Cadre Formel pour le Test de Robustesse des Protocoles de Communication

Fares Saad Khorchef

N° d'ordre : 3292

THÈSE

PRÉSENTÉE À

L'UNIVERSITÉ BORDEAUX I

ÉCOLE DOCTORALE DE MATHÉMATIQUES ET D'INFORMATIQUE

Par **Farès SAAD KHORCHEF**

POUR OBTENIR LE GRADE DE

DOCTEUR

SPÉCIALITÉ : INFORMATIQUE

Un Cadre Formel pour le Test de Robustesse des Protocoles de Communication

---

Soutenue le : 13/12/2006

Après avis des rapporteurs :

Jean Arlat	Directeur de Recherche au CNRS
Hacéne Fouchal ...	Professeur

Devant la commission d'examen composée de :

Jean Arlat	Directeur de Recherche au CNRS	Rapporteur
Richard Castanet .	Professeur	Directeur de Thèse
Hacéne Fouchal ...	Professeur	Rapporteur
Yves Métivier	Professeur	Président du Jury
Hélène Waeselynck	Chargé de Recherches CNRS	Examinatrice
Antoine Rollet	Maitre des conférences	Examinateur

– 2006 –

Remerciements

Tout d'abord, je remercie toutes les personnes qui m'ont aidée tout au long de ce travail.

Je remercie mes parents pour leurs prières, mon frère "Saci", mes sœurs "Habiba" et "Soria" et ma belle sœur "Zahia" pour leur soutien moral.

Un merci solennel à ma future épouse "Hiba" qui m'a aidée, encouragée et supportée durant ces années.

Un grand merci à mes amis Antoine, Ismail, Xavier et Omer pour leur aide précieuse.

Un merci spécial à Richard CASTANET, mon directeur de thèse, pour le soutien qui m'a porté durant les années de thèse.

Je remercie aussi les rapporteurs, Jean ARLAT et Hacène FOUCHAL pour avoir consacré leur temps à la lecture de cette thèse, ainsi que toutes les membres du jury qui ont participé à cette soutenance.

Résumé

Dans le domaine des télécommunications, il est indispensable de valider rigoureusement les protocoles avant de les mettre en service. Ainsi, il faut non seulement tester la conformité d'un protocole, mais il s'avère aussi nécessaire de tester sa robustesse face à des événements imprévus.

La littérature concernant le test de robustesse est beaucoup moins conséquente que le test de conformité. A travers ce document, nous considérons la définition de la robustesse suivante : "la capacité d'un système, conforme à sa spécification nominale, à adopter un comportement acceptable en présence d'aléas". Notre approche se fonde sur l'analyse du comportement du système face à des aléas*. On considérera comme aléa tout événement non prévu amenant le système à une impossibilité temporaire ou définitive d'exécuter une action. Les contributions principales de ce document sont brièvement présentées ci-dessous :*

(1) Proposition d'un cadre formel comportant une approche, une relation de robustesse et méthode pour générer les cas de test de robustesse d'un système modélisé sous forme d'IOLTS. Notre approche comporte deux méthodes :
*La méthode TRACOR (*Test de Robustesse en présence d'Aléas COntrôlables et Représentables*) consiste à vérifier la robustesse d'un système en présence d'aléas contrôlables et représentables. Cette méthode intègre les entrées invalides, entrées inopportunes, sorties acceptables et traces de blocage dans la spécification nominale, pour obtenir une* spécification augmentée*. Cette dernière servira de base pour la génération de séquences de test de robustesse.*
La méthode TRACON (Test de Robustesse en présence d'Aléas COntrôlables et Non représentables) est focalisée sur le test de robustesse en présence d'aléas contrôlables et non représentables. Elle consiste à enrichir la spécification nominale par les sorties acceptables et les traces de suspension afin d'obtenir la spécification semi-augmentée. Cette dernière servira de base pour la génération de séquences de test de robustesse.

(2) Pour formaliser la robustesse d'une implémentation vis-à-vis de la spécification augmentée (ou la spécification semi-augmentée), nous proposons une relation binaire, appelée ***Robust****, basée sur l'observation des sorties et blocages de l'implémentation après l'exécution de traces comportant les aléas.*

(3) Afin de générer les cas de test de robustesse, nous proposons une méthode basée sur un coloriage de la spécification augmentée (ou la spécification semi-augmentée) et un ensemble d'objectifs de test de robustesse.

(4) Les fondements de notre approche sont implémentés dans l'outil RTCG. Cette application offre trois interfaces. La première et la deuxième permettent d'automatiser la méthode TRACOR. La troisième interface permet d'automatiser la méthode TRACON.

(5) Une étude de cas, sur les protocoles SSL handshake et TCP, montrant une évaluation pratique de notre approche.

Mots clés : *Test de robustesse, Protocoles, Système de transitions étiquetées à entrée/sortie, Cas de Test de robustesse, Objectif de Test de robustesse, Aléas, Spécification nominale, Spécification augmentée, Spécification Semi-augmentée.*

Abstract

In the telecommunication field, protocols have to be seriously validated before their startup. Thus, it is necessary to test the conformance of a protocol, but it is also important to test its robustness in presence of unexpected events.

Although a precise definition of robustness is somewhat elusive, functionally the meaning is clear : the ability of a system, conform to its nominal specification, to function in an acceptable way in the presence of hazards. The term "hazards" will be used to gather faults and stressful conditions.

The aim of this document is to provide a formal framework for robustness testing for Internet protocols. In order to decide the robustness of an Implementation Under Test (IUT), a clear criterion is needed, taking into account the system behaviors in the presence of hazards. The main contributions of this document are :

(1) A framework for robustness testing including an approach, a formal definition of robustness and a test generation method from system specification modelled by an IOLTS. Our approach consists in two methods :

The method TRACOR deals with robustness testing in the presence of controllable and representable hazards. It consists in enriching the nominal specification (i.e. protocol standard specification) with invalid inputs, inopportune inputs, acceptable outputs and suspension traces in order to get an increased specification.

The method TRACON deals with with robustness testing in the presence of controllable and unrepresentable hazards. It consists in enriching the nominal specification with acceptable outputs and suspension traces in order to obtain a half-increased specification.

(2) In order to formalize the robustness of an implementation compared to the increased specification, we propose a binary relation, called ***Robust****, based on the observation of outputs and quiescence of an implementation after executing of traces containing hazards.*

(3) In order to generate robustness test cases, we propose a specific method based on coloring of the increased specification (or the half-increased specification). This method allows to generate robustness test cases from the increased specification or the half-increased specification using robustness test purposes.

(4) The theoretical background of our approach is implemented in the RTCG tool. This application provides three main interfaces, the first one permits to obtain the increased specification, the second one permits to generate robustness test cases using the method TRACOR, and the third one automates the method TRACON.

(5) We show how to apply our approach in communicating protocols through a case study on the SSL protocol and TCP.

Keywords : Robustness, Testing, IOLTS, Protocols, Robustness Test Case, Robustness Test Purpose, Nominal specification, Increased Specification, Half-increased Specification.

Table des matières

Table des figures

Chapitre 1

Introduction

Transport, production d'énergie, secteur bancaire... Nous confions ce que nous avons de plus cher (notre vie, nos biens) à des systèmes informatiques. Parallèlement, ceux-ci deviennent de plus en plus complexes et la maîtrise de leur bon fonctionnement se heurte à un certain nombre de difficultés (par exemple, le caractère inconnu, incertain ou malveillant de l'environnement). Cependant, un dysfonctionnement, même temporaire, peut avoir de lourdes conséquences économiques, voire humaines. La vérification de tels systèmes avant leur mise en œuvre est essentielle et doit permettre de garantir la correction de l'ensemble de leurs comportements possibles. D'autre part, les paradigmes de conception à base de composants réutilisables, présents dans les nouvelles méthodes d'ingénierie, impliquent l'intégration de ces composants dans des contextes applicatifs fortement variés. La vérification de ces composants doit donc répondre à la diversité des utilisations potentielles pour, d'une part, consolider leur aptitude à réagir correctement aux sollicitations et, d'autre part, permettre l'identification explicite de leurs modes de défaillances.

La vérification du logiciel consiste à révéler les fautes de conception et d'implémentation. Ces fautes ont pu être introduites au cours de n'importe quelle phase du cycle de développement de système. Pour des raisons de coût et d'efficacité, il est important de les révéler rapidement, dès leur apparition si possible. La vérification doit être intégrée tout au long du processus de développement. Toutefois, l'absence de modèle de fautes général pose un obstacle majeur à la définition d'une technique de vérification "parfaite" : aucune des approches actuelles ne permet de garantir l'élimination de toutes les fautes résiduelles. Une bonne méthode de vérification fait alors appel à un ensemble de techniques qu'il faut choisir de façon judicieuse, en tenant compte du niveau de criticité du système.

Les techniques de vérification peuvent être classées selon qu'elles impliquent ou non l'exécution du système [123] (voir FIG. 1) : la vérification *sans exécution* est dite *statique* ; par opposition à la vérification *dynamique* qui est basée sur l'exécution du système.

L'*analyse statique* [60, 56] correspond aux techniques de *"revues"* ou *"inspections"* qui sont applicables à n'importe quel niveau de vérification. Le principe consiste en une analyse

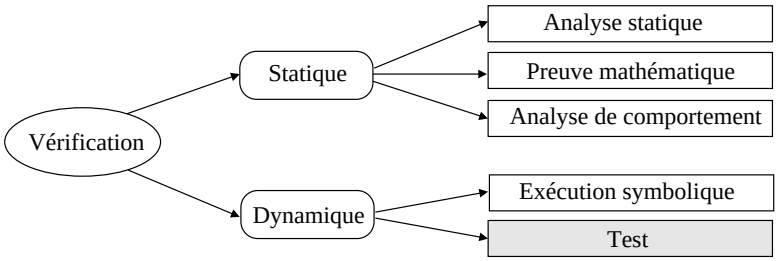


FIG. 1: Classification de techniques de vérification

détaillée du document produit (spécification, code source, etc) effectuée par une équipe différente de celle ayant réalisé le document. L'*analyse statique* peut être manuelle, ou automatique (compilateurs, analyseurs de codes, etc).

La *preuve mathématique* [122, 161] consiste à démontrer qu'un programme ou une implémentation est correct par rapport à sa spécification, à l'aide de concepts mathématiques. Dans l'état actuel, la preuve mathématique n'est utilisée que pour des systèmes critiques de petite ou moyenne taille.

L'*analyse de comportement* [76, 36] est basée sur des modèles comportementaux du système, déduits de la spécification ou de codes sources. Ces modèles doivent permettre de vérifier des propriétés de cohérence, complétude, vivacité, etc. Les plus utilisés sont les systèmes de transitions étiquetées, les machines à états finis, les réseaux de Petri, etc. Ces modèles sont bien adaptés à une décomposition hiérarchique dans le cas des systèmes complexes.

L'*exécution symbolique* [48] consiste à exécuter un système en lui soumettant des valeurs symboliques en entrée : par exemple, si une entrée X est un nombre entier positif, on peut lui affecter une valeur symbolique a qui représente l'ensemble des valeurs entières supérieures à 100. Une exécution symbolique consiste alors à propager les symboles sous formes de formules au fur et à mesure de l'exécution des instructions, et fournit comme résultat les expressions symboliques obtenues pour les sorties.

Enfin, le *test* est la méthode de vérification utilisée à travers ce document, elle sera détaillée dans la section suivante.

1.1 Test de logiciels

Dans la littérature, on y trouve une multitude de définitions de test plus ou moins différentes. Ci-dessous, nous en présentons quelques unes.

- *Le test est l'exécution ou l'évaluation d'un système ou d'un composant par des moyens automatiques ou manuels, pour vérifier qu'il répond à ses spécifications ou identifier les*

différences entre les résultats attendus et les résultats obtenus - IEEE (Standard Glossary of Software Engineering Terminology) [88].

- *Les tests représentent l'essai d'un programme dans son milieu naturel. Ils nécessitent une sélection de données de test à soumettre au programme. Les résultats des traitements de ces données sont alors analysés et confirmés. Si on découvre un résultat erroné, l'étape de débogage commence* [44].

- *Le processus de test consiste en une collection d'activités qui visent à démontrer la conformité d'un programme par rapport à sa spécification. Ces activités se basent sur une sélection systématique des cas de test et l'exécution des segments et des chemins du programme* [203].

- *L'objectif du processus de test est limité à la détection d'éventuelles erreurs d'un programme. Tous les efforts de localisation et de correction d'erreurs sont classés comme des tâches de débogage. Ces tâches dépassent l'objectif des tests. Le processus de test est donc une activité de détection d'erreurs, tandis que le débogage est une activité plus difficile consistant en la localisation et la correction des erreurs détectées* [205].

À travers ce document, nous nous appuyons sur la définition de l'IEEE. Cependant, deux problèmes sont fréquemment posés dans la pratique du test :

Sélection et génération des entrées. Sauf cas exceptionnel, le test exhaustif est impossible, et on est amené à sélectionner de manière pertinente un petit sous-ensemble du domaine d'entrée. Les méthodes actuelles de sélection des entrées peuvent être classées selon deux points de vue : sélection des entrées de test et génération des entrées de test.

La *sélection* des entrées de test : elle peut être liée à un modèle de la structure du système qui est vu comme une "boîte blanche", ou à un modèle de la fonction que doit réaliser le système vu comme une "boîte noire". Il existe de nombreux critères de test, structurels ou fonctionnels, qui guident la sélection des entrées [48, 14]. Par exemple, le critère structurel "toutes-les branches" qui exige que les entrées soient déterminées de façon à exécuter au moins une fois chaque branche du programme. Plusieurs critères fonctionnels sont possibles, qui diffèrent selon les modèles utilisés d'une part, et le type de couverture visé d'autre part (règles, états, transitions, etc.) [197].

La *génération* des entrées de test peut être, dans les différentes approches, *déterministe* ou *probabiliste*. Dans le test déterministe, les scénarii de test sont déterminés par un choix *sélectif* selon le critère considéré. En revanche, dans le test *aléatoire* ou *statistique*, les scénarii de test sont sélectionnés selon une distribution *probabiliste* du domaine d'entrée, la distribution et le nombre de données d'entrée étant déterminés selon le critère considéré [187].

Oracle du test. En pratique, le test pose un problème de mise en œuvre qu'il ne faut pas négliger : le *problème de l'oracle*, ou comment décider de l'exactitude des résultats observés, fournis par l'implémentation du système en réponse aux entrées de test ? [204]. Dans le test statistique, en particulier, la génération automatique d'un grand nombre d'entrées de test

rend indispensable le dépouillement automatique des sorties (correctes/incorrectes).

1.1.1 Les principales stratégies de test

Dans la littérature, certains articles font la distinction entre les techniques de *test ascendant* et *descendant* ou entre les techniques de *test passif* et *actif*. Un autre ensemble de termes qui nécessite une explication est la dichotomie de test structurel (boîte blanche) et le test fonctionnel (boîte noire).

Test fonctionnel versus test structurel

Test fonctionnel. Cet aspect du test s'intéresse aux besoins fonctionnels du système, il est aussi appelé test en *boîte noire*. Ce type de test comporte deux étapes importantes : l'identification des fonctions que le système est supposé offrir et la création de données de test qui vont servir à vérifier si ces fonctions sont bien réalisées par le système ou pas.
Les cas de test qui sont dérivés sans référence à l'implémentation du système, mais plutôt par référence à la spécification, ou à une description de ce que le système est supposé faire. Ils sont appelés aussi : tests en *boîte noire*. Ces tests ne sont pas une alternative aux tests en boîte blanche. Il constituent plutôt une approche complémentaire qui découvre des classes différentes d'erreurs. Dans cette approche, le système est traité comme une boîte noire accessible à travers des points de contrôle et d'observation (PCOs). La fonctionnalité du système est déterminée en injectant au système différentes combinaisons de données d'entrées. L'objectif de ces tests est la vérification de la conformité des fonctionnalités de l'implémentation par rapport à une spécification de référence. Les détails internes des systèmes ne sont pas inspectés. Ces tests ne s'intéressent qu'au comportement externe du système à tester, et ils permettent de détecter une large gamme d'erreurs comme : fonctions incorrectes ou manquantes, erreurs d'interfaces, erreurs de structures de données ou d'accès aux bases de données externes, problèmes de performance, erreurs d'initialisation ou de terminaison. Les tests en boîte noire sont généralement réalisés durant les dernières phases du test.

Test structurel. Cet aspect du test est aussi appelé test en *boîte blanche*. Plutôt que d'être fondé sur les fonctions du système, le test structurel est basé sur le détail du système (d'où son nom). Les tests structurels sont effectués sur des produits dont la structure interne est accessible et sont dérivés en examinant l'implémentation du système (style de programmation, méthode de contrôle, langage source, conception de base de données, etc). Ils s'intéressent principalement aux structures de contrôle et aux détails procéduraux. Ils permettent de vérifier si les aspects intérieurs de l'implémentation ne contiennent pas d'erreurs de logique. Ils vérifient si toutes les instructions de l'implémentation sont exécutables (contrôlabilité). Le système à tester est à comparer à une spécification de référence plus abstraite. Cependant, les connaissances sur le système à tester ainsi que sur la spécification

de référence sont utilisées pour la sélection des données de test, des critères de couverture ainsi que pour l'analyse des résultats du test.

Tests boîte grise. Parfois, le terme "*test de boîte grise*" est utilisé pour dénoter la situation où la structure modulaire du produit système à tester est connue des testeurs, c'est-à-dire lorsqu'on peut observer les interactions entre les modules qui composent le système, mais pas les détails des programmes de chaque composante. L'observation des interactions se fait sur des points spécifiques du système appelés points d'observation. Cette approche n'a donc pas besoin des détails internes des modules, mais exige l'observabilité de la structure du système via les points d'observation. La visibilité de la structure interne du système peut énormément aider lors de l'étape de débogage.

Test passif versus test actif

Test actif. De manière intuitive, le test actif veut dire qu'un testeur envoie une donnée d'entrée, dite *"input"* à l'implémentation et attend une donnée de sortie, dite *"output"*. Si le testeur constate que l'*"output"* appartient à l'ensemble des *"outputs"* attendus selon la spécification, alors le processus continue, sinon une erreur a été détectée dans l'implémentation. Ce type de test est appelé *actif* parce que le testeur a un contrôle total sur les *"inputs"* envoyés à l'implémentation sous test.

Test passif. Dans le test passif [1, 128] l'implémentation sous test s'exécute de manière autonome sans aucune interaction avec le testeur. La trace que l'implémentation exécute est observée et enregistrée pour être analysée. Dans cette dernière approche, un ensemble d'invariants (qui représentent les propriétés attendues de l'implémentation) est évalué sur les traces.

1.1.2 Test et cycle de vie du logiciel

Trois grandes phases sont généralement identifiées dans les modèles de développement du logiciel : spécification des exigences, conception et réalisation. La *spécification* décrit ce que le système doit faire. Comment le faire, relève la phase de *conception*. Le plus souvent, la spécification des exigences donne lieu à un premier document écrit en langage naturel, et on parle alors de *spécification informelle*. Elle peut ensuite être transcrite dans un langage formel, et on parle de *spécification formelle*. La phase de conception est elle-même décomposée en plusieurs étapes qui permettent de passer de la conception *générale* à la conception *détaillée* en affinant progressivement les différentes fonctions qui seront implémentées. C'est à partir des dossiers de la conception détaillé que sont écrits les programmes dans la phase de réalisation.

Dans ce processus de développement (voir Fig. 2), chaque phase produit un nouvel ensemble de documents à partir de celui fourni par la phase de l'étape précédente. Il est

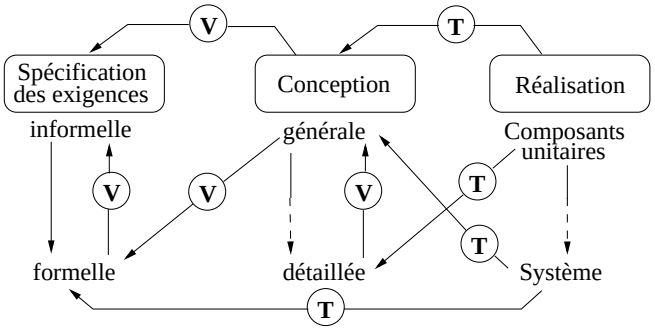


FIG. 2: Le test et la vérification dans le cycle de vie du logiciel

important de vérifier la cohérence et la conformité de ces documents avant de poursuivre le développement. A ce stade, différents niveaux de vérification sont donc indispensables. Dans la phase de réalisation, le test de type *boîte noire* intervient généralement pour valider l'implémentation par rapport à une spécification formelle dérivée à partir de la spécification informelle du système. En revanche, Le test de type *boîte blanche*intervient principalement pour valider le système ou ses composants unitaires par rapport au documents de la spécification générale ou détaillée.

1.2 Contexte : Le Test de protocoles

Un protocole de communication [83] est un ensemble de formats et de règles régissant la communication d'entités par échange de messages. La définition complète d'un protocole ressemble beaucoup à la définition d'un langage :

- Il définit un format précis pour les messages valides : *une syntaxe* ;
- Il définit les règles de procédure pour l'échange de données : *une grammaire* ;
- Il définit un vocabulaire pour les messages (appelés $UDPs$: Unités de données du protocole) valides qui peuvent être échangés, avec leur signification : *une sémantique.*

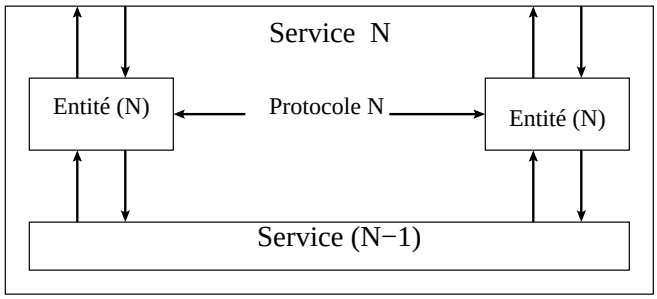


FIG. 3: Modèle de couche (N) OSI

Afin de faciliter le développement et la normalisation des protocoles de communication, ISO a développé le modèle de référence de base *OSI* (ou modèle *OSI* pour Open Systems Interconnection) [47]. Ce modèle est normalisé (ISO-7498 [95]) et offre une classification des fonctions du réseau. Cette classification est réalisée en définissant une pile de sept couches dans lesquelles différentes fonctions du réseau résident. Dans ce modèle, les fonctions qui résident dans la couche N dépendent des fonctions qui résident dans une couche, généralement la couche $N-1$, au dessous de la couche N (voir Fig. 3).

1.2.1 Les principaux tests des protocoles

A part la spécification et la normalisation de ces protocoles, la validation des implémentations des protocoles par rapport aux spécifications constitue un aspect très important pour garantir leur qualité. Plusieurs types de test ont été étudiés pour assurer le bon fonctionnement des implémentations des protocoles de communication. Les méthodes de test pour les protocoles de communication sont développées et décrites en utilisant un cadre de travail spécial pour le test. Ce cadre de travail est normalisé dans ISO (ISO 9646) [91, 89, 90, 92, 93, 94]. Il se base sur le modèle de référence de base pour l'interconnexion des systèmes ouverts (Fig. 3). Ce dernier offre une classification des fonctions du réseau ; il est aussi utilisé dans le développement et la normalisation des protocoles de communication. Nous devons noter que d'autres modèles existent. Cependant, les concepts de base ainsi que la vue générale, par rapport aux protocoles de communication, sont partagés par ces modèles.

Test de conformité [91, 89, 90, 92, 93, 94, 100, 100, 206, 155] est une approche systématique utilisée pour s'assurer que les produits vont se comporter comme prévu. Plus spécifiquement, le test de conformité joue un rôle très important dans la vérification de la conformité des produits aux standards ISO, ITU, aux recommandations et aux spécifications du vendeur. Il consiste à vérifier la conformité de l'implémentation par rapport à la spécification de référence. Ces tests se basent sur le comportement extérieur de l'implémentation. Ils consistent à appliquer un ensemble d'entrées spécifiques (cas de test) au système à tester et à vérifier la conformité de ses sorties. Généralement, un tel test ne peut pas être exhaustif vu le nombre important de cas à traiter. Les entrées sont alors sélectionnées de façon à exécuter des parties du système (fonctions, transitions, états, etc) et à détecter le maximum de fautes. Ce type de test sera détaillé dans le chapitre 3. D'autres types de test sont fortement recommandés pour s'assurer du fonctionnement correct des protocoles dans des contextes applicatifs variés.

Test d'interopérabilité [152, 116, 6, 57, 110, 141, 111, 178, 112, 117, 144, 11, 176] qui consiste à évaluer le degré d'interopérabilité de plusieurs implémentations. Il implique le test des capacités ainsi que du comportement de l'implémentation dans un environnement

inter-connecté. Il permet aussi de vérifier si une implémentation est capable de communiquer avec une autre implémentation de même type ou de type différent. Les deux approches de test d'interopérabilité mentionnées dans le contexte de l'*OSI* sont le test actif et le test passif [68, 86]. Le test actif permet la génération d'erreurs contrôlées et une observation plus détaillée de la communication, alors que le test passif permet uniquement de vérifier si le comportement observé est valide ou non.

