress
Test14	Vérifie le trafic passant d'une interface Ingress vers une interface Egress
Test15	Permet le passage de certaines adresses IP
Test16	Permet le passage des paquets TCP

Test17	Permet le passage des paquets TCP dont le port = 7
Test18	Permet le passage des paquets TCP dont le port est entre 109 et 113
Test19	Permet le passage des paquets UDP dont le port = 69
Test20	Permet le passage des paquets UDP
Test21	Interdit le passage de certaines adresses IP
Test22	Interdit le passage des paquets TCP
Test23	Interdit le passage des paquets TCP dont le port est supérieur à un chiffre donné
Test24	Interdit le passage des paquets TCP dont le port est entre 109 et 113
Test25	Interdit le passage des paquets UDP
Test26	Interdit le passage des paquets UDP dont le port est inférieur à un chiffre donné
Test27	Compare les compteurs avant et après le passage du trafic en utilisant le service de commutation rapide
Test28	Vérifie le service qui récolte les statistiques sur les paquets
Test29	Vérifie le service ACL avec plus de 128 entrées
Test30	Interdit certains paquets IP en utilisant le service de commutation rapide.