Test de performance vise à tester la performance d'un composant du réseau face à des situations normales et parfois extrêmes [180]. Il identifie les niveaux de performance du composant du réseau pour différentes valeurs des paramètres et mesure la performance du composant. Une suite de tests de performance décrit précisément les caractéristiques de performance qui ont été mesurées ainsi que les procédures montrant la manière d'exécuter ces mesures. Ce genre de tests est très utilisé dans les systèmes à temps réel.

1.2.2 Besoins pour le test robustesse

De façon générale, les réseaux de transmission de données assurent des services de transmission qui ne sont pas systématiquement fiables. En effet, à différents niveaux d'un réseau, des unités de données peuvent être perdues ou détruites lorsque des erreurs de transmission surviennent, lorsque des éléments d'interconnexion tombent en panne ou encore lorsque l'ensemble du trafic à écouler ne peut pas être pris en charge. Certains types de réseaux, en particulier ceux utilisant des techniques de routage dynamique, peuvent dupliquer des paquets, altérer leur séquence et, éventuellement, entacher leur transmission de retards importants. De plus, dans un réseau d'interconnexion, les réseaux sous-jacents peuvent imposer une taille maximale aux unités de données ou des temps d'attente dans des files dans le but d'améliorer les politiques de service. Enfin, au plus haut niveau, des programmes d'application peuvent avoir besoin de transmettre de gros volumes de données entre différents ordinateurs qui, par des traitements inappropriés, peuvent engendrer des phénomènes nuisibles au bon fonctionnement de services du réseau.

Dans ces conditions, il est très important de valider l'implémentation du protocole, avec des tests de conformité, afin de s'assurer de son fonctionnement correct dans les conditions nominales et de contrôler son fonctionnement, par des *tests de robustesse*, en dehors de ses conditions nominales (conditions de stress, entrées invalides ou fautes).

Le *test de robustesse* a pour but la vérification de certains aspects du comportement d'un système. Il constitue un domaine d'étude relativement récent aussi sa définition est encore en cours d'évolution, donc imprécise et sujette à interprétation. Dans la littérature, cette notion est quelquefois confondue avec celle de tolérance aux fautes ou avec celle de conformité. Deux définitions ont été proposées :

Définition 1 *L'IEEE définit la robustesse comme " le degré selon lequel un système, ou un composant, peut fonctionner correctement en présence d'entrées invalides ou de conditions*

environnementales stressantes " [IEEE Std 610.12-1990][88].

Définition 2 *Suivant AS23 [33], le test de robustesse vise à vérifier "la capacité d'un système, ou d'un composant, à fonctionner de façon acceptable, en dépit d'aléas". Cette définition a été adoptée par les laboratoires participant à l'action spécifique 23 du CNRS (LaBRI, LAAS, IRISA, LRI et Verimag).*

C'est à ce type de test que nous allons nous intéresser à travers ce document pour mieux éclaircir sa définition, son positionnement et surtout comment générer des tests qui exhibent les aléas.

1.3 Contributions

Afin de maîtriser les coûts et améliorer la qualité de tests, des recherches sont menées afin d'automatiser le processus du test. L'automatisation du test de robustesse impose la formalisation mathématique de certaines étapes du test. Par conséquent, la spécification doit être formelle, afin que les comportements qu'elle décrit soient suffisamment précis et non ambigus. D'autres aspects doivent aussi être formalisés comme la notion de robustesse entre l'IUT et sa spécification, les interactions entre cas de test et l'IUT, l'observation liée à cette exécution et les verdicts émis par le cas de test. Dans ce document, nous définissons un cadre formel pour le test de robustesse. Ce cadre couvre les aspects suivants :

1.3.1 Représentation des aléas

Dans notre approche, on considère comme *"aléa"* tout événement non décrit dans la spécification nominale du système. Cet événement peut provoquer une impossibilité définitive ou temporelle du système à exécuter une action. Du point de vue du test de robustesse, on ne s'intéresse pas aux aléas, mais plutôt à leur influence prévisible sur les entrés/sorties du système. En conséquence, l'influence d'aléas sur les entrées/sorties peut être représentable (entrées, sorties) ou non. Plus précisément, pour les systèmes réactifs (par exemple les protocoles de communication), on peut identifier trois classes d'influence d'aléas. Les *entrées invalides*, les *entrées inopportunes* ou les *sorties imprévues*. Ces classes d'influence seront définies ultérieurement.

Le comportement acceptable en présence d'aléas devrait être donné par les concepteurs du système. Intuitivement, les concepteurs doivent compléter ou augmenter la spécification afin de prendre compte des aléas. Il s'agit généralement de certains comportement communs pour plusieurs états du système (par exemple, des fermetures de connexions, redémarrages des sessions). Afin de simplifier cette tâche, nous proposons une structure appelée *meta-graphe* permettant de décrire le comportement acceptable concernant un ensemble d'états du système et/ou un ensemble d'aléas.

1.3.2 Cadre formel pour le test de robustesse

Dans ce document, nous avons décidé de considérer la robustesse comme un aspect plus large que celui de la conformité. Par conséquent, seuls les systèmes conformes à leur spécification seront des sujets de vérification par le test de robustesse. L'implémentation sous test (IUT) est considérée comme une boîte noire (sa structure interne est inconnue).

Partant de la classification précédente d'aléas, nous proposons deux méthodes, l'une complète l'autre, pour formaliser le processus du test de robustesse des systèmes réactifs modélisables par des systèmes de transitions étiquetées à entrée/sortie (IOLTS pour Input Output Labelled Transition System).

Méthode TRACOR. Cette méthode (*Test de Robustesse en présence d'Aléas COntrôlables et Représentables*) est complètement formelle, elle est fondée sur l'intégration des entrées invalides, entrées inopportunes et sorties acceptables dans le modèle de la spécification nominale afin de décrire proprement le comportement acceptable en présence d'aléas. Le modèle obtenu est appelé la *spécification augmentée*. Elle servira désormais comme une référence pour la génération et l'évaluation des tests de robustesse. Les entrées invalides, les entrées inopportunes et les sorties acceptables sont modélisables par des graphes spécifiques appelés *meta-graphes*.

L'exécution du test de robustesse, selon cette méthode, consiste à interagir avec l'IUT (implementation under test), à travers des points de contrôle et d'observation, par l'émission des séquences d'entrées (combinaisons entre les entrées valides, entrées invalides, entrées inopportunes). Les sorties et les blocages de l'IUT sont comparés avec ceux de la spécification augmentée. Les verdicts obtenus sont : *Acceptable* si le comportement est robuste, *Fail* si le comportement est non robuste et, *Inconclusive* si le comportement est correct mais n'est pas visé par l'objectif de test utilisé. Les fondements de cette méthode sont publiés dans [166, 163].

Méthode TRACON. Cette méthode (*Test de Robustesse en présence d'Aléas COntrôlables et Non représentables*) propose un cadre formel pour générer des séquences de test de robustesse en présence d'aléas contrôlables et non représentables, c'est à dire les aléas physiques dont l'influence sur les entrées/sorties du système est non représentable, voire imprévisible. Dans ce cas, on suppose que les aléas sont commandables par un *contrôleur* matériel ou logiciel (par exemple, interrupteur pour activer et désactiver l'émission des radiations électromagnétiques). Les différentes phases de cette méthode sont : la construction de la *spécification semi-augmentée* et la génération des tests de robustesse.

L'exécution du test de robustesse consiste à interagir parallèlement avec l'IUT par l'émission des séquences d'entrées valides et, avec le contrôleur d'aléas HC pour activer ou désactiver l'aléa physique. Les sorties observées sont comparées à celles de la spécification semi-augmentée. Les verdicts obtenus sont : *Acceptable* si le comportement est robuste, *Fail* si le comportement est non robuste et, *Inconclusive* si le comportement est correct

mais n'est pas visé par l'objectif de test utilisé.

Génération des tests de robustesse. Afin de déduire des verdicts concernant la robustesse d'une implémentation vis-à-vis de la spécification augmentée (ou la spécification semi-augmentée), nous proposons une relation binaire appelée "***Robust***" [167]. Cette relation est basée sur l'observation des sorties et de blocages de l'IUT après l'exécution d'une trace augmentée. Par ailleurs, "***Robust***" exclut toutes les traces nominales qui ont été vérifiées durant le test de conformité.

Partant de la spécification augmentée et la relation ***Robust***, nous allons adapter l'approche de génération développée par l'IRISA et Verimag [63, 109, 107] pour la génération des cas de test de robustesse. En plus du produit synchrone, notre nouvelle méthode est basée sur un coloriage spécial de la spécification augmentée (ou la spécification semi-augmentée), un algorithme de réduction et un algorithme de sélection à la volée privilégiant les traces augmentées. Les fondements de cette partie sont publiés dans [164].

Pour ce qui concerne la méthode TRACON, nous appliquons d'abord les mêmes étapes comme dans la méthode TRACOR sur la spécification semi-augmentée et un objectif de test de robustesse. Le résultat est un cas de test de robustesse non contrôlable. Pour satisfaire la contrôlabilité, nous procédons à une deuxième synchronisation avec le contrôleur d'aléas afin d'obtenir le graphe des cas de test de robustesse contrôlables. Enfin, nous appliquons à nouveau un algorithme de sélection aléatoire pour obtenir un cas de test de robustesse contrôlable.

1.3.3 Outil et étude de cas

Outil RTCG. Les fondements théoriques des méthodes TRACOR et TRACON ont été implémentés dans l'outil RTCG (*Robustness Test Cases Generator*).

RTCG fournit deux interfaces pour automatiser la méthode TRACOR :
La première permet d'assister l'augmentation de la spécification d'un système à partir d'un graphe d'aléas et, optionnellement, d'un autre graphe d'entrées inopportunes.
La deuxième interface permet, grâce à une stratégie de coloriage, de générer des cas de test de robustesse.

De plus, RTCG fournit une troisième interface pour automatiser la méthode TRACON. Tout d'abord, RTCG construit une spécification semi-augmentée à partir de l'automate suspendu de la spécification nominale et d'un graphe de sorties acceptables. Ensuite, il calcule le produit synchrone, le graphe réduit du test de robustesse, et dérive un cas de test de robustesse non contrôlable. Enfin, RTCG procède à une synchronisation du cas de test avec un contrôleur d'aléas. Le résultat est un cas de test de robustesse contrôlable.

Les spécifications sont décrites en SDL ou en DOT , et les cas de test de robustesse sont formatés en TTCN-3, XML ou DOT. De plus, RTCG utilise l'outil *Graphviz* de *AA&T* [53] pour visualiser les résultats sous forme des graphes.

Étude de cas. Les protocoles SSL handshake et TCP sont utilisés comme des exemples concrets afin d'évaluer notre approche. A travers cette évaluation, nous montrons d'abord quelques exemples sur les aléas prévisibles dans le domaine des protocoles. Ensuite, nous donnons quelques résultats démonstratifs concernant l'application de l'outil RTCG. Les résultats de cette évaluation sont publiés dans [165, 166, 163].

1.4 Plan du document

Nous décomposons le document en deux parties : la première partie présente les méthodes et les sémantiques formelles utilisées dans le test de conformité. Cette partie comporte les chapitres suivants : Le chapitre 2 rappelle les différents modèles formels utilisés dans le test. Le chapitre 3 présente la norme ISO 9646 concernant le test de conformité et, définit l'essentiel de sa terminologie. Ensuite, il propose un état de l'art sur les différentes approches développées pour la génération des tests de conformité.

La deuxième partie propose un cadre formel pour le test de robustesse suivi par une mise en oeuvre et deux études de cas. Cette partie comporte les chapitres suivant : Le chapitre 4 propose un état de l'art sur les méthodes développées pour le test de robustesse. Le chapitre 5 détaille le cadre formel proposé. Le chapitre 6 explique notre technique pour la génération des cas de test de robustesse. Le chapitre 7 présente l'outil RTCG et, propose une étude de cas sur les protocoles SSL handshake et TCP. Enfin, le chapitre 8 conclut le document et expose les nouvelles perspectives.

Chapitre 2

Modèles formels

La modélisation mathématique des systèmes informatiques est devenue de plus en plus une nécessité méthodologique, notamment pour le développement et la validation des systèmes complexes. Au fil des années, un bon nombre de modèles a été introduit pour décrire les composants électroniques ou les systèmes informatiques. Nous n'allons pas tous les citer dans ce document, mais nous allons plutôt essayer de se concentrer sur les modèles les plus récemment utilisés pour la validation des protocoles de communication. Rappelons que pour qu'un système soit parfaitement utilisable, il doit être basé sur des fondements mathématiques, et par conséquent sur une sémantique correcte. Le choix du modèle dépend du type du système à modéliser et des conditions de son évolution. Un bon modèle doit être ouvert pour accepter des nouvelles contraintes de modélisation.

Dans ce chapitre, nous rappelons les concepts de base relatifs aux modèles introduits pour modéliser les systèmes discrets (non-temporels) : Machines à états finis, Systèmes communicants, Systèmes de transitions et langages de spécification. Bien entendu, la liste est incomplète si on parle d'autres formalismes qui ne seront pas utilisés à travers ce document. A titre d'exemple, on peut citer les réseaux de Petri [147, 148] utilisant les notions : places, transitions, flux et jetons pour modéliser et analyser les comportements dynamiques d'un système discret ou, les algèbres de processus : ACP de Bergstra et al. [16], CSP de Tony Hoare [84], CCS de Robin Milner [136, 139], MEIJE de R. de Simone [51] et LOTOS [27, 129]. Dans ces algèbres, la spécification du système contient un certain nombre de *processus* élémentaires et un certain nombre d'*opérations* qui, appliquées à des processus, construisent d'autres processus. À chaque processus, il est possible de construire un système de transitions représentant son comportement.

Le modèle des systèmes de transitions à entrées-sorties sera le modèle de base de nos travaux.

2.1 Machines à états finis

Les machines à états finis (en anglais FSM pour *Finite State Machine*) sont l'un des modèles les plus anciens. Elles sont issues de la modélisation des circuits logiques. Une FSM est une machine ayant une quantité finie de mémoire pour représenter les états et, qui distingue entre une sortie (signal émis) et une entrée (signal reçu).

Il existe deux types de machines à états finis séquentielles : machine de *Moore* et machine de *Mealy*. Dans une machine de *Moore*, les sorties dépendent des états seulement, tandis que dans une machine de Mealy les sorties dépendent des états et des entrées. Une théorie complète sur les $FSMs$ peut être trouvée dans [70]. Nous rappelons ici les notions de base relatives aux machines de Mealy.

Machine de Mealy. C'est un graphe décrivant le comportement de la spécification. Les *noeuds* décrivent les états internes du système. Les *arcs* (transitions) sont étiquetés par un ensemble de symboles d'entrée et un ensemble de symboles de sortie. Les sorties changent immédiatement en réaction à un changement d'entrée. Formellement :

Définition 3 (FSM) *Une machine de Mealy est un 6-uplet* $(s_0, S, I, O, \delta, \gamma)$ *où :*

1. s_0 *est l'état initial,*
2. S *est un ensemble fini d'états,*
3. I *est un alphabet fini d'actions d'entrée,*
4. O *est un alphabet fini d'actions de sortie,*
5. $\delta : S \times I \mapsto S$ *est la fonction de changement d'états, qui associe à une entrée* i *dans un état* s*, l'état destination* $s' = \delta((s, i))$*,*
6. $\gamma : S \times I \mapsto O$ *est la fonction de sortie, qui associe à une entrée* i *dans un état* s*, la sortie correspondante* $o = \delta((s, i))$*.* □

Par convention, une transition est notée $s \xrightarrow{i/o} s'$. Si l'entrée i est reçue alors que le système est dans l'état s, la sortie o est produite et le nouvel état du système est s'. Dans ce cas, i est aussi appelé le *déclencheur* (trigger). Si une entrée est reçue et qu'aucune transition n'est définie pour cette entrée, il ne se passe rien (l'entrée est ignorée).

Exemple 2.1 *La* FIG.*4 illustre un exemple simple d'une machine de Mealy. L'alphabet d'entrée et de sortie de cette machine est* $\{1, 0\}$*.* □

Une machine de Mealy $M = (s_0, S, I, O, \delta, \gamma)$ est dite *déterministe* si pour tous les états $s, s', s'' \in S$, pour toutes les entrées $i, i' \in I$ et toutes les sorties $o, o' \in O$, si $s \xrightarrow{i/o} s'$, $s \xrightarrow{i'/o'} s''$ et $s' \neq s''$ alors $i \neq i'$.

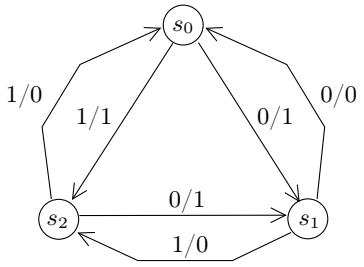


Fig. 4: Machine de Mealy.

Sémantique d'une machine de Mealy. La sémantique d'une machine de Mealy $M = (s_0, S, I, O, \delta, \gamma)$ est la fonction $g_M : I^+ \mapsto O$ définie par les équations récursives suivantes, avec d_M une fonction auxiliaire, $i \in I$ et $t \in I^*$.

$$d_M(\varepsilon) = s_0, \quad d_M(t.i) = \delta((d_M(t), i)), \quad g_M(t.i) = \gamma((d_M(t), i))$$

Le modèle de base FSM a été étendu de plusieurs façons : $CFSM$ qui modélise les communications par des canaux, $EFSM$ qui enrichi le modèle de base par les variables, paramètres et prédicats [70] et $CEFSM$ qui combine entre les deux extensions précédentes.

2.2 Systèmes communicants

Une FSM communicante ($CFSM$)[83] peut être définie comme une machine abstraite qui accepte des données d'entrée, génère des données de sortie et change son état interne selon un plan pré-défini. Les machines FSM communiquent via des files FIFO bornées qui transforment les données de sortie d'une machine en données d'entrée d'une autre. Définissons tout d'abord formellement le concept de file.

Une file à messages est un triplet (S, N, C), où :
- S est un ensemble fini appelé *vocabulaire* de la file ;
- N est un entier qui définit le *nombre de places* dans la file ;
- C est le *contenu de la file*, un ensemble ordonné d'éléments de S.

Les éléments de S et de C sont appelés des messages. Chaque message porte un nom unique mais représente un objet abstrait. Soit M l'ensemble de toutes les files, un exposant $1 \leq m \leq \mid M \mid$ est utilisé pour identifier chaque file, et un indice $1 \leq n \leq \mid N \mid$ est utilisé pour identifier un espace dans la file. ${C_n}^m$ est le message en position n dans la file m .

Un vocabulaire de système V peut être défini comme la réunion de tous les vocabulaires de toutes les files, plus un élément nul que l'on va représenter par ε . Avec l'ensemble de files M numérotées de 1 à $\mid M \mid$, le vocabulaire du système V est défini comme $V = \bigcup_{m=1}^{|M|} S^m \cup \{\varepsilon\}$.

Définition 4 *Une $CFSM$ est un tuple (Q, q_0, M, T) où :*
- *Q est un ensemble fini, non vide d'états ;*
- *q_0 est un élément de Q appelé l'état initial ;*
- *M est un ensemble de files ;*
- *T est une relation de transition.*

□

La relation T prend deux arguments q et a où q est l'état courant et a est une action. Jusque-là, trois types d'actions sont permises : l'entrée des données, la sortie des données et l'action nulle ε . L'occurrence des deux premiers types d'actions dépend de l'état des files. Si elles sont exécutées, elles changeront l'état d'une seule file.

La relation de transition T définit un ensemble de zéro ou plusieurs états successeurs possibles dans l'ensemble Q pour l'état courant q. Cet ensemble contiendra précisément un seul état, sauf si le non-déterminisme est modélisé. Si $T(q, a)$ n'est pas définie explicitement, on suppose que $T(q, a) = \phi$.

$T(q, \varepsilon)$ spécifie les transitions spontanées. Une condition suffisante pour que ces transitions soient exécutables est que la machine soit à l'état q.

Définition 5 *Un système communicant $\prod$ de n $CFSMs$ est un $2 \times n$ tuple $(C_1, C_2, ..., C_n, F_1, F_2, ..., F_n)$ où*
- *C_i est une $CFSM$;*
- *F_i est une file d'attente FIFO pour $C_i, i = 1..n$.*

□

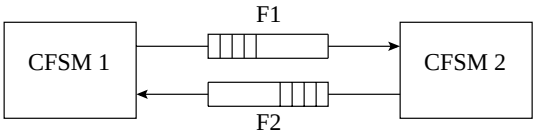


FIG. 5: Exemple de système communicant formé de deux $CFSMs$

Supposons que le système consiste en n $CFSMs$ communicantes : $C_1, C_2, ..., C_n$. Les $CFSMs$ échangent des messages à travers des files d'attente FIFO de capacité finie. Nous supposons qu'une FIFO existe pour chaque $CFSM$ et que tous les messages vers cette $CFSM$ passent à travers cette FIFO. Nous supposons aussi dans ce cas qu'un message interne identifie son émetteur et son récepteur. Un message en entrée peut être interne s'il est envoyé par une autre $CFSM$ ou externe s'il arrive de l'environnement.

Les états accessibles de Π constituent les noeuds du graphe d'accessibilité de Π. Ce dernier est noté R_Π et la procédure pour le construire est appelée analyse d'accessibilité.

Le modèle obtenu à partir d'un système communicant par analyse d'accessibilité est appelé un modèle global. Ce modèle est un graphe orienté $G = (V, E)$ où V est un ensemble d'états globaux et E correspond à l'ensemble de transitions globales.

2.3 Systèmes de transitions

Tout système réel (par exemple un programme) peut se décrire par un système de transitions, c'est à dire un ensemble quelconque d'états et de transitions étiquetées par des actions entre les états. Lorsque l'on se trouve dans l'un des états du système de transitions, il est possible de changer d'état en effectuant une action qui étiquette l'une des transitions sortantes de cet état.

2.3.1 Système de transitions étiqueté (LTS)

Un Système de transitions étiqueté (LTS) (*Labelled Transition System*) est défini en terme d'états et transitions étiquetées entre états, où les étiquettes indiquent le comportement produit durant la transition. Formellement :

Définition 6 (LTS) *Un LTS [8, 9, 29] est un quadruplet* $(Q, q_0, \Sigma, \rightarrow)$ *tel que :*
- Q *est un ensemble dénombrable d'états,*
- $q_0 \in Q$ *est l'état initial,*
- Σ *est un alphabet dénombrable d'actions,*
- $\rightarrow \subset Q \times \Sigma \cup \{\tau\} \times Q$ *est un ensemble de transitions. Un élément* (s, a, s') *de* $\rightarrow$ *sera simplement noté* $s \xrightarrow{a} s'$. □

Un *alphabet* Σ est un ensemble fini d'actions. Une *action* d'un LTS peut prendre plusieurs formes : une entrée, une sortie, un appel de méthode, etc. Les états sont souvent une abstraction d'états du système (habituellement, il est impossible de représenter tous les états du système). Les actions de Σ sont dites actions *observables* . Une transition peut être étiquetée soit par une action observable, ou par une action interne (inobservable, ou encore silencieuse) τ.

Une *séquence* ou *chemin* est une suite finie d'actions que l'on note simplement par juxtaposition : $\sigma = a_1 a_2 \cdots a_n,\ a_i \in \Sigma$. L'ensemble des séquences finies et non vides de Σ est noté Σ^+. La séquence vide est noté ε. Σ^* dénote l'ensemble des séquences de Σ, c'est à dire l'ensemble $\Sigma^+ \cup \{\varepsilon\}$. De plus, τ désigne une action qui n'appartient pas à Σ. τ est dite action *silencieuse* ou action *interne.* L'ensemble $\Sigma \cup \{\tau\}$ est noté Σ_τ.

La *longueur* d'une séquence σ, notée $|\sigma|$, est le nombre d'actions qui la composent. σ_i dénote la *ième* action de σ (l'indice i est compris entre 1 et $|\sigma|$). Le *produit* de concatenation de deux séquences $\sigma_1 = a_1 a_2 \cdots a_n$ et $\sigma_2 = b_1 b_2 \cdots b_m$ est la séquence σ :

$$\sigma = a_1 a_2 \cdots a_n b_1 b_2 \cdots b_m$$

Bien entendu, on a $\varepsilon.\sigma = \sigma.\varepsilon = \sigma$ et $|\sigma_1.\sigma_2| = |\sigma_1| + |\sigma_2|$.

Une *exécution* dans un système de transitions est alors une séquence d'actions $(a_i)_{i\in[1,n]}$ notée $a_1 \cdots a_n$.

Notations standards des LTSs

Soient $S = (Q, q_0, \Sigma, \rightarrow)$ un LTS, $a, \mu_{(i)} \in \Sigma \cup \{\tau\}$ un ensemble d'actions, $\alpha_{(i)} \in \Sigma$ un ensemble d'actions observables, $\sigma \in \Sigma^*$ une séquence d'actions observables et $q, q' \in Q$ deux états. Les notations standards des LTSs sont rappelées ci-dessous :

- $q \xrightarrow{a}$ $\triangleq$ $\exists\, q' \,|\, q \xrightarrow{a} q'$.
- $q \xrightarrow{\mu_1 \ldots \mu_n} q'$ $\triangleq$ $\exists\, q_0 \ldots q_n \,|\, q = q_0 \xrightarrow{\mu_1} q_1 \xrightarrow{\mu_2} \ldots \xrightarrow{\mu_n} q_n = q'$.
- $q \xrightarrow{\alpha} q'$ $\triangleq$ $si\ \alpha = \mu_1 \ldots \mu_n\ \ alors\ \ \exists\, q_0 \ldots q_n \,|\, q = q_0 \xrightarrow{\mu_1} q_1 \xrightarrow{\mu_2} \ldots \xrightarrow{\mu_n} q_n = q'$.

Le comportement observable de S est décrit par la relation $\Rightarrow$:

- $q \overset{\varepsilon}{\Rightarrow} q'$ $\triangleq$ $q = q'$ ou $q \xrightarrow{\tau \ldots \tau} q'$.
- $q \overset{\alpha}{\Rightarrow} q'$ $\triangleq$ $\exists\, q_1, q_2 \,|\, q \overset{\varepsilon}{\Rightarrow} q_1 \xrightarrow{\alpha} q_2 \overset{\varepsilon}{\Rightarrow} q'$.
- $q \overset{\alpha_1 \ldots \alpha_n}{\Rightarrow} q'$ $\triangleq$ $\exists\, q_0 \ldots q_n \,|\, q = q_0 \overset{\alpha_1}{\Rightarrow} q_1 \overset{\alpha_2}{\Rightarrow} \ldots \overset{\alpha_n}{\Rightarrow} q_n = q'$.
- $q \overset{\sigma}{\Rightarrow}$ $\triangleq$ $\exists\, q' \,|\, q \overset{\sigma}{\Rightarrow} q'$.
- $q\, after\, \sigma$ $\triangleq$ $\{q' \in Q \,|\, q \overset{\sigma}{\Rightarrow} q'\}$, l'ensemble des états atteignables à partir de q par σ. Par extension, S after $\sigma \triangleq q_0$ after σ.
- $Traces(q)$ $\triangleq$ $\{\sigma \in \Sigma^* \,|\, q \overset{\sigma}{\Rightarrow}\}$, l'ensemble des séquences observables tirables à partir de q. L'ensemble des séquences observables de S est $Traces(S) \triangleq Traces(q_0)$.

2.3.2 Système de transitions à entrée/sortie.

La distinction entre les entrées et les sorties nécessite un modèle plus riche. Plusieurs modèles ont été proposés dont les automates à entrée/sortie (IOA pour Input Output Automata) de Lynch [132], les automates à entrée/sortie (IOSM pour Input Output State Machines) de Phalippou [149] et les systèmes de transitions à entrée/sortie (IOLTS Input Output Labelled Transition Systems) de Tretmans [190]. Contrairement aux machines de Mealy dont les transitions portent à la fois une entrée et une sortie, les transitions de ces modèles sont soit des entrées, soit des sorties, soit des actions internes. Ces modèles présentent beaucoup de similarité ainsi que les travaux qui en ont découlé. Nous nous intéressons ici principalement au modèle des IOLTSs.

Système de transitions étiquetées à entrée/sortie. Les IOLTSs sont simplement des LTSs où l'on distingue deux types d'actions observables : les entrées et les sorties.

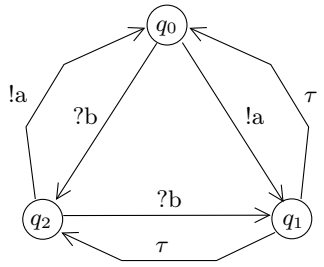


Fig. 6: IOLTS S.

Formellement,

Définition 7 (IOLTS) *Un IOLTS $S = (Q, q_0, \Sigma, \rightarrow)$ est un LTS dont l'alphabet Σ est partitionné en deux sous ensembles $\Sigma = \Sigma_O \cup \Sigma_I$, avec Σ_O l'alphabet de sortie et Σ_I l'alphabet d'entrée.* □

L'ensemble des IOLTSs définis sur Σ est noté $\mathcal{IOLTS}(\Sigma)$ ou encore $\mathcal{IOLTS}(\Sigma_I, \Sigma_O)$. Une entrée $a \in \Sigma_I$ est notée $?a$ et, une sortie $a \in \Sigma_O$ est notée $!a$.

Exemple 2.2 *La Fig.6 illustre un exemple d'un IOLTS. Il est composé de trois états et six transitons. L'alphabet Σ correspond à $\{!a, ?b\}$ avec $\Sigma_I = \{?b\}$ et $\Sigma_O = \{!a\}$.* □

Notations des IOLTSs. En plus des notations standards des LTSs, nous utiliserons les notations suivantes pour les IOLTSs :

- $Out(q) \quad \stackrel{\triangle}{=} \quad \{a \in \Sigma_O \mid q \stackrel{a}{\Rightarrow}\}$, l'ensemble des sorties possibles de q.
- $Out(P) \quad \stackrel{\triangle}{=} \quad \{Out(q) \mid q \in P\}$, l'ensemble des sorties possibles de $P \subseteq Q$.
- $Out(S, \sigma) \quad \stackrel{\triangle}{=} \quad Out(S \text{ after } \sigma)$.

Exemple 2.3 *Prenons encore la Fig.6.*

- $q_0 \stackrel{!a}{\Rightarrow} q_1 \stackrel{\triangle}{=} q_0 \stackrel{!a}{\rightarrow} q_1$, $q_0 \stackrel{?b}{\Rightarrow} q_2 \stackrel{\triangle}{=} q_0 \stackrel{?b}{\rightarrow} q_2$,
- $q_1 \stackrel{!a}{\Rightarrow} q_0 \stackrel{\triangle}{=} q_1 \stackrel{\tau}{\rightarrow} q_2 \stackrel{!a}{\rightarrow} q_0$, $q_1 \stackrel{!a}{\Rightarrow} q_1 \stackrel{\triangle}{=} q_1 \stackrel{\tau}{\rightarrow} q_0 \stackrel{!a}{\rightarrow} q_1$, $q_1 \stackrel{?b}{\Rightarrow} q_1 \stackrel{\triangle}{=} q_1 \stackrel{\tau}{\rightarrow} q_2 \stackrel{?b}{\rightarrow} q_1$,
- $q_2 \stackrel{!a}{\Rightarrow} q_0 \stackrel{\triangle}{=} q_2 \stackrel{!a}{\rightarrow} q_0$, $q_2 \stackrel{?b}{\Rightarrow} q_1 \stackrel{\triangle}{=} q_2 \stackrel{?b}{\rightarrow} q_1$,

Ainsi, on peut déduire que :

- $Out(q_0) = Out(q_1) = Out(q_2) \stackrel{\triangle}{=} \{!a\}$,
- $Out(S, !a) = Out(S, ?b) \stackrel{\triangle}{=} \{!a\}$,
- $Out(S, ?b.!a) = Out(S, !a.?b) \stackrel{\triangle}{=} \{!a\}$. □

Soit $S = (Q, q_0, \Sigma, \rightarrow)$ un IOLTS. S est dit :

- *déterministe* si aucun état n'admet plus d'un successeur par une action observable. Formellement, pour tout $a \in \Sigma$, pour tout $q \in Q$, si $q \stackrel{a}{\Rightarrow} q_1$ et $q \stackrel{a}{\Rightarrow} q_2$ alors $q_1 = q_2$
- *observable* si aucune transition n'est étiquetée par τ. Formellement, si $q \stackrel{a}{\rightarrow}$ alors $a \neq \tau$.
- *input-complet* s'il accepte toute entrée a dans chaque état. Formellement, pour tout $a \in \Sigma_I$, pour tout $q \in Q$, alors $q \stackrel{a}{\Rightarrow}$.

Exemple 2.4 *Il est facile de voir que l'IOLTS de la* FIG.*6 est :*

- *indéterministe (vu que* $q_1 \stackrel{!a}{\Rightarrow} q_1$ *et* $q_1 \stackrel{!a}{\Rightarrow} q_0$*),*
- *inobservable (*$q_1 \stackrel{\tau}{\rightarrow} q_0$*),*
- *input-complet.* □

2.4 Langages de spécification

Dans le cadre de la normalisation pour l'interconnexion des systèmes ouverts (OSI), certains groupes ont été établis avec ISO et ITU afin d'étudier la possibilité d'utiliser des spécifications formelles pour la définition des protocoles et de services de l'OSI. Leurs travaux ont abouti à la proposition des langages de spécification tels que SDL, LOTOS ou Estelle. Ces langages sont appelés *techniques de description formelles* ou $TDFs$ puisqu'il possèdent non seulement une syntaxe formelle mais aussi une sémantique formelle. Les $TDFs$ ont été développées pour assurer la clarté, la complétude, la consistance et la traçabilité des spécifications.

2.4.1 SDL

Normalisé par l'ISO[1] et le CCITT[2][105], SDL (*Specification and Description Language*) est recommandé par le ITU[3] pour spécifier les systèmes de télécommunication d'une manière non ambiguë (ITU-T standard Z.100 et Z.105). SDL est principalement connu dans le domaine des télécommunications, mais possède aussi un domaine d'application plus large. Un tutoriel complet du langage peut être trouvé dans [12, 13, 104]. Dans SDL, une spécification est vue comme une séquence de réponses à une séquence de stimuli. Le modèle de la spécification est basé sur le concept de machines à états finis communicantes étendues (CEFSM). SDL offre aussi des concepts structurés qui facilitent la spécification des systèmes larges ou complexes. Ces constructions permettent de partitionner la spécification en unités pouvant être manipulées et comprises indépendamment. Le partitionnement peut

[1]International Standardization Organisation
[2]Comité Consultatif International Téléphonique et Télégraphique
[3]International Telecommunication Union

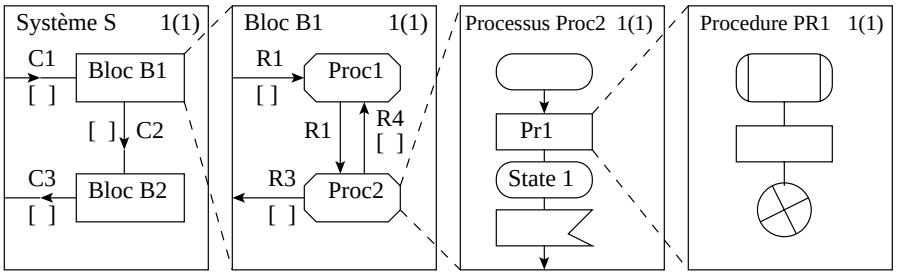


Fig. 7: Description d'un système en bloc, processus et procedure

être réalisé en suivant un certain nombre d'étapes résultant en une structure hiérarchique d'unités définissant le système à différents niveaux. La spécification du système constitue

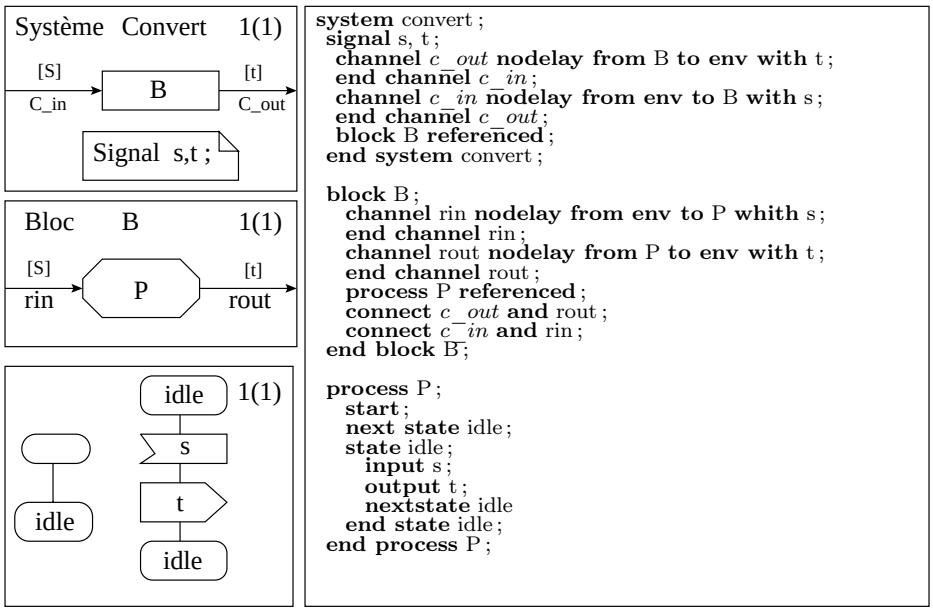


Fig. 8: Exemple d'un système en SDL/GR et SDL/PR

le plus haut niveau hiérarchique. Un *système* est une machine abstraite communicant avec son environnement via des canaux. Un *bloc* est une partie d'un système. Il est aussi composé de *processus* qui communiquent entre eux et avec les blocs à travers des *routes* et des *signaux*. Chaque processus est composé d'un ensemble de *procedures*. Dans la Fig. 7, nous présentons la description hiérarchique d'un système en SDL.

SDL est entièrement un langage Orienté-Objet, il supporte l'encapsulation, le polymorphisme et la liaison dynamique. Ainsi, SDL offre deux formes syntaxiques à utiliser pour représenter un système (voir Fig. 8) : la forme graphique (SDL/GR) et la forme textuelle (SDL/PR). Comme ces deux formes sont des représentations concrètes du même système, elle sont donc équivalentes. Chacune de ces grammaires possède une définition de sa propre

syntaxe ainsi que sa relation avec la grammaire abstraite.

2.4.2 LOTOS

LOTOS (*Language Of Temporal Ordering Specification*) est une norme internationale [96]. D'autres définitions du langage apparaissent dans [19, 69, 124, 27, 129]. La spécification LOTOS possède deux parties clairement séparées. La première offre un modèle comportemental dérivé des algèbres de processus, principalement de CCS (*Calculus of Communicating Systems*, [139]), mais aussi de CSP (*Communicating Sequential Processus*, [84]). La deuxième partie du langage permet de décrire des types de données abstraites et des valeurs, et est basée sur le langage des types de données abstraits ACT-ONE [55].

LOTOS utilise les concepts de *processus*, *événement* et *expression comportementale* comme concepts de base pour la modélisation.

```
SELL [MONEY, TEA, COFFEE, CHOCOLATE, CANCEL, CHANGE] (0, 0)
  where
 process SELL [MONEY, TEA, COFFEE, CHOCOLATE, CANCEL, CHANG(D, C :NAT) : noexit;
    MONEY?DOLLARS : NAT ,CENTS : NAT;
     (
      let DO : NAT = (D + DOLLARS) , CO : NAT = (C + CENTS) in
       ( [TOTAL (DO , CO) ge COST] ->
           (choice DRINK in [TEA, COFFEE, CHOCOLAT] [ ]
               DRINK;
                 CHANGE !CHG_DOLLARS (C0 , C0) !CHG_CENTS (D0, C0)
                     SELL [MONEY, TEA, COFFEE, CHOCOLATE, CANCEL, CHANGE] (0, 0)
           )
           [ ]
           CANCEL
              CHANGE !DO !C
                 SELL [MONEY, TEA, COFFEE, CHOCOLATE, CANCEL, CHANGE] (0, 0)
           [ ]
           SELL [MONEY, TEA, COFFEE, CHOCOLATE, CANCEL, CHANGE] (D0, C0)
       )
     )
endproc
```

FIG. 9: Spécification d'un distributeur de boissons en *LOTOS*

Les spécifications LOTOS décrivent les comportements observables des systèmes. Le comportement observable est l'ensemble de toutes les séquences d'interactions dans lesquelles le système peut participer. Ainsi, le concept d'*interaction* est fondamental dans le modèle LOTOS. Les interactions sont représentées par des *événements*. Ceux-ci sont des interactions atomiques, instantanées et synchrones. A chaque événement est associée une porte (celle où l'événement a lieu). La transmission de données à travers une interface peut être considérée comme un événement. Aussi, le début et la fin de la transmission peuvent aussi être modélisés comme des événements distincts.

Le comportement observable d'un système LOTOS est décrit par une construction du langage dans laquelle les séquences d'événements permises sont définies. Cette construction

est appelée *expression comportementale*, elle définit les expressions en termes du modèle sémantique de LOTOS. Les expressions comportementales peuvent être représentées graphiquement par des arbres de comportement. Cette représentation aide à visualiser la séquence d'événements ainsi que leur dépendance.

2.4.3 StateCharts

Le modèle des $FSMs$ est un des formalismes les plus utilisés pour spécifier des systèmes. Deux des inconvénients majeurs de ce modèle sont sa séquentialité inhérente et sa nature non-hiérarchique. Le langage *StateCharts* [75, 77, 54] a été conçu principalement pour spécifier les systèmes réactifs, qui sont dirigés par les événements et dominés par le contrôle, tels que ceux utilisés en aviation et dans les réseaux de télécommunication. Il dépasse le modèle de FSM traditionnel pour inclure trois éléments additionnels : la hiérarchie, la concurrence et la communication.

Le langage *Statecharts* a été proposé comme un formalisme visuel pour décrire les états et les transitions d'une manière modulaire, permettant le groupage, l'orthogonalité (c'est à dire la concurrence) et le raffinement, et encourageant les capacités de zoom pour se déplacer facilement en arrière et en avant entre les niveaux d'abstraction. Techniquement parlant, le noyau de cette approche est l'extension des diagrammes d'états conventionnels par des décompositions d'états en And/Or avec les transitions entre les niveaux, et un mécanisme de diffusion pour la communication entre les composants concurrents.

Le mécanisme *StateCharts* a pour but de raviver l'approche naturelle de FSM pour la spécification de systèmes, en l'étendant afin de surmonter ses difficultés. Les états dans le langage *StateCharts* peuvent être combinés d'une façon répétitive en états de plus haut niveau utilisant des AND et des OR. Les transitions dans le langage *StateCharts* ne sont pas restreintes à certains niveaux, et peuvent aller d'un état se trouvant à un certain niveau à un autre. Les données de sortie peuvent être associées aux transitions, comme dans les machines Mealy, en écrivant a/b sur une flèche ; les transitions seront déclenchées par a et entraîneront l'occurrence de l'événement b. Similairement, b peut être associé à un état, comme dans les machines de Moore. Dans les deux cas, b peut être un événement interne ou externe. L'usager peut aussi spécifier une borne inférieure et supérieure sur le moment où il faut être à un certain état.

Le langage *StateCharts* permet la hiérarchie comportementale en permettant à toute spécification d'être décomposée en une hiérarchie d'états. Ceci peut être accompli de deux manières :

- Décomposition OR (séquentielle) : un état peut être composé d'une machine à états finis comprenant des sous-états séquentiels et des transitions.
- Décomposition AND (concurrente) : un état peut aussi être composé de sous-états orthogonaux, dans lesquels tous les sous-états deviennent actifs dès que l'état père le devient.

L'inconvénient majeur du langage *StateCharts* est qu'il sépare le contrôle des données.

2.4.4 Autres...

D'autres langages ont été proposés pour modéliser les systèmes en général et les protocoles en particulier. A titre d'exemple, on peut citer :

Estelle (*Extended State Transition Language*) est la troisième technique de description formelle TDF normalisée par le CCITT et ISO (ISO9074 [97]), elle est basée sur le modèle d'automate à états-transitions finis et sur le langage de programmation $Pascal$.

UML (*Unified Modeling Language*) de l'OMG (*Object Management group*). En particulier, la version 2.0 Superstructure [195] fournit un package spécial pour les machines à états (*StateMachine package*). Ce package permet de modéliser le comportement discret d'un système à travers un LTS. Il existe plusieurs types de machines à états : machine à états comportementale (*behavioral state machine*) et machine à états protocolaire (*protocol state machine*). On peut citer d'autres profils UML, qui sont utilisés pour décrire les systèmes temps-réel (TURTLE [5], UML-RT ou RoseRT [115]).

IF (*Intermediate Format*) est un langage permettant la description d'arbres abstraits [22, 23, 21], développé au sein du Laboratoire Verimag. Ce langage est une représentation à base d'automates temporisés communicants. IF (dans sa version évoluée IF-2.0 [24]) permet de modéliser des systèmes à plusieurs composants.

Promela [20] est un langage de spécification de systèmes asynchrones. Ce qui en d'autres termes veut dire que ce langage permet la description de systèmes concurrents, comme les protocoles de communication. Il autorise la création dynamique de processus. La communication entre ces différents processus peut se faire en partageant les variables globales ou en utilisant des canaux de communication.

2.5 Conclusion

Nous venons de présenter un état de l'art concernant les modèles les plus utilisés pour spécifier les systèmes logiciels ou matériels. Les machines à état finis (FSM) sont souvent utilisées dans le domaine du matériel pour décrire le fonctionnement des circuits logiques. Dans le domaine des protocoles, les langages de spécification (SDL, Lotos, Estelle, etc), normalisés par l'ISO et ITU, sont les formalismes les plus utilisés pour écrire les spécifications. Ces langages sont généralement fondés sur la sémantique des systèmes de transitions étiquetées à entrée/sortie (IOLTS). C'est pour cette raison que nous avons décidé d'utiliser ce formalisme comme un modèle de base pour nos travaux.

Chapitre 3

Approches formelles pour le test de conformité

3.1 La norme *ISO* 9646

Le test de conformité [102] consiste à réaliser des expérimentations sur une implémentation d'un système afin d'en déduire la correction vis à vis d'une spécification de référence. C'est un test boîte noire car le code de l'implémentation n'est pas connu. Le testeur, qui simule l'environnement, interagit avec l'implémentation sous test (IUT Implementation Under Test) en exécutant des tests élémentaires, générés auparavant, appelés *cas de test*. Il observe le comportement de cette dernière à travers des interfaces appelées *Points de Contrôle et d'Observation (PCO)*. Un verdict est alors formulé en analysant les réactions de l'IUT : si pour chaque cas de test, les sorties de l'IUT coïncident avec celles attendues (c'est à dire, celles dérivées de la spécification) alors l'IUT est dite *conforme* à sa spécification ; sinon l'IUT est dite *erronée* et un processus de diagnostic est entamé pour localiser l'erreur. Le processus du test de conformité défini par la norme *ISO* 9646 est donné dans la Fig.10.

Dans le cadre de cette norme, le test de conformité est :
- *Fonctionnel* car on ne s'intéresse qu'à la correction vis à vis des fonctions réalisées par la spécification ;
- *Boîte noire* car le code de l'implémentation est inconnu.

3.1.1 Terminologie du test de conformité

Nous décrivons ci-dessous quelques termes définis dans la norme ISO 9646 [102].

Spécification, Implantation, Modèles. Une *spécification* est la description d'un comportement à l'aide d'un moyen formel. Une *implémentation* est une entité matérielle et/ou

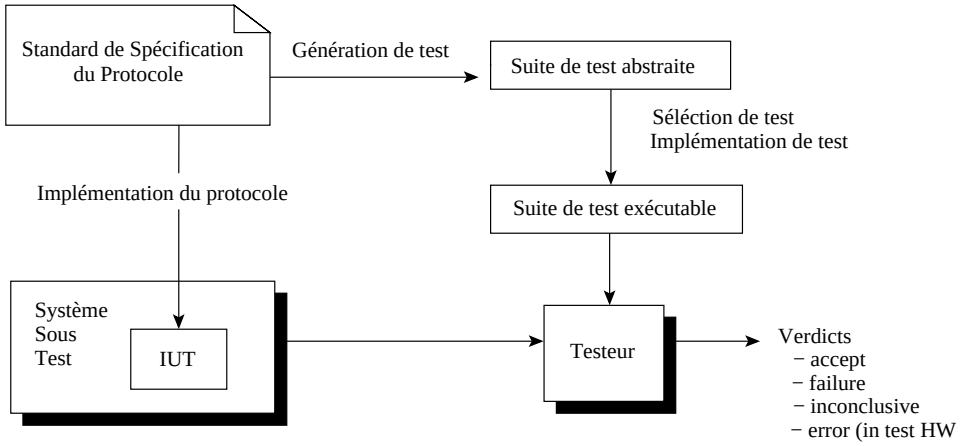


FIG. 10: Le processus du test de conformité

logicielle possédant des interfaces à travers lesquelles elle peut communiquer avec son environnement. La spécification est un objet formel, tandis que l'implémentation est physique. Afin de pouvoir lier de tels objets (de nature différente) par la conformité, on suppose (*hypothèse dite de test*) que l'implémentation peut être modélisée dans le même formalisme de la spécification. Cette hypothèse suppose l'existence de la modélisation de l'implémentation, mais à priori la description formelle de la modélisation est inconnue.

Conformité. La *conformité* est une propriété observable sur le comportement d'une implémentation. Elle est relative au fonctionnement spécifié. La conformité peut être définie au moyen d'une relation binaire. Une implémentation I est dite *conforme* à une spécification S selon la relation binaire $\boldsymbol{R}$, si l'on a $\boldsymbol{R}(I, S)$. La relation $\boldsymbol{R}$ est un moyen d'exprimer la conformité entre S et I.

Testeur. Le *testeur* est un programme exécutant des tests sur l'IUT et observant son comportement.

Architecture de test. C'est la description abstraite de la situation du testeur vis à vis de l'IUT, c'est à dire, quelles sont les interfaces contrôlables ou observables (*points de contrôle et d'observation PCOs*) par le testeur et comment s'effectue la communication entre le testeur et l'IUT. L'écriture de tests dépend étroitement de l'architecture choisie. Il existe plusieurs architectures normalisées : locale, distante, etc.

Suite de test. Une *suite de tests* est un ensemble de tests élémentaires (*cas de test*). En $TTCN$[1] (*Tree and Tabular Combined Notation*), langage de description de test issu de la norme ISO 9646, une suite de test contient plusieurs parties :

- Le *sommaire* contient la *structure*, c'est à dire comment la suite est organisée en groupes et sous-groupes et l'*index* qui décrit chaque *objectif de test* ;
- Les *déclarations* définissent les types de données, c'est à dire les $PDUs$ (*protocol data unit*) et $ASPs$ (*abstract service primitive*) , les timers, les $PCOs$, les variables de la

[1]La version 3 du TTCN : Testing and Test Control Notation

suite de test et les opérations de base ;
- Les *contraintes* définissent les valeurs des différents champs des $PDUs$ et ASP utilisées dans les cas de test ;
- Les *comportements* définissent l'ensemble des *cas de test*.

Objectif de test. L'*objectif* du test décrit de manière abstraite le but d'un cas de test, c'est à dire la propriété qu'il est supposé tester sur l'IUT. En général, un objectif découle d'une exigence de conformité décrivant une propriété que le système (la spécification et son implémentation) est censé vérifier.

Cas de test. Un *cas de test* décrit un programme constitué d'interactions entre le testeur et l'IUT, des opérations sur des temporisateurs de test et des verdicts. Il est généralement constitué d'un *préambule*, d'un *corps de test*, d'une *séquence d'identification* et d'un *postambule*.

Préambule. Il consiste en un sous-programme effectuant des interactions avec l'IUT de façon à mener cette dernière dans un état particulier permettant d'appliquer le *corps du test*.

Corps du test. C'est la partie du cas de test servant à vérifier l'objectif de test.

Identification. C'est une partie de cas de test servant à identifier l'état dans lequel se trouve l'IUT.

Postambule. Il consiste en un sous-programme effectuant des interactions avec l'IUT de façon à mener cette dernière dans son état initial afin d'enchaîner un autre cas de test.

Verdicts. L'exécution d'un cas de test conduit à un *verdict* qui est une affectation de résultats à l'expérience de test. Les verdicts sont en général : **Pass**, **Fail**, **Inconclusive**.

3.1.2 Les niveaux du test de conformité

Outre les concepts du test de conformité, la norme ISO 9646 décrit les différents niveaux ou types de test suivants :

Tests d'interconnexion de base. Ils permettent de vérifier les caractéristiques principales du protocole (les interconnexions de base), en particulier la capacité à ouvrir des connexions, transmettre des données et fermer des connexions.

Tests de capacité. Ils permettent de vérifier si toutes les possibilités statiques (externes et observables) de l'exécution sont valides par rapport aux contraintes statiques de la conformité (options, classe de protocole, etc).

Tests de comportement. Ils consistent à vérifier la conformité pendant l'exécution par rapport aux conditions d'exécution dynamiques définies dans la norme de protocole.

Tests pour résoudre la question de conformité. Ils regroupent des tests spéciaux visant des objectifs au-delà des objectifs visés par le test de comportement (par exemple, les exceptions).

3.2 Génération des tests de conformité

Afin de réduire les coûts du test, plusieurs travaux de recherche ont abordé la conception et le développement des techniques formelles pour l'automatisation du processus de la génération des tests de conformité. Toutefois, ces travaux diffèrent sur leurs ingrédients de base : la modélisation des systèmes, les algorithmes utilisés pour la génération et la sélection des cas de test, les architectures utilisées pour implémenter et exécuter les tests et les verdicts produits pas chaque méthode de test.

Les méthodes de génération formelle de test se décomposent en plusieurs catégories. On ne parlera pas ici de certaines méthodes *ad hoc* qui n'ont pas de bases formelles même si elles utilisent parfois des techniques de description formelle comme SDL. Les trois catégories auxquelles on s'intéresse, dans le domaine du test de protocoles, diffèrent sur le modèle utilisé pour décrire les systèmes et la portée du test.
Dans cette partie nous n'allons pas faire une synthèse complète sur les méthodes de génération des test de conformité, mais plutôt nous allons essayer de donner les ingrédients de base de ces méthodes. Le but de ce panorama est de comprendre la sémantique de chaque méthode afin d'en déduire une méthode adéquate pour la génération des tests de robustesse, le sujet de cette thèse.

3.2.1 Méthodes fondées sur les $FSMs$

Nous évoquons dans cette section les différents travaux de recherches menées dans le domaine de la génération des cas de test à partir des spécifications décrites par des $FSMs$. En général, les méthodes issues de ces travaux sont fondées sur l'identification des états, ou la couverture des transitions grâce à certaines séquences de test. Cependant, l'application de ces méthodes exige, quelquefois, le déterminisme, la complétude et la forte connexité du modèle de la spécification. Par ailleurs, ces méthodes permettent d'analyser le flux de contrôle des spécifications et laissent la partie "données" des systèmes non testées. Un panorama détaillé peut être trouvé dans [126, 127, 101, 42]. En général, les fautes visées par ces méthodes sont : les *fautes de sortie* (sorties additionnelles, sorties manquantes, etc), les *fautes de transfert* (états additionnels, états manquants, etc) et/ou *les fautes de sortie et de transfert*, qui présentent une combinaison entre les deux types de fautes précédents.

Tour de transition (TT)

Principe : un tour de transition TT [142] est une séquence de transitions permettant de passer par chaque transition au moins une fois. Un tour minimal peut être obtenu par le chemin d'Euler.
La méthode TT permet de détecter toutes les fautes de sorties mais peut ignorer certaines fautes de transfert car aucune vérification n'est effectuée pour contrôler l'état d'arrivée.
L'application de cette technique exige la forte connexité, la complétude et le déterminisme

de la FSM de la spécification.

Méthode de séquence de distinction (DS)

Principe : Une *séquence de distinction DS* [71] est une suite d'entrées provoquant une suite de sorties différentes pour chaque état de la FSM auquel elle est appliquée. Tout état est donc identifiable grâce à cette séquence.
La construction des séquences de test consiste en la concaténation de trois parties permettant respectivement de synchroniser la FSM à tester, de tester les transitions et d'identifier les états.
Une *séquence de synchronisation* SS est une suite d'entrées provoquant le transfert dans un état de synchronisation quel que soit le point d'application de cette séquence. Un *reset* est une séquence de synchronisation dans l'état initial.
La méthode DS est applicable à des FSM déterministes, complètement spécifiées, fortement connexes et pour lesquels il existe une DS et une SS.

Méthodes UIO, UIO_V, UIO_P

Principe : une UIO *(unique input output)* [168, 31, 183] est une séquence d'entrée et de sortie décrivant le comportement de la spécification de façon unique. Par conséquent, chaque état est identifiable grâce à cette séquence d'entrées et de sorties.
Si un état q n'a pas d'UIO, alors, on utilise une *signature*. C'est un ensemble de séquences d'entrées-sorties minimales ayant pour origine l'état q et capables de distinguer cet état des autres états de la FSM. Ces séquences sont concaténées avec les séquences de transfert permettant de ramener la FSM à l'état q après chaque séquence.
Deux processus sont applicables pour vérifier l'implémentation :

- Processus U_V qui consiste à vérifier chaque état avec l'UIO appropriée.
- Processus T_t qui consiste à vérifier les transitions ; la séquence de test amène l'implémentation dans un état particulier, applique une entrée appropriée, vérifier l'état final en utilisant l'UIO appropriée et *"reset"* l'implémentation.

Cette méthode détecte les fautes de sortie et de transfert. En revanche, elle est applicable qu'à des FSM déterministes, complètement spécifiées, minimales, fortement connexes et possédant de séquence UIO pour chaque état.

Il existe de nombreuses améliorations de la méthode UIO. Nous citons par exemple la méthode UIO avec vérification (UIO_V) [200] et la méthode UIO_p [31].

La Méthode UIO_V permet de vérifier les séquences UIO par le biais d'un processus supplémentaire UV. Il consiste en vérifier l'unicité de l'UIO pou chaque état. En d'autres termes, on vérifie pour un état q l'absence de l'UIO concernant un autre état q'. Si les sorties générées par q dans le processus U_V sont identiques à celles attendues par q' alors q' ne possède pas l'UIO attendue. Cependant, la vérification n'intervient que dans le cas où les UIO pour les états q et q' sont identiques, alors leurs sorties doivent être différentes.

La Méthode UIO_P [31] consiste à calculer uniquement pour les états qui n'ont pas de séquence UIO, plusieurs séquences partielles UIO_P permettant d'identifier ces états.

Méthodes W, W_P

Principe : Un ensemble de caractérisation W [42] est un ensemble de séquences d'entrées permettant d'identifier chaque état dans la spécification. Une séquence d'entrées W distingue deux états si, appliquée respectivement à chacun des états, elle produit des sorties différentes.
Cette méthode consiste à accéder aux différents états à l'aide d'un ensemble P de préambules couvrant les transitions de la spécification et à tenter de les distinguer les uns des autres par un ensemble W de séquence d'entrées. Le point fort de cette méthode est sa capacité de détecter les fautes de transfert et de sortie. Toutefois, il se peut qu'il n'existe pas de séquence de distinction pour chaque état.

La méthode W_P [67] est une optimisation de la méthode W et de la méthode UIO_V. Elle consiste à effectuer deux phases de test, la première a pour but de vérifier les états et la deuxième, pour vérifier les transitions.
Soit Q un ensemble de séquences permettant d'accéder à tous les états de la machine et W un ensemble de caractérisations générées par la méthode W. La première phase consiste à tester avec un ensemble de résultats de la concaténation des séquences des deux ensembles Q et W. La deuxième phase consiste à vérifier l'état dans lequel est la machine après une transition. Cependant, il suffit d'utiliser un ensemble W réduit qui permet de caractériser l'état à identifier.

3.2.2 Méthodes fondées sur les techniques de test du flux de données et/ou du flux de contrôle

La plupart des méthodes de test basées sur des langages de spécification (par exemple, SDL) ne génèrent pas directement les cas de test à partir de ces spécifications, elles les transforment plutôt en un modèle mathématique approprié (FSMs [131, 130], les EFSMs ou les graphes de flux de données GFD [196] . Cette transformation exige la normalisation (la mise en forme normale) de ces spécifications.

La normalisation [173] permet de transformer une spécification initiale, décrite en langage de spécification , en une deuxième dont chacune des transitions est accessible par un chemin unique. Quand une spécification est sous forme de EFSM, les méthodes indiquées dans la section précédente ne sont pas suffisantes, en effet, la partie "données" doit aussi être testée afin de déterminer le comportement de l'implémentation. Quelques travaux de recherches ont abordé cette problématique, parmi ceux-ci :

Sarikaya 1987 Cette méthode [172] consiste à dériver des tests vérifiant les aspects de flux de données, elle est applicable aux spécifications formelles écrites en *Estelle*.
La génération des tests est réalisée en quatre étapes :

1. Transformation de la spécification Estelle en forme normale ;
2. Construction du graphe de contrôle qui est une représentation graphique de la FSM sous-jacente ;
3. Construction du GFD *"graphe de flux de données"* ;
4. Extraction des cas de test à partir de GDF.

En utilisant des méthodes de décomposition et de partitionnement fonctionnel, les auteurs dérivent des blocs GFD qui sont considérés comme des fonctions de flux de données FFD. Pour chaque bloc, un tour est choisi à partir du GFC (*graphe de flux de contrôle*). Le tour est choisi tel que ses transitions (en forme normale) couvrent toutes les transitions du bloc fonctionnel. Par la suite, des données de test (valeurs des paramètres des données d'entrées) sont déterminées. Finalement, le tour choisi ainsi que les valeurs pour les paramètres d'entrées sont utilisés pour construire une suite de test complète.

Ural 1991 Cette méthode [197] permet la génération des séquences de tests à partir d'une spécification en forme normale NFS (*Normal Form Specification*). D'abord, la NFS est transformée en graphe de flux, ensuite, les définitions et usages de chaque variable de contexte ainsi que chaque paramètre en entrée et en sortie sont identifiés. La sélection des séquences de tests est basée sur l'identification et la couverture de chaque association entre un paramètre et une sortie (*IO-dataflow-Chains*).
Dans cette méthode, les séquences de test sont choisies pour couvrir ces associations au moins une fois. Cette méthode génère moins de séquences de test que d'autres méthodes, mais elle ne vérifie pas si les séquences sont exécutables ou non. De plus, c'est l'utilisateur qui doit construire les séquences à partir des chemins *Définition-Sortie* générés et la sélection des données de test n'est pas offerte.

Miller 1992 Cette méthode [138] utilise une version limitée d'Estelle ; elle consiste, tout d'abord, à convertir la EFSM en FSM équivalente avec des entrées-sorties modifiées. Ensuite, le GFD est construit à partir de la FSM et, les chemins qui couvrent le flux de données (critère *toutes-les-définitions-sorties*) et le flux de contrôle, sont générés et combinés afin de générer des séquences de tests exécutables.

Vuong 1992 Les auteurs de cette méthode [201, 202] suggèrent une technique hybride qui combine deux techniques de génération de cas de test pour aboutir à une séquence de test qui couvre séparément le flux de contrôle et le flux de données des protocoles spécifiés en Estelle. Le flux de contrôle est testé en premier, suivi par le flux de données qui utilise une technique de variation de paramètre. La génération et la sélection de cas de test sont guidées par des contraintes sur les suites de test ainsi que sur la variation de paramètres spécifiés par l'utilisateur.

Chanson 1994 Cette méthode [34, 35, 175] est fondée sur les techniques d'analyse de flux de données. Cette méthode utilise le critère *toutes-les-définitions-usages* pour déterminer toutes les dépendances (données et contrôles) entre les transitions. Ensuite, elle applique l'une des méthodes UIO , W ou DS pour tester le flux de contrôle. Deux inconvénients sont remarquables :

- Plusieurs séquences de tests générées par cette méthode sont éliminées du fait qu'elles ne contiennent pas la boucle influente ;
- Après la génération des chemins *def-clear*, ceux-ci sont fusionnés pour constituer une séquence de test. Cette fusion peut résulter en des séquences finales non exécutables dûes à la non exécutabilité d'un ou plusieurs chemins *def-clear*.

Huang 1995 L'auteur de cette méthode [87] propose une technique pour la génération de séquences de test exécutables pour le test du flux de données pour des protocoles modélisés par des EFSM. Pour le flux de données, le critère *toutes-définition-sorties* est utilisé. Une séquence de test exécutable est composée de trois parties : un *préambule*, un *chemin-définition-sortie* et un *postambule*.

Ramalingom 1995 Cette méthode [153] permet la génération des cas de tests pour des protocoles modélisés par des EFSMs, en utilisant les *séquences uniques indépendantes du contexte* (CIUS). Elle considère la faisabilité des cas de test pendant la génération. Un nouveau type de séquence d'identification, appelé *Trans-CIUS*, est défini. Le critère Trans-CIUS utilisé dans le test du flux de contrôle permet de générer une séquence unique d'entrée, ainsi qu'un préambule exécutable indépendant du contexte pour atteindre l'état de départ de la transition à tester, tel que les prédicats de chaque transition du préambule sont indépendants de tout contexte valide à cet état de départ. Ce critère est supérieur aux critères existants de couverture de flux de contrôle pour les EFSMs du fait, qu'un CIUS d'un certain état est suffisant pour tester les transitions aboutissant à ce dernier. Afin d'offrir une observabilité, le critère *tous-les-usages* est étendu à ce qui appelé *def-use-ob* (*ob* pour observation). Finalement, un algorithme d'exploration en largeur à deux phases est conçu pour la génération d'un ensemble de tours de test exécutables couvrant les critères mentionnés.

3.2.3 Méthodes basées sur les systèmes de transitions

Les principaux travaux de cette catégorie sont fondés sur la comparaison entre le modèle de la spécification et celui de l'implémentation à l'aide d'une relation formelle. Cette relation est dite *relation de conformité* ou *relation d'équivalence*.

Les *relations de conformité* sont un moyen de comparer deux systèmes, et comme nous le verrons par la suite, elles permettent d'exprimer ce qu'est la conformité d'une implantation par rapport à une spécification. En conséquence, une implantation est dite *conforme* si elle est en relation (de conformité) avec une spécification donnée.

Concept de la relation de conformité

"Qu'est ce que l'on entend par une implantation valide ?" La réponse à une telle question peut s'obtenir à l'aide du concept de validité par équivalence usuellement rencontré dans les techniques basées sur les algèbres de processus.

Étant données une spécification S et une relation d'équivalence R, on peut caractériser l'ensemble des implantations valides de S par l'ensemble des systèmes qui lui sont équivalents par R, c'est à dire l'ensemble :

$$\{\, I \mid I \,\boldsymbol{R}\, M \,\}$$

Diverses relations d'équivalence ont été proposées [49, 79]. Bien que de telles relations soient potentiellement intéressantes, il est reconnu qu'elle ne sont pas indispensables pour exprimer le lien entre une implantation et une spécification [125]. En particulier, le caractère symétrique de la relation n'est pas indispensable ; un système I peut implémenter un système S sans que les deux soient équivalents pour autant. Par exemple, il est généralement admis qu'une implantation réelle soit plus déterministe que sa spécification [125] qui est en général plus abstraite. Ainsi, les relations de conformité sont souvent (mais pas toujours) des préordres, c'est à dire des relations réflexives et transitives (pas nécessairement symétriques).

Les relations de conformité que nous présentons ici ont été souvent utilisées [28, 149, 189] car elles s'accordaient assez bien au contexte de conformité et de test de l'ISO.

Relation $\leq_{tr}$ Parmi les notations standards des $LTSs$ (section 2.3.1), nous avons défini $Traces(S)$ comme étant l'ensemble des séquences de S de l'état initial. La première relation de conformité que nous présentons est l'inclusion des traces. Cette relation vise à vérifier que toute trace de l'implantation est une trace de la spécification. Formellement,

Définition 8 *Soient* $S, I \in \mathcal{IOLTS}(\Sigma)$ *:*

$I \leq_{\mathbf{tr}} S =_{\Delta} Traces(I) \subseteq Traces(S)$. □

Il est facile de voir que $\leq_{tr}$ est réflexive et transitive, donc c'est un préordre.

Exemple 3.1 *Considérons les IOLTSs de la* Fig. *11.* $\neg(I_1 \leq_{tr} S)$ *vu que la trace* $?b.!z$ *de* I_1 *n'est pas une trace de* S*. Cependant* $I_2 \leq_{tr} S$*. Finalement,* $\neg(I_3 \leq_{tr} S)$ *vu que* $?a.!z$ *est une trace* I_3 *mais n'est pas une trace de* S*.*

Relation *conf* Cette relation a été proposée par E. Brinksma [26]. ***conf*** est basée sur l'observation de l'évolution et les blocages d'un système. Formellement :

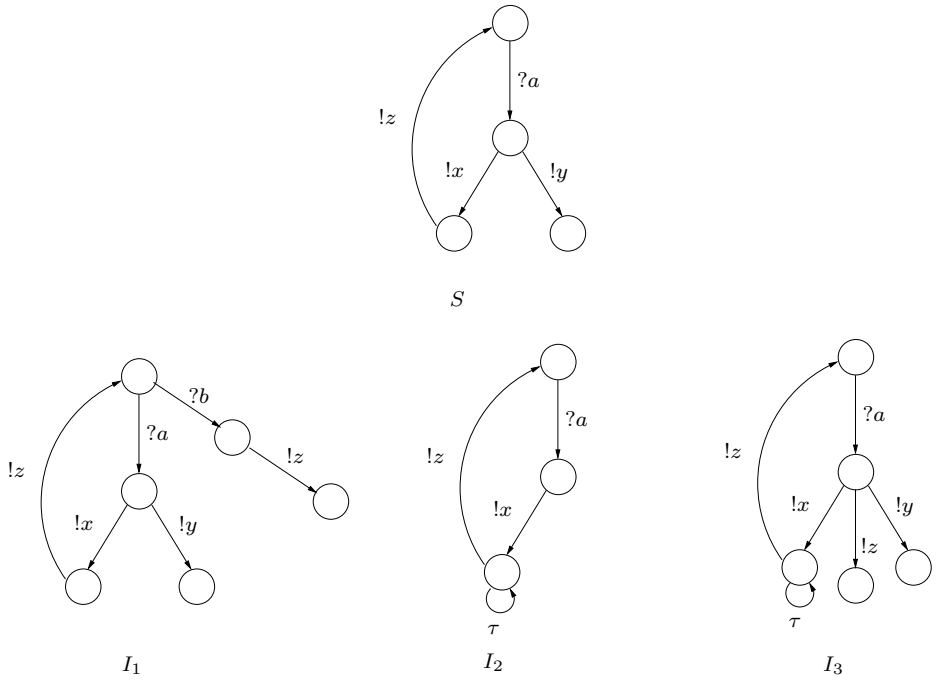


FIG. 11: Relation $\leq_{tr}$.

Définition 9 *Soient B_1, B_2 deux systèmes :*

$$\begin{array}{rcl} B_1 \textbf{\textit{ conf }} B_2 \quad si & \forall \sigma \in Traces(B_2), \forall A \in \Sigma & \\ si & \exists B_1' \ \forall a \in A \ \ B_1 \stackrel{\sigma}{\Rightarrow} B_1' \stackrel{a}{\not\Rightarrow} & \\ alors & \exists B_2' \ \forall a \in A \ \ B_2 \stackrel{\sigma}{\Rightarrow} B_2' \stackrel{a}{\not\Rightarrow} . & \end{array}$$

B_1 ***conf*** B_2 implique que si le système B_1 évolue après l'exécution de la trace σ alors, le système B_2 devrait aussi évoluer. De plus, si B_1 se bloque après une action a alors, le système B_2 devrait se bloquer aussi après la même action.

Relation *ioconf* La relation $conf$ est basée sur les traces et ne fait pas de distinction entre les entrées qui sont contrôlables par l'environnement du système et les sorties qui sont contrôlables par le système lui même. *Tretmans* [190] a introduit une relation ***ioconf*** qui fait une telle distinction. Cette relation établit qu'une implantation I est conforme à une spécification S si, après une trace de S, les sorties produites par I sont prévues par S. Rappelons que $Out(S, \sigma)$ est l'ensemble des événements produits par S après l'application de la trace σ (section 2.3.2). Formellement,

Définition 10 *Soient $S, I \in \mathcal{IOLTS}(\Sigma)$:*

I ***ioconf*** $S =_\Delta \forall \sigma \in Traces(S) \Rightarrow Out(I, \sigma) \subseteq Out(S, \sigma)$. □

La relation ***ioconf*** permet la spécification partielle et par conséquent, l'implantation peut ajouter un traitement additionnel lors d'une entrée non prévue par la spécification.

Exemple 3.2 *Considérons encore les IOLTSs de la* Fig.*11.* I_1 **ioconf** S *malgré le fait que* I_1 *a ajouté la branche dont la trace est* $?b.!z$*. En effet,* $?b.!z$ *n'est pas dans* $Trace(S)$*. De même,* $I_2\, ioconf\, S$*, malgré le fait que* I_2 *n'a pas implémenté l'émission de y après la réception de a (*$Out(I_2, ?a) = \{!x\} \subseteq Out(S, ?a) = \{!x, !y\}$*). Finalement,* $\neg(I_3\, \mathbf{ioconf}\, S)$*, car* $Out(I_2, ?a) = \{!x, !y, !z\} \not\subseteq Out(S, ?a) = \{!x, !y\}$*).* □

Relation *ioco* Le test permet d'observer le blocage d'un système par l'utilisation de temporisateurs. Le blocage se produit lorsque le système s'arrête d'évoluer.

Le blocage ou l'absence d'action dans un système peut être dû à plusieurs causes. On distingue :

- *Blocage de sortie* : se produit lorsque le système est en attente dans un état d'une entrée venant de son environnement. Concrètement, ce type de blocage se produit dans un état q lorsque $Out(q) = \emptyset$.
- *Blocage vivant (livelock)* : se produit lorsque le système diverge par une suite infinie d'actions internes.
- *Deadlock* : se produit lorsque le système ne peut plus évoluer. Ce cas correspond à un état q tel que $\forall a \in \Sigma\ ,\ q \overset{a}{\not\rightarrow}$.

La théorie ***ioco*** de $Tretmans$ [190, 191] considère les sorties et les blocages possibles d'une spécification. Pour ce faire, nous avons besoin de modéliser ceux-ci dans la spécification. En effet, l'observation d'un blocage de l'implantation ne doit pas forcément produire un verdict **Fail**, qui correspond au rejet de l'implantation, car il se peut que ce blocage soit prévu dans la spécification.

À partir de l'$IOLTS$ S de la spécification, nous avons besoin de considérer un $IOLTS$ S^δ, appelé automate *suspendu*, obtenu par l'ajout des informations de blocage. Dans S^δ, un blocage est modélisé par un événement de sortie $!\delta$ visible par l'environnement et ne faisant pas partie des événements de la spécification. S^δ est construit en ajoutant des boucles $q \xrightarrow{!\delta} q$ pour chaque état de blocage q.

Maintenant, la relation de conformité entre I et S est exprimée par la relation **ioco** définie par :

Définition 11 *Soient* $S, I \in \mathcal{IOLTS}(\Sigma)$ *:*

I ***ioco*** $S =_\Delta \forall \sigma \in Traces(S^\delta) \Rightarrow Out(I^\delta, \sigma) \subseteq Out(S^\delta, \sigma)$. □

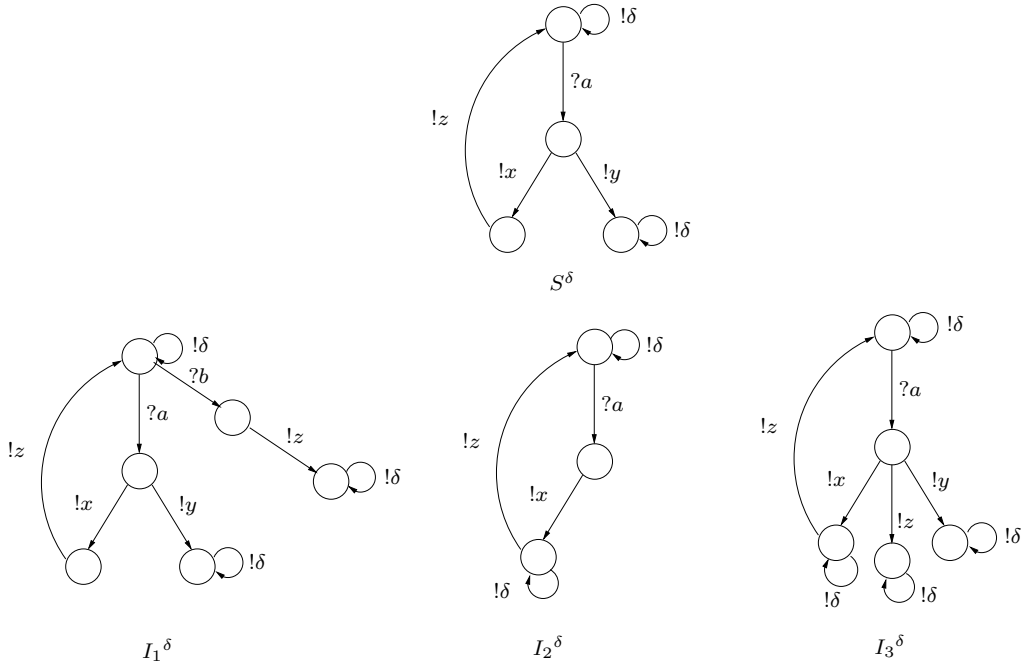


FIG. 12: Relation ***ioco***.

ioco étend ***ioconf*** en considérant non seulement les traces de S mais aussi les traces avec blocages de S, c'est à dire les traces de l'automate S^δ. La relation ***ioco*** est expliquée dans la FIG.12. La première implantation est conforme à la spécification tandis que les deux suivantes ne le sont pas :

- I_1 ***ioco*** S : on peut vérifier qu'en tout état, les sorties de I_1 sont incluses dans celles de S. I_1 permet une entrée supplémentaire par rapport à S mais ceci est autorisé : seules les sorties comptent. On peut donc faire des spécifications partielles.
- $\neg(I_2$ ***ioco*** $S)$: le blocage de I_2 après la séquence $?a.!x$ n'est pas autorisé dans la spécification.
- $\neg(I_3$ ***ioco*** $S)$: la sortie $!z$ après l'entrée $?a$ n'est pas autorisée dans la spécification.

Génération des tests

Les méthodes de génération des tests de conformité qui sont basées sur les systèmes de transitions peuvent être classées en deux sous-catégories : la génération aléatoire des tests et la génération à travers des objectifs de test.

Génération aléatoires. Nous présentons ici les travaux qui ont été conçus en collaboration entre l'*université de Twente*, l'*université de Technologie de Eindhoven* et les industriels *Philips Research Laboratories* et *Lucent Technologies*. L'outil *TorX* a été produit dans le cadre de cette collaboration.

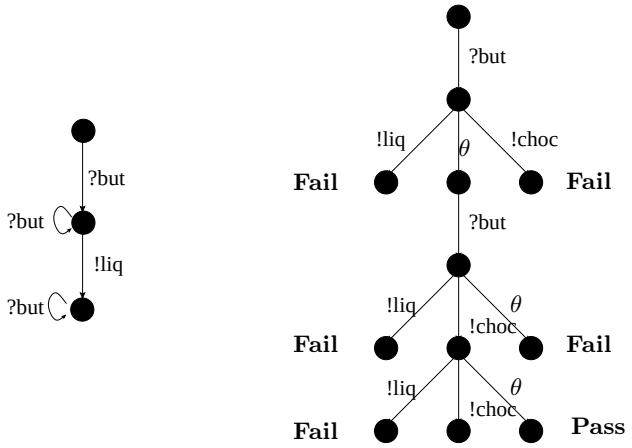


Fig. 13: Exemple d'un LTS et un cas de test dérivé par l'outil TorX

Pour les relations $ioconf$ et $ioco$, Tretmans [190, 191] a décrit des algorithmes de génération de test aléatoires. L'algorithme parcourt le système de transition S (ou S^δ pour la relation $ioco$). A chaque pas, soit :

1. On s'arrête et le verdict associé à l'état courant est $Pass$.

2. On choisit une entrée i de S (ou S^δ pour la relation $ioco$) et on applique l'algorithme récursivement sur $S\ after\ i$ (ou $S^\delta\ after\ i$ pour la relation $ioco$).

3. On considère toutes les sorties (et les blocages pour la relation $ioco$) de l'alphabet. Dans ce cas, une sortie ou un blocage x de S (ou S^δ) et on applique l'algorithme récursivement sur $S\ after\ x$ (ou $S^\delta\ after\ x$ pour la relation $ioco$), pour chaque sortie ou blocage non autorisé dans S ou (ou S^δ) on associe un verdict $Fail$.

L'algorithme peut produire un nombre infini de cas de test.

Génération à travers des objectif de test. Nous présentons ici les travaux qui ont été effectués en collaboration entre les laboratoires $IRISA$ et $Verimag$ et qui ont mené à développer l'outil TGV (*Test Generation with Verification technologies*) [108, 109, 63]. Un des ingrédients principaux de cette technique est l'utilisation d'objectifs de test pour générer les cas de test à partir d'une spécification formelle d'un système. Chaque objectif de test est considéré comme une description abstraite de comportements à tester (une partie de la spécification). Formellement, un objectif de test est modélisé par un $IOLTS$ équipé de deux états spéciaux *Accept* et $Reject$ pour décrire la satisfaction de cet objectif.

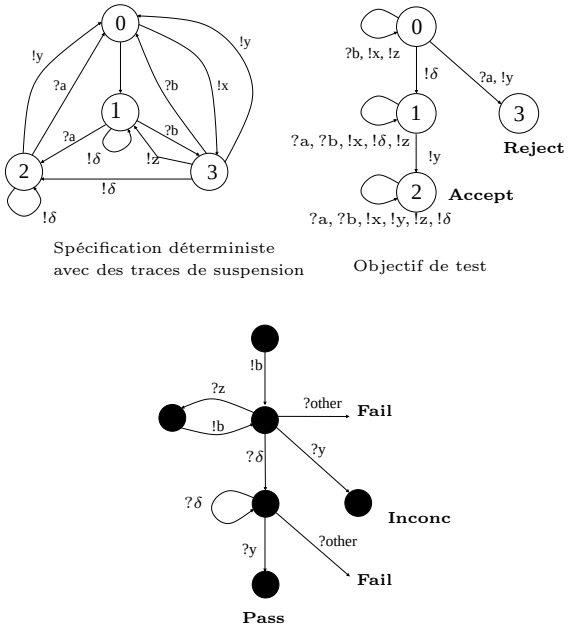


FIG. 14: Exemple de cas de test dérivé par l'outil TGV

L'objectif de test sert à sélectionner des séquences de la spécification. La dérivation des cas de test se décrit fonctionnellement en plusieurs étapes :

1. Un produit synchrone entre la spécification et l'objectif de test ;
2. La suppression des actions internes et la déterminisation du modèle ;
3. La synthèse d'un sous-graphe et la résolution des conflits de contrôlabilité.

Les algorithmes utilisés fonctionnent *à la volée* (*on-the-fly*) ce qui signifie qu'une exécution de l'algorithme principal va activer l'exécution de chacune des étapes intermédiaires. Le terme *à la volée* signifie également que pour produire le résultat, il ne sera pas nécessaire de construire les résultats complets de chaque étape, ni même les entrées complètes.

Un cas de test est une expérience réalisée par le testeur sur l'implémentation. Il est modélisé par un $IOLTS$ dont le graphe associé est un arbre. Les branches d'un cas de test décrivent les séquences d'interactions entre le testeur et l'IUT. Le rôle d'un cas de test est de détecter si l'IUT est conforme à sa spécification (la conformité étant donnée par une relation de conformité : *ioconf*, *ioco*,...).

D'autres techniques utilisent des objectif de test pour la sélection des tests. Notons par exemple, l'outil *Samstag* (voir la section 3.3) où les objectifs sont fournis sous forme de MSC (*Message Sequence Charts*) et la spécification est donnée en SDL, l'outil STG (*Symbolic Test Generator*) [40, 41] qui prend en entrée une spécification et un objectif de test symboliques.

3.3 Outils pour la génération des tests de conformité

Une grande partie des études porte sur la génération de tests à partir des modèles mathématiques comme les $FSMs$ (et leurs variantes) et les systèmes de transitions étiquetées. Les méthodes de génération de tests à partir des spécifications formatées en langages de spécification, par exemple SDL, s'appuient sur la transformation de ces dernières en $FSMs$, LTS ou Arbres d'accessibilité [86]. En conséquence, les modèles obtenus par cette transformation sont en général de très grande taille, ceci génère un nombre important et donc pratiquement inexploitable de cas de test. De plus, pour certains systèmes, cette transformations est pratiquement infaisable.

TGV

L'implémentation du TGV [63, 109, 107] automatise l'architecture fonctionnelle présentée dans la section 3.2.3. L'$IOLTS$ de la spécification et de l'objectif du test sont fournis soit de manière explicite par un graphe dans un format *bcg* ou *Aldbaran* soit de manière implicite par un compilateur d'un langage de spécification (le générateur de simulateur d'ObjectGéode pour SDL, Open/Caesar pour $LOTOS$ et $Umlaut$ pour UML). TGV utilise une API (application programming interface) fournit par la boîte à outil $CADP$ [62, 62]. TGV utilise les techniques de vérification telles que le calcul de produit synchrone et la vérification à la volée pour la génération des cas de test. Les cas de test sont formatés en $TTCN$, *bcg* et *Aldebaran*. La plateforme de cet outil est donnée dans Fig.15.

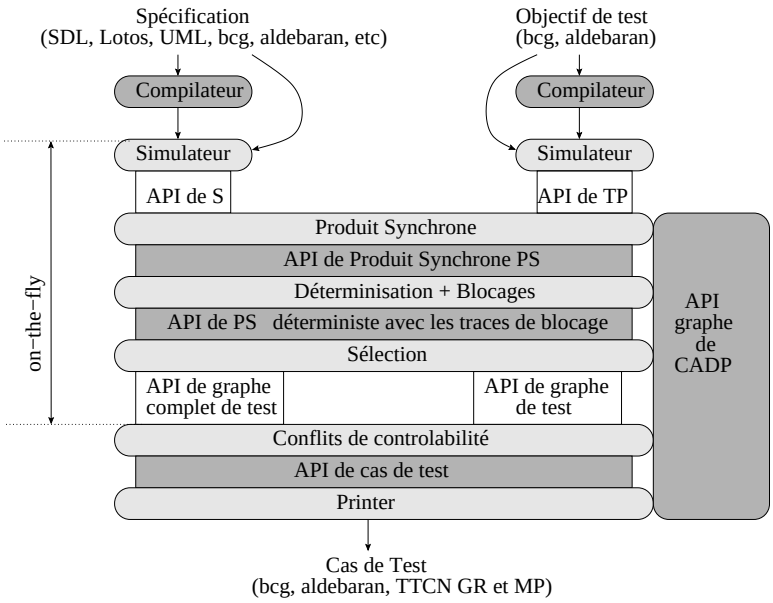


Fig. 15: Plateforme de l'outil TGV

TorX

Comme nous l'avons dit auparavant, l'outil $TorX$ [192, 15] est le résultat d'une collaboration entre l'université de *Twente* (*J. Tretmans* et *Ed. Brinksma*), l'*université de Technologie de Eindhoven* et les industriels *Philips Research Laboratories* et *Lucent Technologies*, conçu dans le cadre du projet "*Cote de Résyste*" (*COnformance TEsting of REactive SYSTEms*). L'environnement de $TorX$ permet actuellement la dérivation et l'exécution automatique de cas de test pour des spécifications $LOTOS$, $PMLP$ (Protocol Meta Langage Promela), ou SDL. Le principe de cet outil a été discuté dans la section 3.2.3. L'outil $TorX$ se compose de plusieurs modules : Explorateur, amorce, conducteur, adaptateur et stockage de $TTCN$. L'interface utilisée entre les modules de $TorX$ est fournie par la boîte à outil $OPEN/CAESAR$, qui offre toutes les fonctionnalités requises. La plateforme de cet outil est donnée dans FIG.16.

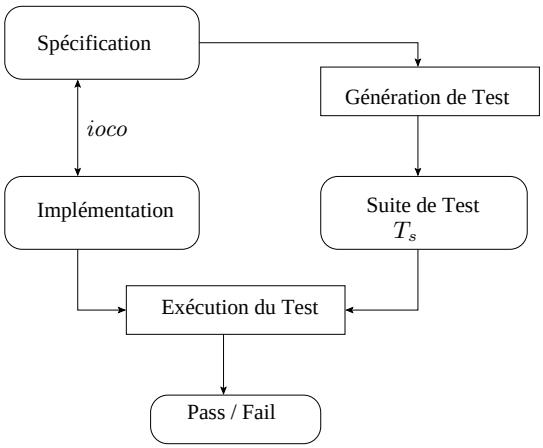


FIG. 16: Plateforme de l'outil $TorX$

SaMsTaG

Cette méthode "*SDL And Msc based Test cAse Generation*" [72, 73] permet de générer des cas de test abstraits à partir d'une spécification formelle et d'un ensemble d'objectifs de test. La spécification formelle est écrite en SDL et les objectifs de test sont exprimés par des $MSCs$ (Message Sequence Charts). En d'autres termes, $SaMsTaG$ consiste à réaliser une composition parallèle de la spécification SDL et du MSC par simulation ; donc, elle fournit : un simulateur SDL, un simulateur MSC et un générateur de cas de test.

Autolink

Cet outil [179] permet de générer des suites de test $TTCN$ à partir des spécifications SDL et de besoins formulés sous forme de MSC. La génération de cas de test passe par plusieurs étapes :

– Description des chemins SDL qui vont servir de base pour la génération des cas de test. Chaque chemin est enregistré sous forme de MSC. Ce dernier peut montrer uniquement le comportement observable du système avec son environnement.
– Définition d'une configuration car une suite de test dépend de plusieurs options (par exemple, l'usager peut choisir parmi plusieurs formats pour les données de sortie).
– Calcul d'une représentation interne pour chaque cas de test à l'aide des MSC et le fichier de configuration. Cette représentation contient toutes les séquences d'envoi et de réception de messages qui entraînent un verdict *Pass* ou *Inconclusive*. De plus, elle garde une trace de la structure du cas de test.
– Génération de suites de tests $TTCN$ à partir de la représentation interne des cas de test et de la liste de contraintes.

Tveda V3, TVeda V3+

Cet outil [43] a pour but la génération automatique de cas de test à partir de spécifications *Estelle* ou SDL. Il propose de calculer un ensemble de buts de test sur le graphe syntaxique de la spécification SDL (un but de test correspondant à une transition du graphe SDL). Les tests générés par cet outil sont décrits soit dans le langage $TTCN$ ou *Menuet*. Pour déterminer les chemins à tester, l'outil utilise deux méthodes : l'évaluation symbolique ou l'analyse d'accessibilité. La première méthode pose beaucoup de restrictions sur les spécifications à tester. L'analyse d'accessibilité quant à elle ne pose aucune restriction sur la spécification, de plus, $Tveda\ V3$ a comme interface un outil commercial très puissant d'analyse d'accessibilité appelé $Veda$. La plateforme de cet outil est donnée dans Fig.17.

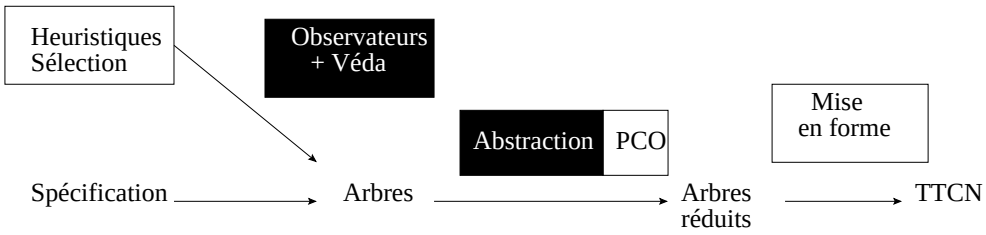


Fig. 17: Plateforme de l'outil $TVeda$

$TVedaV3+$ [188] est une extension de l'outil précédent. Il définit un but de test non pas comme une transition SDL mais comme un chemin du graphe syntaxique SDL partant de l'état initial de la spécification et se terminant soit par un $STOP$ soit par un retour à l'état initial. Ainsi, les fonctionnalités du système décrites par le but de test correspondent à des comportements de bout en bout. Les cas de test sont automatiquement générés à partir des buts de test et d'une construction partielle du graphe d'accessibilité de la spécification SDL.

TestGen

Cet outil [3, 4] utilise la méthode classique d'UIO de façon partielle grâce à la construction d'un graphe d'accessibilité et à l'application d'algorithmes de réduction à ce graphe. $TestGen$ est basé sur une utilisation des méthodes classiques de test, basées sur les machines à états finis, qui requièrent de disposer d'un automate fini calculable modélisant un protocole réel. La plupart du temps le graphe d'accessibilité de tels protocoles ne peut être calculé car il est trop gros. En fixant les valeurs des paramètres des messages en entrée, le graphe obtenu est de taille raisonnable. Aussi, $TestGen$ résout le problème de la longueur des séquences de test en utilisant un outil de réduction de graphe, *Aldbaran*, de la boîte à outil $CADP$, pour minimiser l'automate en fonction de l'architecture de test, avant d'engendrer les tests. La plateforme de cet outil est donnée dans FIG.18.

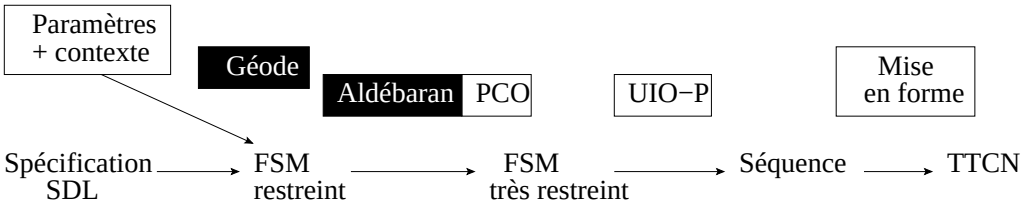


FIG. 18: Schéma général de l'outil $TestGen$

TOPIC V2

$TOPIC\ V2$ (prototype de $TTCGEN$: générateur automatique de cas de test pour *ObjectGéode* de $Verilog$) [2], fait la co-simulation de la spécification SDL et d'un observateur représentant le but du test. Cette co-simulation permet d'explorer un graphe contraint, c'est à dire une partie du graphe d'accessibilité de la spécification, ce qui permet d'utiliser cette méthode pour les graphes infinis. L'observateur est décrit dans le langage $GOAL$ (*Geode Observation Automata Language*). Afin de faciliter l'utilisation de $TOPIC$, il est possible de générer les observateurs à partir des $MSCs$. A partir du graphe contraint, des procédures sont exécutées pour générer les tests en $TTCN$.

Test Composer

Cet outil [198] est fondé sur l'utilisation des techniques d'exploration d'espace d'états et le langage $TTCN$ pour décrire les séquences de test. Il est connu par sa flexibilité pour exprimer les objectifs de test et son adaptabilité avec les différents langages de spécification connus dans le test dont le langage de base est SDL. Par ailleurs, Test Composer offre la possibilité de la génération automatique des postambules.

TAU

C'est une famille d'outils qui offrent un support pour la conception, le développement et le test d'applications très diverses. Sa dernière version (*TAU Generation 2*) [186] supporte les standards industriels de modélisation et de test tels que UML-2.0 et $TTCN$-3. Elle est composée de : *TAU/Architect* pour la conception des systèmes. *TAU/Developer* pour le développement des logiciels. *TAU/Tester* pour le test. *TAU/Logiscope* pour la garantie de la qualité et la métrique des logiciels. Les outils de première generation de TAU (*Telelogic TAU Generation 1*) sont largement utilisés dans le développement des systèmes de télécommunications et des logiciels embarqués, supportant les standards de spécification tels que SDL et de test de conformité tels que $TTCN$-2.

Autres

Il existe de nombreuses autres outils pour la génération des test de conformité. Nous citons par exemple :

$TESDL$ [30] est un prototype pour la génération automatique de cas de test à partir de spécifications SDL. Ces dernières ont certaines restrictions (un processus par bloc, deux processus ne peuvent recevoir le même genre de signal, etc). Les éléments SDL suivants sont supportés : canaux, blocs, un processus par bloc, les routes de signaux, état, entrée, sortie, tâche, décision, etc. L'outil accepte des spécifications SDL et produit des cas de test en $TTCN$.

STG [40, 41] (Symbolic Test Generator) est un générateur de cas de test symboliques pour des systèmes réactifs. STG prend en entrée une spécification et un objectif de test symbolique décrites par des $IOSTS$ (In/Output Symbolic Transition System) extension du langage IF.

$TTCNLink$ (ou $LINK$) [103] est un environnement pour le développement de suites de test en $TTCN$ à partir de spécifications SDL de l'ensemble d'outils SDT 3.0 [182].

3.4 Conclusion

Ce chapitre nous a permis de voir un état de l'art sur les différentes approches et outils développés pour la génération des tests de conformité.

La majorité des méthodes basées sur les FSM permettent l'identification de l'état d'arrivée. En revanche, leur applicabilité est limitée d'une part, à cause des hypothèses portées sur les modèles (complétude, forte connexité, déterminisme,...) et, d'autre part à cause de la complexité de leurs algorithmes. De plus, la sémantique du modèle FSM est différente de celle utilisée dans les langages de spécification. Ceux-ci sont en général basées sur la sémantique des systèmes de transitions.

Les méthodes basées sur les systèmes de transitions sont plus proches des langages de

spécification. De plus, elles ne font à priori aucune hypothèse sur les modèles ce qui élargit leur applicabilité. D'un point de vue algorithmique, ces techniques présentent une meilleure complexité (linéaire excepté la déterminisation qui est exponentielle)[109]. En revanche, ces méthodes sont non-exhaustives et notamment à cause l'utilisation d'objectifs de test. De plus, elle perdent la notion d'identification d'états.

Un inconvénient commun entre les deux familles de méthodes précédentes est l'absence total de la considération des données dans le test. Pour ce faire, les méthodes basées sur les $EFSM$ et les graphes de flux de données ont offert une autre vision pour coupler les contrôles et les données dans les séquences de test. En revanche, l'utilisation de la sémantique des FSM limite leur applicabilité comme dans la première famille de méthodes.

Dans la fin du chapitre, nous avons exposé quelques outils du test de conformité. Chaque outil contourne le problème de l'explosion combinatoire en imposant certaines restrictions sur la spécification ou bien en considérant un sous-ensemble de la spécification (objectifs de test). Certains d'entre eux fixent les valeurs des paramètres en entrée, d'autres permettent de faire une simulation de la spécification par rapport à certains buts de test que l'usager doit entrer (sous forme de $MSCs$ ou à l'aide d'un langage, par exemple $GOAL$), ce qui permet de tester un sous-ensemble de la spécification seulement, et finalement d'autres posent des restrictions sur les constructions du langage de spécification.

Chapitre 4

État de l'art sur le test de robustesse

Depuis quelques décennies, de nombreux travaux ont été effectués dans le domaine du test de conformité. Cependant, il est apparu récemment un besoin au niveau de la robustesse des systèmes : comment le système réagit-il face à des événements imprévus ? Cette réponse ne peut pas être apportée par le test de conformité qui ne tient pas compte, en général, du caractère hostile de l'environnement.

Dans ce chapitre nous passons en revue quelques travaux développés afin de répondre à certaines sollicitations de robustesse. Ces travaux peuvent être essentiellement classés en deux catégories.

4.1 Méthodes fondées sur des modèles liés au domaine d'entrée

Plusieurs travaux utilisent l'injection de fautes pour la caractérisation de la robustesse de composants logiciels, tels que les micro-noyaux [39, 7], les systèmes d'exploitation à usage général [193, 119] et les intergiciels [134, 146, 133]. Dans cette section, nous passons en revue quelques exemples de techniques d'injection de fautes dédiées à la caractérisation de la robustesse : *Ballista*, $MAFALDA$, $FINE$, $FUZZ$, *Xception*, $PROTOS$, $CRASHME$, $REIDDLE$ et $JCrasher$.

4.1.1 Ballista

Ballista, développé à l'université de Carnegie Mellon, est un outil d'évaluation de la robustesse des composants logiciels sur étagère [118, 119]. Il combine des approches de test de logiciel et des approches d'injection de fautes. *Ballista* est basé sur la combinaison de cas de tests prenant en compte les paramètres valides et invalides des appels système et des fonctions d'un composant logiciel exécutif (voir FIG.19).

Pour caractériser la robustesse du système cible, *Ballista* définit l'échelle $CRASH$ composée de six modes de défaillance :

1. *Catastrophique* (*Crash*) : une défaillance majeure du système qui nécessite un redémarrage ;
2. *Redémarrage* (*Restart*) : une tâche est bloquée et nécessite une relance ;
3. *Arrêt d'exécution* (*Abort*) : interruption ou arrêt anormal de la tâche ;
4. *Silencieux* (*Silent*) : aucun code d'erreur n'est retourné ;
5. *Hindering* : le code d'erreur retourné est incorrect.
6. Aucune réaction du système.

Les trois premiers modes de défaillance sont observés automatiquement tandis que les trois derniers nécessitent un traitement manuel.

Pour chaque cas de test, un appel système est effectué une seule fois avec un ensemble de paramètres bien particulier. A chaque paramètre est associé un nombre de valeurs prédéfinies. Ces valeurs sont issues d'une base de données valides et exceptionnelles associées à chaque type de données de chaque argument de l'appel système. La détermination des valeurs valides et invalides ne dépend pas de l'aspect fonctionnel de l'appel système mais plutôt du type du paramètre. L'avantage majeur de cette approche est la génération automatique du code source des cas de test avant exécution.

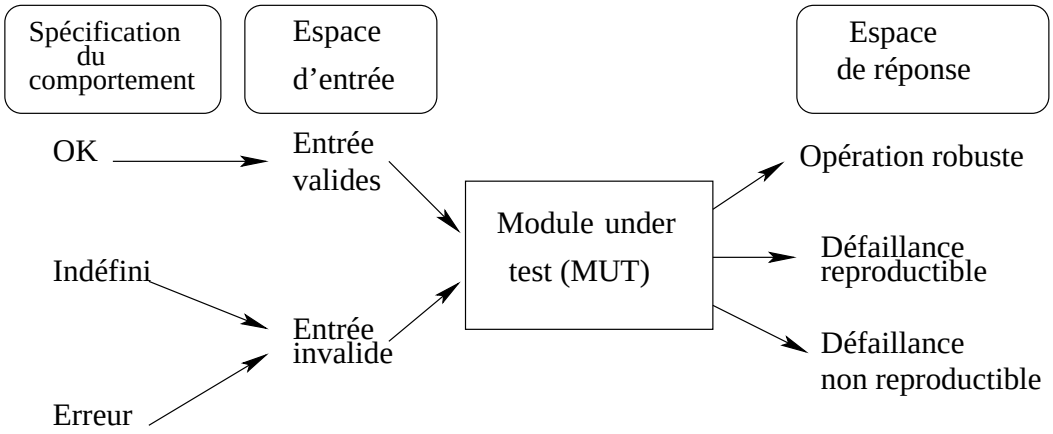


FIG. 19: Schéma de l'approche *Ballista*

Dans un premier temps, cette approche de test de robustesse a été appliquée aux systèmes d'exploitation supportant l'interface $POSIX$ (*Portability Operating Systems Interface*). Les résultats obtenus en utilisant *Ballista* pour 15 systèmes d'exploitation $POSIX$ ont montré que le mode de défaillance *catastrophique* est présent dans 6 systèmes d'exploitation. Le *redémarrage* d'application n'a été observé que dans deux cas. Le mode de défaillance *arrêt d'exécution* quant à lui a été révélé par tous les systèmes d'exploitation. Par ailleurs, le mode de défaillance *silencieux* ne garantit pas forcément un comportement sûr

de fonctionnement. Il convient de noter qu'une défaillance de type *arrêt d'exécution* est un comportement attendu de la part du système après l'introduction de fautes. En revanche, les défaillances *catastrophiques* et *redémarrages* reflètent des comportement anormaux et graves.

Par la suite, l'approche *Ballista* a été portée aux systèmes de la famille Windows (2000, NT, 95, 98, 98SE et CE) ciblant ainsi l'interface standard WIN32 de Microsoft pour permettre la comparaison des familles Windows et Linux [177].

D'autre application de *Ballista* concernant l'étude de la robustesse de plusieurs *ORB CORBA* [146]. Ces travaux ont montré la portabilité de l'outil et sa facilité d'utilisation à différents niveaux du système.

Quelques outils ont été utilisés pour l'injection de fautes :
- *FTAPE* [194] : injecte les défauts comme erreurs de parité dans des segments de mémoire par l'intermédiaire du logiciel.
- *FIAT* [174] : injecte les défauts en faisant des changements sur l'image binaire d'un *MUT* (Module Under Test).
- FAUST [185] : exécute des mutations sur le code source d'un *MUT*.

L'approche *Ballista* s'appuie sur deux types de test : *"boîte blanche"*, basé sur l'accès au code source pour tester les valeurs de données exceptionnelles ou, *"boîte noire"*. Cependant, on ignore les actions internes et, on compare des implémentations différentes d'un *MUT*. En particulier, deux types de "boîte noire" sont utilisés :
- Test de domaine "*domain testing*" : localiser et sonder des points autour des extrémités et des discontinuités dans le domaine d'entrée ;
- Test de syntaxe "*syntax testing*" : construction des chaînes de caractères d'essai qui sont conçues pour examiner la robustesse du système.

4.1.2 MAFALDA

MAFALDA, (*Microkernel Assessment by Faut injection Analysis and Design Aid*), développé au *CNRS−LAAS*, est un outil d'injection de fautes par logiciel dont les objectifs sont d'aider à caractériser les modes de défaillance d'un exécutif à base de micro-noyau et de faciliter la mise en œuvre de mécanismes d'empaquetage du noyau pour améliorer sa sûreté de fonctionnement [59, 170, 171]. Le premier micro-noyau considéré a été *Chorus*. Dans l'outil *MAFALDA*, deux modèles de fautes sont considérés :
- Des fautes d'interaction par sollicitation intensive et erronée des services du micro-noyau via son interface ;
- Des fautes simulant l'effet des fautes physiques, elles visent les segments de code et de données du micro-noyau.

L'injection de fautes d'interaction consiste à intercepter les appels au micro-noyau et à inverser un bit de façon aléatoire dans la chaîne des paramètres avant de poursuivre l'exécution de l'appel dans le micro-noyau. Une activité synthétique est associée à chaque composant fonctionnel du micro-noyau cible (par exemple, synchronisation, ordonnancement,

communication, etc). Deux modules logiciels sont chargés dans chaque micro-noyau cible : le module d'interception et le module d'injection. Les paramètres caractérisant une campagne d'injection de fautes sont stockés sur la machine hôte dans deux fichiers d'entrée : le descripteur de campagne et le descripteur d'activité.

Le banc *MAFALDA* est composé de deux entités :
- Un ensemble de 10 machines cibles, exécutant des micro-noyaux cibles en parallèle ;
- Une machine hôte dont le rôle est de définir, d'exécuter, de contrôler les expériences sur les machines cibles et d'analyser les résultats.

MAFALDA permet également d'injecter des fautes dans les segments de mémoire du micro-noyau. Le but est d'évaluer l'efficacité des mécanismes internes de détection d'erreur et de tolérance aux fautes. *MAFALDA* procède à une distribution aléatoire et uniforme des fautes injectées sur l'espace d'adressage du module cible, mais ne réalise que des expériences où les fautes sont effectivement activées. L'étude expérimentale a montré la différence des distributions de modes de défaillance, d'une part entre les différents composants fonctionnels, et d'autre part suite à l'injection de fautes dans les segments de texte et de données.

4.1.3 FINE, DEFINE

FINE (*Fault INjection and monitoring Environnement*), développé à l'université d'Illinois [113], est un outil d'injection de fautes dont la cible est le noyau Unix. Son objectif est d'étudier les canaux de propagation d'erreurs dans le noyau Unix et d'évaluer l'impact de divers types de fautes sur son comportement.

DEFINE (*DistributEd Fault INjection and monitoring Environnement*) [114] est un outil d'injection de fautes dans un environnement réparti basé sur *FINE*. Il reprend les fautes et les techniques d'injection de *FINE* localement sur chaque site du système distribué. Les fautes injectées, logicielles ou matérielles, sont activées par des activités synthétiques. Les logiciels de type applicatif n'ayant pas accès aux ressources internes du système d'exploitation, la partie de *FINE* responsable de l'injection de fautes a été implantée directement sur le noyau.

Les fautes logicielles considérées dans ces outils sont les fautes d'initialisation, d'affectation, de branchement ou affectant une fonction. Quant aux fautes matérielles, elles sont classées par localisation de manifestation : la mémoire, le bus, le processeur et les bus d'entrées/sorties.

Les résultats obtenus montrent que les fautes affectant la mémoire et les fautes logicielles ont généralement une latence de détection élevée tandis que les fautes affectant le bus et le processeur causent un arrêt immédiat du système. À peu près 90 % des fautes détectées sont causées par le matériel. En ce qui concerne la propagation des erreurs, les expériences ont révélé que seules 8 % des fautes injectées dans un composant fonctionnel ont été propagées vers d'autres composants. En ce qui concerne la propagation des erreurs entre les machines, les expériences montrent que la propagation des fautes du serveur vers les clients est plus

importante que dans le sens inverse.

4.1.4 FUZZ

L'outil FUZZ [137, 65] consiste à vérifier la robustesse de certains systèmes d'exploitation (UNIX, Windows NT) et leurs utilitaires. L'environnement d'exécution est connu (description de machines, architectures matérielles, etc). En revanche, les spécifications et le code du système ne sont pas disponibles. Le principe du test proposé est basé sur l'expérience et la connaissance des fautes. Les éléments testés sont des chaînes de caractères représentant certaines commandes systèmes. Le programme FUZZ est fondamentalement un générateur des caractères aléatoires. Il produit les chaînes de caractères (imprimables ou de contrôle) continues dans son fichier de sortie standard. Initialement, le modèle de faute est vide est dès qu'une faute est provoquée, la commande relative à cette panne est incluse dans le modèle de fautes. Le test effectué est comparable à celui de la conformité. Dans son principe, les fautes sont injectées en tant que commandes système, si le résultat de la commande est correct (exécution attendue), alors le système est robuste pour ce type de faute. Dans un premier temps, 88 utilitaires ont été cibles sur 7 versions de UNIX. Ensuite, 47 applications tournant sur Windows NT ont été visées. Des exemples d'utilitaires visés sont respectivement *diff*, *cat*, *emacs*, *lisp*, *latex*, *telnet* pour les systèmes UNIX et *Access*, *Netscape*, *Word*, *Wordpad* pour les systèmes Windows. L'outil a été également utilisé pour tester des applications avec une interface graphique sous UNIX. Les pourcentages de défaillances se trouvent respectivement dans l'intervalle 25% - 33% dans le cas des utilitaires UNIX et dans l'intervalle 72%-90% pour applications tournant sous Windows NT. Les résultats comprennent également une liste de catégories de causes de défaillances d'utilisateurs.

4.1.5 Xception

Xception [32] est un outil d'injection de fautes par logiciel, développé par l'université de *Coimbra* (Portugal). Il cible les logiciels de type applicatif et exécutif. *Xception* utilise les fonctions avancées du débogueur interne aux processeurs Intel et Power PC pour établir un certain nombre de points d'arrêts (breakpoints) permettant entre autres de corromper un ou plusieurs bits des registres mémoire, du bus ou du processeur. Initialement, *Xception* a été utilisé pour injecter des fautes représentatives des fautes matérielles. Les résultats obtenus montrent que l'impact des fautes est très dépendant du composant affecté. Par exemple, les expériences d'injection de fautes ciblant le bus d'adresses ont provoqué des défaillances majeures du système dans 65% des cas. Par contre, les fautes affectant le bus de données n'ont pas eu un impact si dramatique. Des défaillances majeures du système ont été enregistrées seulement dans 30% des cas. Plus tard, une étude des fautes de programmation a été effectuée [133] afin de permettre l'émulation de ce type de fautes, ce qui est du plus grand intérêt dans le processus de validation de mécanismes de tolérance aux fautes.

4.1.6 Autres outils

PROTOS [121]. Il vise à tester particulièrement des implémentations de protocoles de communication vues comme boîtes noire (test fonctionnel). *PROTOS* utilise une méthode de *mini-simulation* basée sur la *grammaire d'attribut* (BNF) pour modéliser à la fois, la syntaxe d'entrées, le comportement du système et les cas de test. Cette méthode a été pratiquée pour tester la robustesse des navigateurs implantés dans des terminaux mobiles.

CRASHME [37]. Il a été conçu pour évaluer la robustesse de systèmes d'exploitation Windows et UNIX. *CRASHME* est un programme simple qui remplit une zone de mémoire avec des données aléatoires et tente de faire exécuter cette zone par un système d'exploitation comme si c'était un programme. Ensuite, il analyse les exceptions, les erreurs et les signaux générés par le matériel ou par le logiciel.

Riddle [74, 135]. Cet outil emploie une grammaire d'entrée pour produire des combinaisons correctes, ou incorrectes, ou des entrées couvrant au mieux les fonctionnalités du système à tester. Il fournit des critères d'échec appartenant à des parties bien précises des systèmes. Ceci exige la connaissance préalable du système.

JCrasher [46]. C'est un testeur de robustesse automatique pour les codes Java, *JCrasher* examine le type de données d'un ensemble de classes de Java et construit les fragments de code qui créeront des exemples de différents types pour tester le comportement des méthodes publiques en présence des données aléatoires. *JCrasher* essaye de détecter des anomalies en faisant tomber en panne le programme sous test, c'est-à-dire, jette une exception d'exécution non déclarée.

4.2 Méthodes fondées sur des modèles de comportement

Cette catégorie de méthodes est basée sur la disponibilité d'un modèle formel décrivant le comportement du système ainsi que quelques hypothèses pour évaluer sa robustesse (spécification dégradée, propriété de robustesse, etc). Plusieurs approches sont issues de cette démarche.

4.2.1 STRESS

STRESS [184] (*Systematic Testing of Robustness by Evaluation of Synthesized Scenarios*) est une méthode basée sur les principes de FOTG (*Fault Oriented Test Generation*) [78]. FOTG utilise l'injection d'erreurs pour orienter la génération de cas de test. La robustesse est définie en présence de fautes. Les fautes sont soient spécifiques comme la caractéristique dynamique du support "Internet", soient données sous forme de modèle de

faute pour définir : Les pertes de paquets, les corruptions de données, les réorganisations de topologies, les pannes de machines, etc. Le formalisme de description est les machines d'états finis (GFSM) (*Global finite state machine*). Les cas de test sont perçus comme des scénarii à réaliser. Ils incluent les patterns : $< Ev, T, F >$ où Ev décrit un ensemble d'événements, T est une *Topologie* représentée par l'ensemble $< N, L >$, avec L l'ensemble des *Liens* de connexion et N les *Noeuds* de connexion et, finalement F représente les *Fautes*. La propagation d'une erreur de F peut être rendue visible grâce à la connaissance de la topologie T du réseau.

Le principe de la génération de test est le suivant : A partir d'une erreur contenue dans une spécification d'un protocole (perte de messages, accident de noeud, etc), à l'aide de la méthode FOTG, on parcourt (en avant) la spécification pour atteindre un état d'erreur (un état d'erreur est perçu par le protocole comme un état qui ne peut apporter une réponse à la condition souhaitée). Dès cet état d'erreur, FOTG produit, par un parcours en arrière, la séquence de cas de test.
Les différentes phases sont détaillées dans la Fig.20.

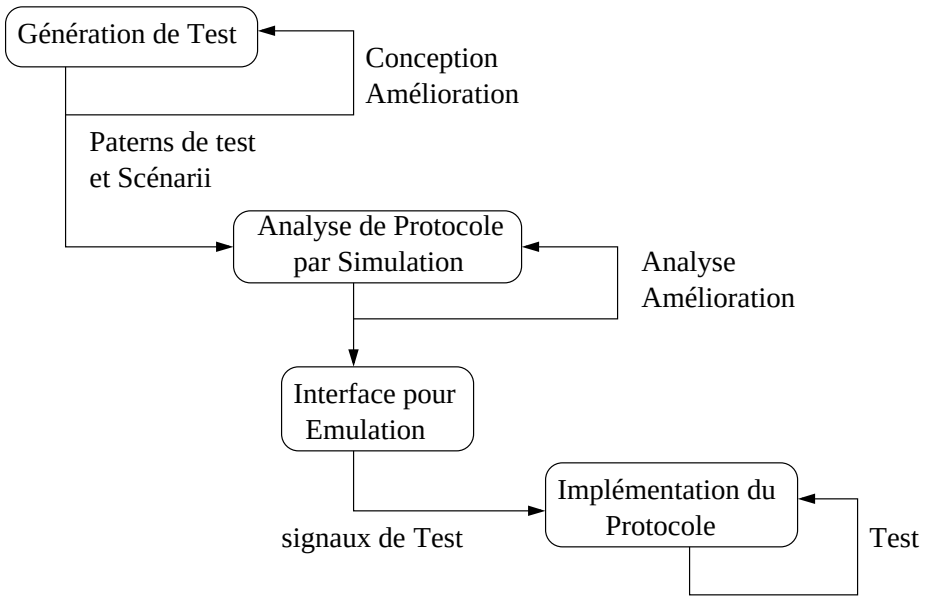


Fig. 20: Plateforme de la méthode STRESS

- **Génération de test** : Cette phase consiste à définir le modèle du protocole ainsi que les critères de la correction. Ensuite, elle génère les séquences de test.
- **Analyse** : Cette phase développe une simulation détaillée du protocole et analyse le comportement attendu après l'application des séquences de test.
- **Test** : Cette phase consiste à appliquer des signaux de test et évaluer la couverture du test.

4.2.2 Approche Verimag

Dans l'approche effectuée par le laboratoire Vérimag [145, 64], le critère de robustesse est vu comme une propriété externe P : une implémentation est dite robuste si et seulement si elle satisfait P en présence d'aléas. Cette approche est fondée sur les éléments suivants :

- Une *spécification* S du comportement attendu de l'implémentation dans son environnement nominal (hors aléas). S peut être décrite par exemple par un ensemble de systèmes de transitions. Cette spécification servira de référence pour la synthèse des cas de test, mais il n'est pas nécessaire que l'implémentation lui soit conforme.
- Un *modèle de fautes* $\mathcal{MF}$ décrivant l'ensemble des aléas considérés, aussi bien ceux dus à l'environnement (entrées incorrectes, stress, valeurs de paramètres hors domaine, etc.) que ceux dus à des défaillances possibles de composants internes à l'implémentation (pannes, erreurs de communication, etc.). Divers formalismes peuvent être envisagés pour décrire ce modèle de fautes en fonction de l'application considérée, mais il doit pouvoir se ramener à un ensemble de mutations de S (modification de valeurs de timer, de la topologie des communications, des valeurs échangées, etc).
- Une *propriété de robustesse* P définissant le comportement acceptable de l'implémentation en présence d'aléas. En pratique P peut caractériser soit un invariant du système, soit une propriété de progrès (par exemple, "l'implémentation revient toujours en mode nominal après être entrée en mode dégradé"). P est considérée comme une propriété linéaire, décrivant l'ensemble de séquences d'exécutions robustes (finies ou infinies) de l'implémentation.

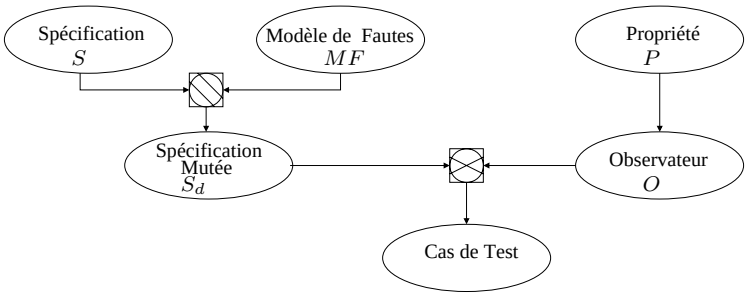


FIG. 21: Schéma de l'approche Verimag

Le processus de la génération de tests est résumé de la manière suivante (voir FIG.21) :

- Génération d'une spécification dégradée S_d par mutation de S avec le modèle de faute $\mathcal{MF}$,
- Génération d'un observateur $\mathcal{O}$, exprimé par un automate de Rabin [151], décrivant l'ensemble des séquences de P. Cet observateur identifie les séquences non robustes de S_d.
- Synthèse de cas de tests à partir de S_d et $\mathcal{O}$: les séquences d'exécution non robustes sont extraites à partir de S_d et transformées en cas de test (modulo les critères de contrôle et d'observation considérés). Un verdict de robustesse leur est alors associé.

4.2.3 Approche Rollet-Fouchal

Dans l'approche proposée dans [158, 157, 66], les auteurs ont proposé une méthode pour le test de robustesse, avec prise en compte des aspects temporels. Le concepteur du système fournissait deux spécifications, une *nominale* pour décrire le comportement en conditions nominales, et une *dégradée* décrivant les actions vitales, qui devaient être assurées par l'IUT dans les situations critiques et inattendues. Cette approche peut se résumer par le cheminement suivant :

1. Génération des séquences de test à partir de la spécification nominale S,
2. Application de radiations magnétiques sur l'IUT,
3. Application des séquences générées sur l'IUT,
4. Fin d'application de radiations magnétiques sur l'IUT,
5. Analyse des résultats et verdict partiel,
6. Ajout d'aléas aux séquences de test,
7. Application des séquences mutantes sur l'IUT et,
8. Analyse des résultats et verdict final.

Après la session de test, les traces d'exécution sont analysées à l'aide d'une relation d'acceptation. Les verdicts donnés par cette approche sont : "assez robuste" ou "pas assez robuste".
Par ailleurs, les auteurs ont proposés une deuxième approche basée sur la génération des tests à partir de la spécification dégradée pour générer les cas de test de robustesse.

4.2.4 Autres approches

Quelques prospectives concernant le test de robustesse ont été enregistrées dans le cadre de l'action scientifique n°23 du CNRS, notons brièvement :

La prospective IRISA [33], fondée sur la disponibilité d'une spécification du système à tester et de ses propriétés de robustesse. Les auteurs emploient les techniques de synthèse de contrôleur pour empêcher la spécification d'atteindre des comportement non robustes. Ensuite, ils proposent d'exploiter des techniques déjà développées pour le test de conformité (par exemple TGV) pour générer les cas de test de robustesse.

Par ailleurs, la prospective proposée par le LaBRI [33] est fondée sur l'extension du test de conformité, plus précisément, un testeur de robustesse permet de vérifier que l'IUT d'une part a un comportement correct en accord avec la spécification et d'autre part il détecte des événements corrompus ou inconnus (provenant de l'environnement externe). A partir de ce testeur, il est possible de dériver des tests abstraits. Plusieurs étapes sont considérées pour la construction du testeur de robustesse : construction du graphe de refus de la spécification pour avoir l'ensemble des refus associés à chaque état (déterminisation des situations de blocages), complétion du graphe de refus avec des traces de fautes sur

chaque état et ajout d'un événement générique θ représentant les événements inconnus ou corrompus (le testeur passe dans un état de refus qui représente une situation dégradée et revient soit dans l'état de départ avec une transition non observable, soit dans un état de reprise identifié) et enfin prise en compte des actions du graphe de refus étendu afin de générer les tests abstraits.

4.3 Conclusion

Dans ce chapitre nous avons exposé quelques méthodes et outils développés pour traiter certains aspects du test de robustesse des systèmes. Toutefois, l'utilisation du terme "robustesse" est souvent confondue avec les notions de conformité, de sécurité ou de sûreté de fonctionnement.

Les méthodes formelles, fondées sur un modèle de comportement, issues de l'AS23 ont défini les ingrédients de base pour formaliser le test de robustesse tels que la distinction entre "*le comportement acceptable*" et "*le comportement correct*", la classification des aléas vis-à-vis des frontières du systèmes et les notions "*mode dégradé*" et "*mode nominal*". En revanche, la représentation formelle d'aléas et la méthodologie de construction d'un modèle de référence pour le test de robustesse (spécification dégradée ou augmentée) ne sont pas assez étudiées.

Nous pensons que le test de robustesse devrait s'appuyer sur un modèle de référence décrivant le comportement d'un système en présence d'aléas. En conséquence, les méthodes de génération des tests de robustesse doivent utiliser des critères de sélection visant à dériver uniquement des traces d'exécution produites par les aléas. Enfin, nous signalons l'absence d'une véritable étude de cas permettant de positionner proprement le test de robustesse par rapport aux autres types de test.

Chapitre 5

Un cadre formel pour le test de robustesse

L'automatisation du test de robustesse impose la formalisation mathématique de certains aspects du test. Tout d'abord, la spécification doit être formelle, afin que les comportements qu'elle décrit soient suffisamment précis et non ambigus. D'autres aspects doivent aussi être formalisés : les aléas, les comportements en présence d'aléas, la relation de robustesse entre l'IUT et sa spécification ainsi que les modèles de fautes issus de cette relation et, les interactions entre le testeur et l'IUT. Dans ce chapitre, nous proposons un cadre formel pour automatiser le test de robustesse appliqué aux protocoles de communication. Les résultats de ce chapitre sont publiés dans [166, 163, 167].

Le chapitre est organisé comme suit : La section 5.1 met l'accent sur la notion de robustesse. La section 5.2 expose les points de différence ainsi que les nouveaux enjeux comparés au test de conformité. La section 5.3 propose une définition, une classification et une représentation des aléas relative au domaine des protocoles. La section 5.4 donne le plan général de l'approche proposée, ainsi que les concepts formels utilisés par la suite. Les sections 5.5 et 5.6 détaillent les deux méthodes proposées. La section 5.7 propose une relation pour tester la robustesse d'une implémentation, ainsi que les modèles de fautes visés par cette relation. Enfin, la section 5.8 conclut le chapitre.

5.1 Notion de robustesse

Le test de robustesse constitue un domaine d'étude relativement récent aussi sa définition est-elle encore imprécise et sujette à interprétation. Dans la littérature, la notion de robustesse est quelquefois confondue avec celle de sûreté de fonctionnement ou avec celle de conformité. En particulier, si on considère que le test de robustesse ne peut s'appliquer qu'à un système qui fonctionne correctement, dans les conditions nominales, il est alors un sur-ensemble du test de conformité. En revanche, si on considère que la robustesse est

définie par rapport à la réalisation d'une propriété, il peut être nécessaire de modifier la spécification afin d'assurer cette propriété, par exemple en ajoutant des comportements dégradés, ce qui peut être orthogonal à la conformité. Dans ce dernier cas, le test de robustesse et le test de conformité sont deux domaines bien distincts. Ces constatations ont conduit à deux principales définitions de la robustesse ; elles sont rappelées ci-dessous.

Définition 12 (La robustesse selon l'IEEE) *est le degré selon lequel un système, ou un composant, peut fonctionner correctement en présence d'entrées invalides ou de conditions environnementales stressantes [88].*

Cette définition est très vague parce que, dans la plupart des cas, le comportement du système en présence de fautes ou de conditions de stress n'est pas assez spécifié ou entièrement omis de la spécification. Ceci est justifié par l'impossibilité de prévoir toutes les fautes et les conditions de stress provenant d'un environnement préalablement inconnu. Par conséquent, si le système montre un comportement différent du comportement attendu alors ceci pose un problème pour l'acceptation ou le refus de ce comportement (absence de référence). Un exemple illustratif de cette ambiguïté est un distributeur d'argent implanté à partir de la spécification suivante : "*un utilisateur identifié retire l'argent après avoir saisi son code secret et choisi un montant autorisé*". Si le distributeur rejette la carte bancaire après l'épuisement de l'argent, ce comportement est alors orthogonal avec celui de la spécification nominale. Naturellement, ce comportement est peut être acceptable même s'il est orthogonal avec la spécification.

Dans la définition précédente, la spécification incomplète pose un véritable problème pour l'évaluation du comportement correct du système dans les situations imprévues. Partant de cette problématique, les auteurs de l'action scientifique N°23 du CNRS [33] ont proposé la définition suivante :

Définition 13 (La robustesse selon l'AS 23) *Le test de robustesse vise à vérifier la capacité d'un système, ou d'un composant, à fonctionner de façon acceptable, en dépit d'aléas.*

Dans cette définition, l'utilisation du terme générique "*aléa*" désigne à la fois les entrées invalides et les conditions environnementales stressantes. L'expression "*fonctionner de façon acceptable*" indique que les comportements attendus par le test de robustesse peuvent être incorrects, c'est à dire non conformes à ceux qui sont attendus dans la spécification nominale. En conséquence de cette contradiction, nous avons décidé de définir la robustesse de la manière suivante :

Définition 14 *Un système est considéré comme robuste s'il est d'abord conforme à sa spécification nominale et, qui prouve un comportement acceptable en dehors de ses conditions nominales (en présence d'aléas).*

Nous pensons d'abord que le test de robustesse devrait s'appliquer à des systèmes conformes à leur spécification nominale, c'est à dire la description du service attendu du

système dans les conditions normales. Le comportement acceptable désigne tout comportement qui peut être toléré par les concepteurs et les utilisateurs du système. Pour éviter toutes sortes d'ambiguïté, ce comportement devrait être spécifié ou défini par les concepteurs du système. Nous proposons par la suite un formalisme permettant de faciliter cette tâche.

5.2 Positionnement du test de robustesse

Nous rappelons que le test consiste à exécuter le système à tester, celui-ci prenant un ensemble d'entrées et fournissant un ensemble de sorties comme résultats. La justesse du système est alors déterminée en comparant les résultats obtenus aux résultats décrits dans la spécification (cahier de charge, standard du protocole, etc). Deux approches sont envisageables pour la construction des tests : l'approche *"boîte noire"* ou l'approche *"boîte blanche"*. Dans l'approche boîte noire, la spécification joue un rôle central pour la sélection des entrées du test et aussi pour l'évaluation des résultats obtenus.

La spécification du système décrit habituellement la fonction ou le service attendu du système dans les conditions normales (règles de communication, format de messages, etc). Lorsque l'on considère des systèmes critiques ou de sécurité, la spécification est généralement complétée par l'énoncé de ce qui ne devrait pas arriver (par exemple, les états dangereux pouvant mener à une catastrophe, ou à la divulgation d'informations sensibles). Cette spécification conduit à définir un ensemble de fonctions additionnelles pour éviter les défaillances (par exemple identification d'un utilisateur et vérification de ses droits). Par ailleurs, la spécification peut être exprimée selon divers points de vue ou degrés de détail. Elle peut être aussi décomposée selon la présence ou l'absence de défaillance (mode dégradé, mode nominal). Ces constatations peuvent conduire à distinguer plusieurs types de test dependant du degré de détail dans la spécification et de contraintes environnementales d'exécution.

Dans le domaine des protocoles de communication, la spécification est en général une description rédigée en langue naturelle (spécification CCITT par exemple) ou en langage spécialisé (SDL, Lotos, etc). Elle décrit souvent les règles de communications distantes dans les conditions normales. En d'autres termes, elle définit un certain nombre de messages permettant au système de dialoguer avec son environnement. Ces messages sont souvent présentés sous une forme structurée, regroupant des champs, eux-mêmes subdivisés en d'autres champs et ainsi de suite, de longueurs fixes ou variables, obligatoires ou optionnels. Dans certains cas, la spécification décrit aussi quelques règles additionnelles pour éviter certaines défaillances (par exemple, messages d'erreur).

A partir de cette spécification, le testeur peut se concentrer uniquement sur les *fonctions de base* (les fonctions attendues sans aucune considération de fautes). Dans ce cas, le testeur dérive *une spécification de base* qui servira comme une référence pour le test de conformité.

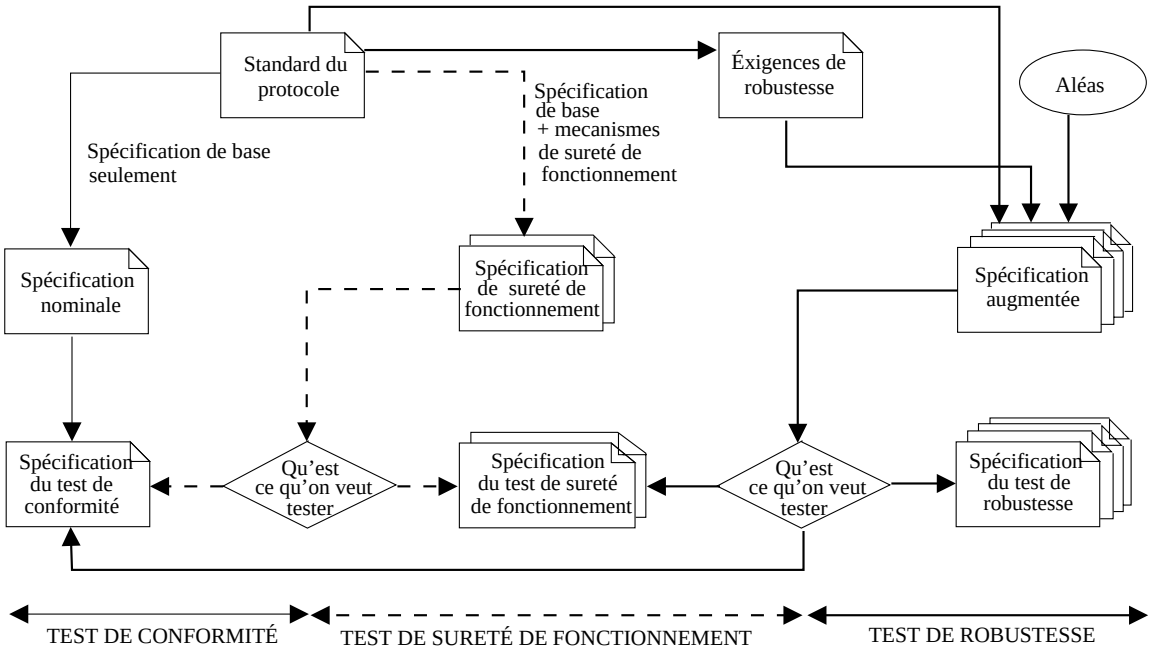


FIG. 22: Le degré de détail dans la spécification permet de distinguer plusieurs types de test

Parallèlement, la considération des fautes spécifiées conduit à dériver une spécification de base enrichie par les fonctions décrivant les *mécanismes de sûreté de fonctionnement* (voir FIG.22). Le test basé sur cette dernière spécification est appelé *test de sûreté de fonctionnement*. Ce terme est peu utilisé dans le domaine du test de protocoles et, on parle souvent du test de conformité pour décrire l'ensemble des tests qui peuvent être dégagés directement à partir de la spécification du protocole.

Par la suite, nous appellerons une *"spécification nominale"* la description de tout ce qui est spécifié préalablement par les concepteurs du système. Notons aussi que la spécification parfaite ou exhaustive est impossible. Naturellement, la spécification est initialisée dès les premières phases du cycle de vie d'un système et elle se poursuit généralement tout au long de la vie du système, en raison de la difficulté à identifier ce qui est attendu d'un système et notamment dans un environnement inconnu.

Pour ce qui concerne le test de robustesse, nous avons choisi de le considérer comme une extension (sur-ensemble) du test de conformité. Ce type de test vise à vérifier la capacité d'un système à fonctionner de façon acceptable en présence d'aléas. La spécification utilisée comme une référence pour ce test est appelée "*la spécification augmentée*". Elle est obtenue par l'intégration d'aléas représentables (les entrées et les sorties qui exhibent les aléas) dans la spécification nominale. Toutefois, si on considère les aléas non représentables (voir la section 5.3), la spécification nominale peut être utilisée, dans certains cas, comme une référence pour le test de robustesse (si on cherche à vérifier l'exécution de quelques fonctions nominales en présence d'aléas), la seule différence dans ce cas se trouve au niveau de

l'exécution de l'implémentation sous test. Plus précisément, dans le test de conformité, l'environnement est supposé normal et sûr ; par opposition, dans le test de robustesse on vise à soumettre le système à des situations critiques ou inattendues, et à vérifier s'il est capable de réaliser les fonctions qu'on attend de lui.

Dans la littérature, on peut trouver d'autres classifications de la "robustesse" et du "test de robustesse". Nous rappelons particulièrement certains travaux menés au LAAS définissent la robustesse comme un attribut secondaire de sûreté de fonctionnement : *"la robustesse est un exemple d'un attribut secondaire spécialisé de la sûreté de fonctionnement, c'est à dire la sûreté de fonctionnement en présence de fautes externes caractérisant les réactions du système face à certaines classes spécifiques des fautes"* [10]. Par ailleurs, les auteurs de [123] proposent un modèle globale des principales activités nécessaires pour le développement d'un système sûr de fonctionnement. Trois processus ont été proposés : processus de *création*, processus d'*élimination de fautes* et processus de *prévision de fautes*. Dans ce modèle, le test de robustesse est considéré comme un moyen *réalisation* durant le processus d'*élimination de fautes*.

5.2.1 Les nouveaux enjeux comparés au test de conformité

Pour mieux comprendre les spécificités du test de robustesse, nous fondons sa définition par rapport à celle du test de conformité utilisant un modèle ne prenant pas en compte les aléas (voir Fig.23), et nous analysons les écarts induits par la considération de ces derniers. En conséquence, les aléas et la définition du comportement acceptable font deux points de différenciation possibles : 1) le domaine d'entrées n'est pas nécessairement le même et, 2) le comportement de l'implémentation sous test peut être observé et évalué selon un point de vue différent.

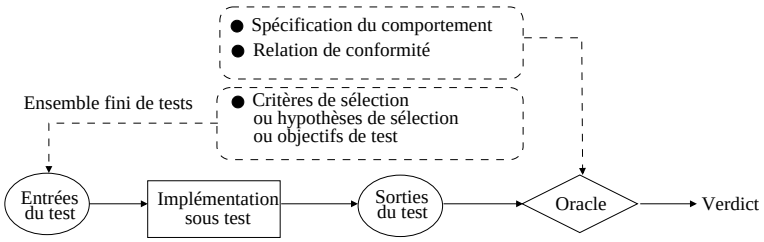


Fig. 23: Architecture du test de conformité

En conclusion de cette comparaison, une série de questions nécessite des réponses claires afin de définir proprement le test de robustesse :

- Comment spécifier le comportement du système en présence d'aléas ?
- Comment élargir le domaine d'entrées ?
- Comment modifier le domaine de sorties ?
- Comment définir la relation de robustesse ?

– Comment définir les critères de sélection, les hypothèses de sélection ou les objectifs de test de robustesse ?
– Comment évaluer les résultats du tests (les verdicts) ?

Dans ce document, nous proposons des réponses à toutes ces questions. Nos réponses sont fondées sur notre définition de la robustesse.

5.3 Aléas

Il y a de nombreuses définitions possibles d'un aléa. Du point de vue du test de robustesse, nous considérons la définition suivante :

Définition 15 *Le terme "aléa" désigne tout événement qui n'est pas spécifié dans la spécification nominale du système. Les aléas peuvent, dans certains cas, provoquer l'impossibilité temporaire ou définitive d'un système à exécuter une action.*

Dans cette section, nous proposons une extension de la classification des aléas donnée dans l'AS 23 [33]. Notre classification, comme nous le précisons par la suite, considère la situation des aléas vis-à-vis du testeur.

5.3.1 Situation vis-à-vis des frontières du système

Cette classification est adoptée par l'AS23. Les auteurs ont proposé de distinguer les aléas selon leur situation par rapport aux frontières du système. Trois classes de base sont issues de cette classification (voir la FIG.24) :

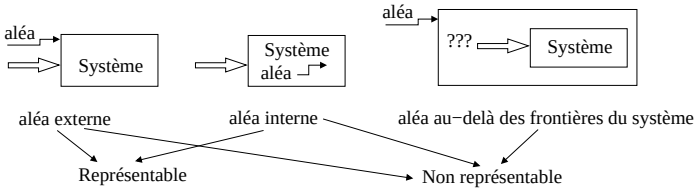


FIG. 24: Classification d'aléas selon l'AS 23

1. *Aléas internes* décrivant tout ce qui peut arriver, accidentellement ou intentionnellement, à l'intérieur du système. Dans l'exemple d'un distributeur de billets, l'épuisement de billets, encre, papier d'impression et la défaillance d'un circuit électronique sont des aléas internes.
2. *Aléas externes* décrivant tout ce qui peut arriver, accidentellement ou intentionnellement, dans l'environnement extérieur et qui a un impact direct sur l'interface du système. Par exemple, insertion d'une carte défectueuse, saisie d'un code invalide, saisie d'une somme non autorisée, etc.

3. *Aléas au-delà des frontières du système* décrivant tout ce qui peut arriver, accidentellement ou intentionnellement, dans l'environnement extérieur et qui n'a pas d'impact direct sur l'interface du système. Par exemple, baisse de température, hausse d'humidité, coupure d'accessibilité au distributeur, etc.

Par ailleurs, les auteurs de l'AS23 ont reclassé les aléas internes, externes et au-delà des frontières du système selon leur représentabilité dans un modèle formel. Ceci a abouti par l'identification de cinq classes d'aléas : aléas externes représentables, aléas externes non représentables, aléas internes représentables, aléas internes non représentables et aléas non représentables au-delà des frontières du système.

5.3.2 Situation vis-à-vis du testeur

Intuitivement, le testeur de robustesse vise à exécuter le système en présence d'aléas ou, à injecter les aléas dans l'interface du système et, à comparer ensuite les sorties observées à celles décrites dans la référence. En conséquence, le testeur de robustesse devrait être capable de 1) contrôler l'exécution d'aléas et, 2) observer et interpréter les réponses ou les sorties fournies par le système en présence d'aléas. Plus précisément, les aléas peuvent être classés selon les trois points de vue suivants :

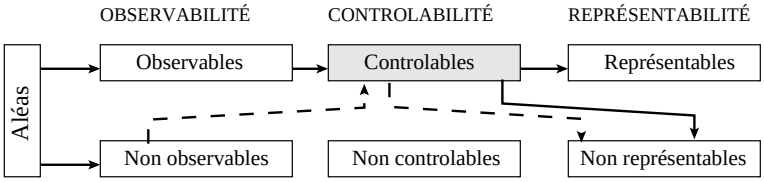


Fig. 25: Classification des aléas selon le point de vue du testeur.

Observabilité. On distingue des aléas observables par le testeur (entrées ou sorties) et d'autres non observables ou internes. Si on considère l'exemple d'un distributeur d'argent, les interactions utilisateur/distributeur (les entrées) : insertion d'une carte défectueuse, saisie d'un code incorrect, sélection d'une somme non autorisée, ainsi que les interactions distributeur/utilisateur (les sorties) : absence d'affichage, billets falsifiés, ticket vide sont des aléas observables par le testeur. Ainsi, le testeur peut observer certaines conditions environnementales stressantes, par exemple l'éclairage faible, la mauvaise visibilité d'affichage, l'accessibilité difficile au distributeur et certains comportements douteux d'utilisateurs. En revanche, les problèmes liés à la préparation de billets, à la vérification du code et à l'impression du ticket peuvent être considérés comme des aléas internes et par conséquent, ils ne sont pas observables par le testeur. Notons que l'observabilité d'aléas dépend de leur situation vis-à-vis les frontières du système, ce qui implique qu'un aléa interne est par défaut invisible ; par opposition à un aléa externe qui est observable par le testeur.

Contrôlabilité. Le testeur peut contrôler les entrées invalides ou inopportunes injectées dans l'interface du système, mais aussi quelques conditions environnementales stressantes. A titre d'exemple, le testeur peut contrôler les entrées inattendues ou incorrectes : carte défectueuse, code incorrect, ainsi que les conditions environnementales stressantes : éclairage faible, visibilité réduite d'affichage, température élevée, accessibilité difficile au distributeur. En revanche, le testeur ne peut pas contrôler d'autres conditions environnementales comme : les problèmes liés à la préparation de billets ou à l'identification de l'utilisateur, etc.

Dans ce contexte, la contrôlabilité implique la capacité du testeur à activer ou à désactiver l'exécution d'aléas.

Représentabilité. Ce critère est lié à l'application des méthodes formelles dans le test. En effet, si on veut modéliser le fonctionnement d'un système, il est alors nécessaire de choisir un formalisme de description (langage de spécification, automate, etc). Ces modèles diffèrent par leur sémantique, mais aussi par leur syntaxe de description (par exemple, la distinction entre les entrées et les sorties, la représentation des contraintes temporelles ou la modélisation des données, etc). A titre d'exemple, on ne peut pas décrire la durée d'attente entre l'insertion de la carte et la saisie du code dans un modèle de description discret non temporisé.

Nous précisons que dans la suite de ce travail, la représentabilité d'aléas est liée aux caractéristiques du système de transitions étiquetées à entrée/sortie (IOLTS) qui sera utilisé comme un formalisme de base pour modéliser les spécifications des systèmes.

La combinaison entre les trois points de vue précédents identifie les classes d'aléas suivantes (voir FIG.25) :

1. ***Aléas observables, contrôlables et représentables***. Cette classe regroupe les entrées exceptionnelles ou inattendues du système. A titre d'exemple, les aléas issus de l'utilisation normale : saisie d'un code incorrect, insertion d'une carte périmée, saisie du code avant l'insertion de la carte, saisie d'une somme qui n'est pas multiple de 10 peuvent être considérés comme des aléas observables, contrôlables et représentables. Ainsi, le testeur peut observer, contrôler et représenter d'autres aléas issus de l'utilisation frauduleuse (par exemple, l'insertion d'une carte téléphonique à puce pour provoquer un déni de service). Cette classe est un sous-ensemble de la classe d'aléas externes donnée dans l'AS 23.
2. ***Aléas observables, contrôlables et non représentables***. Cette classe regroupe les conditions de stress et les pannes contrôlables par le testeur, c'est à dire le testeur est capable de manipuler leur exécution et, qui ne sont pas représentables à travers des entrées ou sorties. Par exemple, la hausse ou la baisse importante de température ce qui peut endommager les composants électroniques de la machine. Cette classe d'aléa est un sous ensemble d'aléas externes et au-delà des frontières du système données dans l'AS 23.

3. ***Aléas non observables, contrôlables et non représentables***. Cette classe regroupe certaines défaillances internes commandables par le testeur mais qui ne sont pas modélisables par des entrées/sorties dans un modèle formel. L'exemple illustratif de cette classe est : billets d'argent mal pliés, épuisement de billets, épuisement d'encre ou de papier. En effet, le testeur peut soumettre le distributeur à des situations pareilles malgré le fait qu'elles soient situées à l'intérieur du distributeur.

Les aléas non contrôlables sont éliminés car le testeur est incapable de gérer ou de provoquer leur apparition. De plus, les aléas non observables, contrôlables et représentables sont aussi éliminés parce que ceux-ci sont considérés comme des actions internes dans le modèle IOLTS que nous utilisons pour modéliser les spécifications (transitions étiquetées par τ). Cette modélisation est inutile parce que le test est focalisé seulement sur les entrées et sorties observables du système.

5.3.3 Les aléas dans le domaine de protocoles

Le caractère inconnu, incertain ou même malveillant d'environnements d'exécution pose un véritable obstacle pour la prévision d'aléas. En conséquence, l'idée de prévoir les aléas pour tester la robustesse d'un système devient de plus en plus coûteuse, voire irréalisable. Afin de développer une approche raisonnable pour le test de robustesse, nous proposons de se concentrer sur la prévision d'effets ou d'influences possibles d'aléas sur l'ensemble d'entrées/sorties du système (par exemple, modification d'un message d'entrée/sortie, la modification d'alternance prévue de messages). L'application de cette idée dépend essentiellement de type du système à tester et surtout de son architecture.

Étant intéressés particulièrement par les protocoles de communication, nous considérons les types d'influences suivants :

Réception d'entrées invalides. L'occurrence d'aléas dans les réseaux de communication (par exemple, défaillance d'un routeur, redémarrage d'un serveur, tentative de pénétration) peut produire des fautes, accidentelles ou intentionnelles, dans le contenu de différents messages de communication. Par exemple, messages contenant certains champs endommagés ou erronés. De plus, il se peut qu'une entité d'un protocole reçoive des messages inconnus (par exemple, messages provenant d'un utilisateur mal intentionné visant à provoquer un déni de service). Par la suite, cette classe d'influence d'aléas sera appelée : *les entrées invalides.*

Définition 16 (Entrée invalide) *On considère comme une entrée invalide toute entrée :*

1. *qui n'est pas spécifiée dans la spécification nominale du système ou,*
2. *spécifiée et erronée (messages connus par le protocole, mais contenant certaines fautes).*

Réception d'entrées inopportunes. Certains aléas peuvent provoquer des retards dans

les réseaux ou même la duplication des messages échangés. Dans certains cas, la réception d'un message imprévu peut modifier l'exécution courante de l'implémentation, voire, l'arrêter complètement. Par opposition à la classe d'aléas précédente, les messages reçus ici sont syntaxiquement corrects, mais seulement leur réception est inappropriée, c'est à dire les messages en retard ou même en avance (des messages provenant d'une ancienne connexion par exemple). Par la suite, cette classe d'influence d'aléas sera appelée : *les entrées inopportunes.*

Définition 17 (Entrée inopportune) *On considère comme entrée inopportune toute entrée syntaxiquement correcte, spécifiée dans la spécification nominale du système, mais sa réception n'est pas attendue dans l'état courant du système.*

Émission des sorties inattendues. Tenir compte des aléas peut mener, dans certains cas, à observer quelques sorties additionnelles émises par le système. Par exemple, émission d'un message inattendu (fermeture exceptionnelle d'une connexion, re-émission d'une demande de synchronisation, etc), erroné ou inconnu. Cette classe d'influence d'aléas sera appelée : *les sorties inattendues.*

Définition 18 (Sortie inattendue) *On considère comme sortie inattendue toute sortie :*

1. *spécifiée, mais son émission n'est pas attendue dans l'état courant du système ou,*
2. *qui n'est pas spécifiée.*

Certaines sorties inattendues peuvent être considérées comme des comportements acceptables. Pour éviter les problèmes d'oracle liés à l'observation de ces sorties, nous proposons d'ajouter toutes les sorties acceptables dans la spécification nominale.

Influence non représentable ou imprévisible. Toutefois, l'influence d'aléas sur l'ensemble d'interactions (entrées/sorties) entre les entités protocolaires peut être non représentable dans le formalisme de description utilisé, voire imprévisible. Ceci est justifié en général par : 1) la complexité d'aléas, 2) leur impact indirect sur l'interface du système (aléas au-delà des frontières du système) ou 3) la limitation du formalisme utilisé à représenter leur influence. En conséquence, pour tester la robustesse en présence d'aléas contrôlables et non représentables, il ne reste qu'à considérer physiquement ces aléas dans le test, c'est à dire soumettre le système physiquement à leur influence.

En particulier, dans le domaine des protocoles modélisables par des systèmes de transitions étiquetées (IOLTS), ce type d'aléas peut être illustré par la défaillance de certaines routines de système d'exploitation ou mêmes par certains bugs provenant du système applicatif. Précisément, si on considère le testeur de robustesse comme une entité communicante avec l'entité protocolaire sous test (IUT), le testeur est peut être incapable d'observer les interactions internes entre l'IUT et son système d'exploitation, mais aussi les interactions entre l'IUT et le système applicatif.

5.4 Architecture de l'approche proposée

Partant de notre définition de la robustesse et de la classification précédente d'aléas, nous proposons une approche composée de deux méthodes, l'une complète l'autre, pour formaliser le processus du test de robustesse.

La méthode TRACOR (*Test de Robustesse en présence d'Aléas COntrôlables et Représentables*) automatise le test de robustesse en présence d'aléas contrôlables et représentables. Elle est fondée sur l'insertion d'entrées invalides, entrées inopportunes et sorties acceptables, ainsi que les traces de suspension dans la spécification nominale. La nouvelle spécification, obtenue après la déterminisation, est appelée "*la spécification augmentée*", elle servira comme une référence pour le test de robustesse en présence de cette classe d'aléas.

La méthode TRACON (*Test de Robustesse en présence d'Aléas COntrôlables et Non représentables*) automatise le test de robustesse en présence d'aléas contrôlables et non représentables. Elle est basée sur l'ajout de sorties acceptables dans la spécification nominale, sans aucune considération d'entrées inopportunes ou invalides. La description obtenue est appelée *la spécification semi-augmentée*, elle servira comme une référence pour le test de robustesse en présence de cette classe d'aléas. Avant de détailler chaque méthode, nous exposons d'abord quelques définitions et concepts de base.

5.4.1 Modèle de spécification

Les protocoles de communication sont spécifiés en général dans des langages de description spécialisés (par exemple SDL, Lotos, UML, IF). La sémantique opérationnelle de ces langages, qui définit les comportements possibles de la spécification, est parfois décrite dans le modèle LTS. Les états représentent les configurations possibles (états de contrôle, valeurs de variables, contenus des files, etc.) et, les actions sont les transitions atomiques.

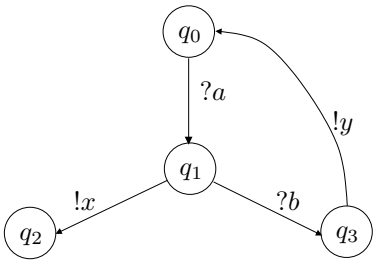


Fig. 26: Un exemple d'un IOLTS S

Dans l'approche du test boîte noire, nous avons besoin de distinguer les entrées recevables par une implémentation sous test (IUT), mais aussi les sorties émises par celle-ci. En conséquence, l'interprétation de la sémantique LTS en IOLTS est indispensable, ceci

nécessite parfois un découpage de transitions et d'identifier les entrées, sorties et actions internes.

Nous supposerons par la suite que les comportements de la spécification sont modélisés par un seul IOLTS $S = (Q, q_0, \Sigma, \rightarrow)$ et on ne fait a priori aucune hypothèse particulière sur cet IOLTS. De plus, on ne fait aucune distinction entre le "mode nominal" et "mode dégradé". En effet, cette distinction est moins fréquente dans les standards des protocoles.

Notations additionnelles.

Soit un IOLTS $S = (Q, q_0, \Sigma, \rightarrow)$ tel que Σ est l'alphabet d'actions observables. Σ est partitionné en alphabet d'entrée Σ_I et alphabet de sortie Σ_O, q_0 est l'état initial et, $\rightarrow$ est la relation de transition,

Outre les notations standards définies dans la section 2.3, nous définissons les notations suivantes :

- $In(q)$ est l'ensemble d'entrées recevables (attendues) dans l'état q. Formellement, $In(q) = \{a \in \Sigma_I \ , \ q \xrightarrow{a}\}$.
- $ref(q) = \{\Sigma_I / In(q)\}$ est l'ensemble d'entrées non recevables (inopportunes) dans l'état q.

Exemple 5.1 *Dans l'IOLTS S de la* FIG.*26 :*

- $In(q_0) = \{?a\}$, $In(q_1) = \{?b\}$ *et* $In(q_2) = In(q_3) = \emptyset$.
- $ref(q_0) = \{?b\}$, $ref(q_1) = \{?a\}$ *et* $ref(q_2) = ref(q_3) = \{?a, ?b\}$.

5.4.2 Modélisation du comportement acceptable

Dans le modèle du système de transitions étiquetées à entrée/sortie (IOLTS), les entrées invalides, les entrées inopportunes et les sorties acceptables sont définies formellement comme suit :
Soit un IOLTS $S = (Q, q_0, \Sigma, \rightarrow)$,

- Les entrées inopportunes dans un état $q \in Q$ sont données par l'ensemble $ref(q)$;
- Une entrée invalide désigne toute entrée n'est pas spécifiée dans l'alphabet Σ_I. Formellement, $?a'$ est une entrée invalide si $?a' \notin \Sigma_I$. On ne considère pas les entrées spécifiées et erronées car dans le modèle IOLTS, les entrées et les sorties sont supposées atomiques et par conséquent, on ne peut pas représenter directement les fautes sur les entrées. Pour ce faire, on considère les entrées erronées comme des actions hors de l'alphabet de la spécification nominale ;
- Les sorties acceptables dans un état $q \in Q$ peuvent être des sorties spécifiées mais leur émission n'est pas attendue dans cet état, mais aussi des sorties complètement non spécifiées. Formellement, $!x'$ est une sortie acceptable si $!x' \notin \Sigma_O$ ou $!x' \in \Sigma_O \ \wedge \ !x' \notin Out(q)$.

Comme nous l'avons mentionné dans la section 5.1, les concepteurs (spécificateurs) du système doivent préciser ce que signifie le comportement acceptable en présence d'aléas. La description fournie est en général moins détaillée que la spécification nominale, il s'agit dans la plupart des cas de l'ajout de quelques fonctions additionnelles qui ramènent le système dans un état de fonctionnement sûr afin d'éviter les risques ou les défaillances, par exemple la fermeture exceptionnelle d'un service, fermeture d'une connexion ou redémarrage d'une session. Ces fonctions peuvent être identiques dans plusieurs états du fonctionnement du système. De plus, ces fonctions peuvent être définies vis-à-vis de plusieurs aléas. En conséquence, l'utilisation d'un IOLTS pour modéliser le comportement acceptable n'est pas très raisonnable à cause de la taille (nombre d'états et de transitions). Pour ce faire, nous proposons ci-dessous un nouveau formalisme, appelé *meta-graphe*, permettant de modéliser le comportement acceptable du système en présence d'entrées invalides, entrées inopportunes et sorties acceptables.

Meta-graphe

Le meta-graphe est une structure qui complète la spécification nominale. Il sera employé pour modéliser le comportement du système en présence d'aléas. Les états de ce graphe sont appelés *meta-états*. Un meta-état est associé à un ensemble d'états de l'IOLTS de la spécification ayant le même comportement en présence des mêmes aléas.

Définition 19 *Soit $S = (Q, q_0, \Sigma, \rightarrow)$ un IOLTS. Le meta-graphe associé à S est le triplet $G = (V, E, L)$ défini par :*

- *$V = V_d \cup V_m$ un ensemble d'états. $V_m \subseteq 2^Q$ est appelé ensemble de meta-états et V_d est appelé ensemble d'états dégradés tel que $V_d \cap Q = \emptyset$.*
- *L un alphabet d'actions,*
- *$E \subseteq V \times L \times V$ un ensemble de transitions.*

Exemple 5.2 *Le meta-graphe G donné par la* Fig.*27 est associé à l'IOLTS S de la* Fig.*26. G possède deux meta-états (q_0, q_1, q_2, q_3) et (q_0) et, deux états dégradés d_1 et d_2. L'alphabet L est composé de deux entrées invalides $\{?a', ?b'\}$, une sortie acceptable $!x'$ et une entrée nominale $?a$.*

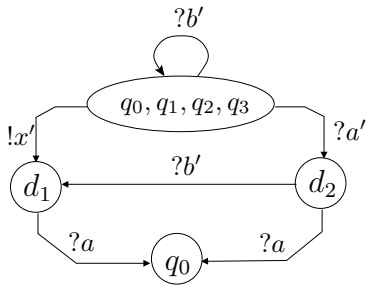


Fig. 27: Un exemple de meta-graphe (G)

Sémantique *: Dans chaque état de l'IOLTS S, si le système reçoit l'entrée invalide* $?a'$ *il se déplace alors à l'état dégradé* d_2*. En parallèle, si le système émet la sortie acceptable* $!x'$*, il se déplace à l'état dégradé* d_1*. En revanche, dans les états* q_0*,* q_1*,* q_2 *et* q_3*, si le système reçoit l'entrée invalide* $?b'$*, il reste alors dans le même état de départ. Dans les deux états dégradés* d_1 *et* d_2*, il est possible que le système retourne à son état initial s'il reçoit l'entrée nominale* $?a$*. Dans l'état dégradé* d_2*, si le système reçoit l'entrée invalide* $?b'$ *il se déplace dans l'état dégradé* d_1*.*

Classes d'équivalence d'aléas

Il est possible que les concepteurs veuillent que le système réagisse de la même façon en présence de plusieurs aléas. Ceci est modélisé, dans le meta-graphe, par plusieurs transitions entre les deux mêmes états (voir la FIG.28.*a*). Afin de minimiser la taille du meta-graphe, nous remplaçons cet ensemble d'aléas par une seule *classe d'équivalence*. Les transitions étiquetées par des classes d'équivalence sont appelées *meta-transitions*.

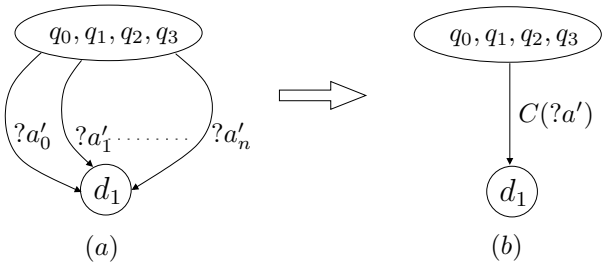


FIG. 28: Un exemple d'un meta-graphe avec une classe d'équivalence

Définition 20 *Soit un meta-graphe* $G = (V, E, L)$*. Une classe d'équivalence associée à l'ensemble d'aléas* a'_0,a'_1,...,a'_n *de* L *est* $C(?a') = a'_0, a'_1, ..., a'_n$ *tel que :* $\forall\ q_i, q_j \in V$ *,* $q_i \xrightarrow{a'_0} q_j \wedge q_i \xrightarrow{a'_1} q_j \wedge ... \wedge q_i \xrightarrow{a'_n} q_j$*. Les transitions* $q_i \xrightarrow{a'_0} q_j$*,* $q_i \xrightarrow{a'_1} q_j$,..., $q_i \xrightarrow{a'_n} q_j$*, dans le meta-graphe G, peuvent être remplacées par une seule transition* $q_i \xrightarrow{C(?a')} q_j$*.*

Exemple 5.3 *Dans les spécifications des protocoles, les entrées/sorties sont en général des messages regroupant des champs, eux-mêmes subdivisés en d'autres champs et ainsi de suite. Si on considère un message d'entrée* $M =< v_1, v_2, ..., v_k >$*, naturellement, la prévision d'entrées invalides relatives à* M *est basée sur l'insertion de fautes, de manière aléatoire ou probabiliste, dans un ou plusieurs champs de* M*, par exemple,* $M_1 =< f(v_1), v_2, ..., v_k >$*,* $M_2 =< v_1, f(v_2), ..., v_k >$*..., avec* $f(v_i)$ *est une mutation aléatoire du champ* v_i *. En conséquence, le nombre d'entrées invalides, obtenu par des mutations aléatoires d'un ou plusieurs champs, peut être infini. Si les concepteurs veulent que le protocole retourne à son état initial après la réception d'un message* M *erroné, alors ceci peut être modélisé tout simplement par une seule meta-transition étiquetée par la classe d'équivalence* $C(M) = \{M_1, M_2, ...\}$ *dans le meta-graphe.*

5.4.3 Architecture de la méthode TRACOR

Dans cette méthode, les concepteurs du système doivent fournir deux meta-graphes spécifiques modélisant le comportement acceptable du système en présence d'aléas contrôlables et représentables (entrées inopportunes, entrées invalides et sorties acceptables).
Le premier est appelé le *graphe d'aléas* et noté par HG (en anglais *hazard graph*). Ce meta-graphe modélise le comportement acceptable du système en présence d'entrées invalides.
Le deuxième est appelé le *graphe d'entrées inopportunes* et noté par IIG (en anglais, *inopportune input graph*). IIG modélise le comportement acceptable du système en présence de ses entrées inopportunes.
Les *sorties acceptables* décrivent les réponses du système à la réception d'entrées inopportunes ou invalides, elle sont modélisables directement dans HG et IIG.

Durant notre travail, nous avons rencontré une interrogation sur l'utilisation de deux meta-graphes, mais aussi sur la raison pour laquelle les concepteurs ne modélisent pas directement les aléas dans l'IOLTS initial de la spécification ?

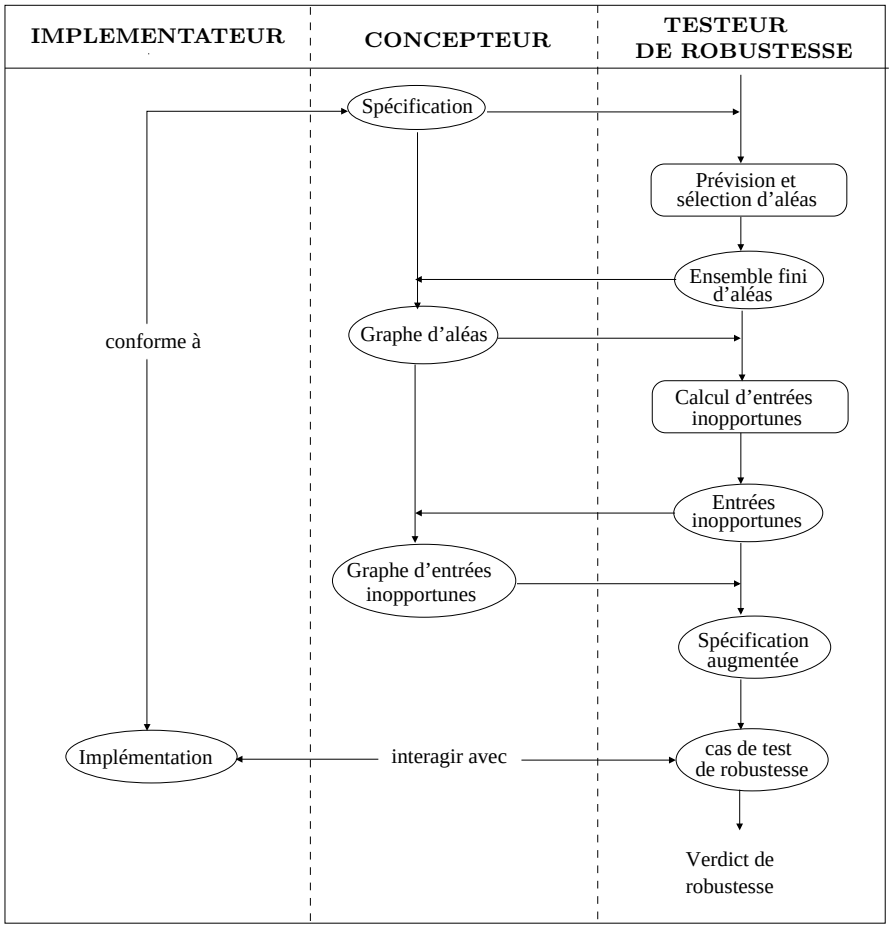


Fig. 29: Répartition de tâches entre les concepteurs et les testeurs de robustesse

Mettons les choses en ordre, la spécification nominale du système ou du protocole est déjà écrite et, les industriels l'ont utilisée pour implanter leurs produits (par exemple, implémentations différentes d'un protocole). Les testeurs de conformité ont validé ces produits par des tests de conformité. Maintenant, c'est le rôle des testeurs de robustesse pour évaluer la capacité de ces implémentations à fonctionner de façon acceptable en présence d'aléas. La répartition de tâches entre les testeurs de robustesse et les concepteurs du système est donnée dans la FIG.29.

1. Les testeurs de robustesse prévoient d'abord un ensemble fini d'influences d'aléas susceptibles de perturber le fonctionnement du système (dans ce cas, il s'agit d'un ensemble d'entrées invalides) ;
2. Les concepteurs précisent ensuite ce qu'ils attendent du système en présence de ces entrées invalides, leur réponse est formulée par *le graphe d'aléas* HG ;
3. Les testeurs utilisent *le graphe d'aléas* et la spécification nominale pour calculer les entrées inopportunes de chaque état du système et, ils demandent à nouveau aux concepteurs de préciser ce qu'ils attendent du système en présence d'entrées inopportunes. Leur réponse est formulée dans un deuxième meta-graphe, appelé le *graphe d'entrées inopportunes* IIG.
4. Les testeurs utilisent le graphe d'entrées inopportunes IIG pour construire la spécification augmentée qui servira comme référence pour la génération et la réalisation des tests de robustesse.

5.4.4 Architecture de la méthode TRACON

Dans la méthode TRACON, les testeurs prévoient un ensemble d'aléas contrôlables non représentables susceptibles d'entraver le fonctionnement du système (par exemple, baisse de température, hausse de pression, radiations électromagnétiques, coupures d'un réseau, etc). Ensuite, ils demandent aux concepteurs de spécifier le comportement acceptable du système en présence de cet ensemble d'aléas. Dans ce cas, la spécification du comportement acceptable est basée seulement sur la description *des sorties acceptables* en présence d'aléas car ceux-ci sont non représentables et par conséquent, les entrées ne changent pas. Du point de vue formel, leur réponse est formulée cette fois-ci par un meta-graphe appelé *graphe de sorties acceptables* AOG (en anglais, *acceptable output graph*).

Les testeurs de robustesse construisent ensuite une *spécification semi-augmentée*, celle-ci est obtenue par l'ajout de sorties acceptables et d'états dégradés à la spécification nominale. La spécification semi-augmentée servira comme une référence pour le test de robustesse en présence d'aléas contrôlables et non représentables. Afin de pouvoir exécuter les tests de robustesse, les aléas contrôlables non représentables sont supposés commandables par un *contrôleur* logiciel ou matériel HC (en anglais, *hazard controller*), par exemple, émetteur de radiations électromagnétiques. L'architecture du test de robustesse dans ce cas est donnée dans la FIG.30.

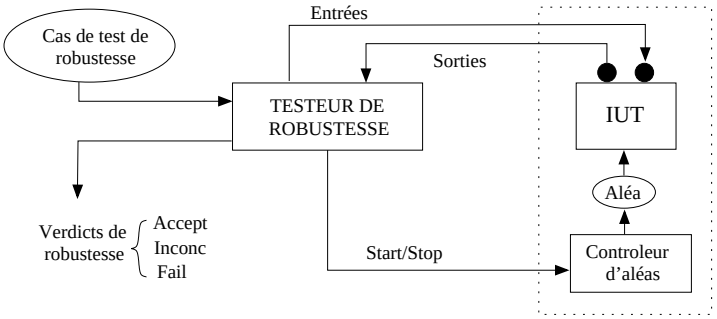


FIG. 30: Architecture de la Méthode TRACON

L'exécution du test de robustesse en présence d'aléas contrôlables et non représentables consiste donc à dériver des traces, appelées les cas de test, à partir d'un modèle de référence (spécification semi-augmentée ou spécification nominale) et, les appliquer sur une implémentation soumise aux aléas physiques. Le testeur compare ensuite les sorties observées à celles de la référence et, émet des verdicts traduisant la robustesse ou non de cette implémentation.

Signalons que la prévision d'aléas, dans les deux méthodes, est fondée sur l'analyse de la spécification (pour l'identification d'entrées inopportunes) et sur l'expérience concernant l'identification des anomalies provenant de l'environnement (pour définir les entrées invalides les plus susceptibles d'entraver le fonctionnement du système, mais aussi les aléas non représentables).

5.5 Méthode TRACOR

Comme nous l'avons mentionné dans le paragraphe 5.4.3, la méthode TRACOR (*Test de Robustesse en présence d'Aléas COntrôlables et Représentables*) est focalisée sur la formalisation du test de robustesse en présence d'entrées invalides, entrées inopportunes et sorties acceptables. Les différentes étapes de cette méthode sont données ci-dessous (la FIG.31) :

- **Phase 1** : Augmentation de la spécification. Cette phase regroupe les étapes suivantes :
  1. Ajout de traces de suspension,
  2. Intégration d'entrées invalides,
  3. Intégration d'entrées inopportunes,
  4. Mise à jour de traces de suspension et déterminisation.
- **Phase 2** : Génération des cas de test de robustesse. Cette phase consiste à dériver et à sélectionner des traces d'exécution assimilant les entrées invalides, entrées inopportunes et sorties acceptables. Brièvement, la génération des cas de test de robustesse

consiste à calculer, à partir de la spécification augmentée, des traces satisfaisant un besoin spécifique appelé "*objectif de test de robustesse*" RTP (en anglais, *robustness test purpose*). Cette phase est applicable sur les deux méthodes que nous proposons ; pour cette raison elle sera détaillée indépendamment dans le chapitre 6.

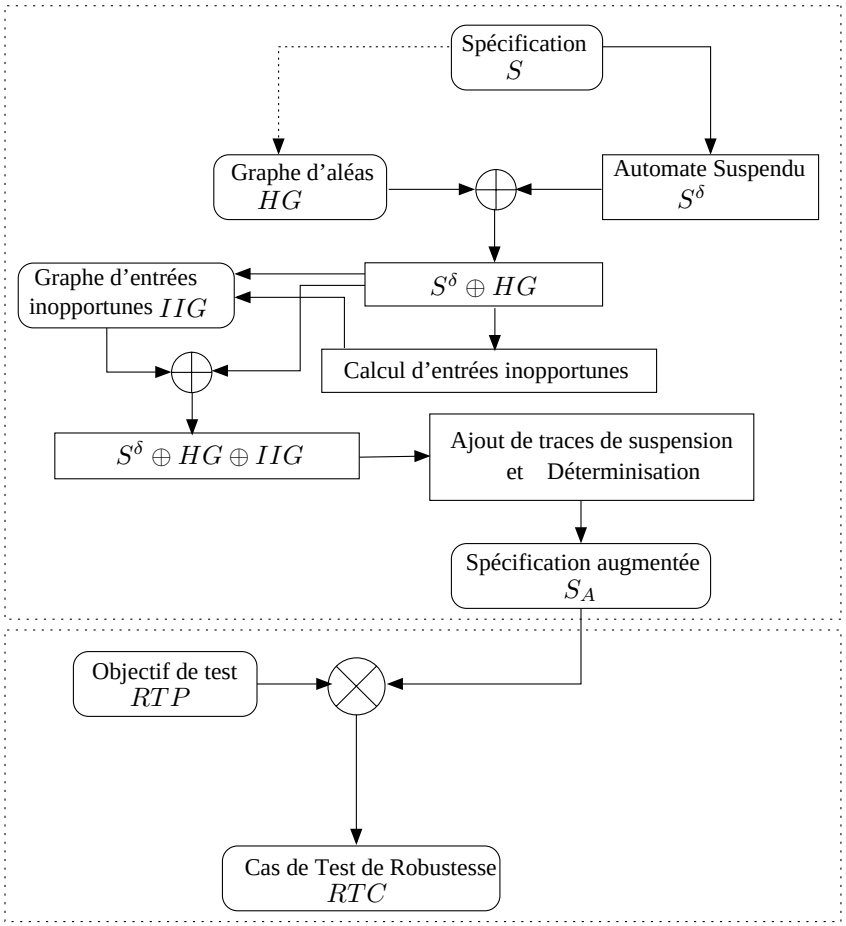


FIG. 31: Plateforme de la méthode TRACOR

5.5.1 Augmentation de la spécification

L'augmentation de la spécification consiste à intégrer les entrées inopportunes, entrées invalides, sorties acceptables et blocages valides dans le modèle de la spécification nominale. Ceci est résumé dans les étapes suivantes :

Ajout de traces de suspension

Dans la pratique du test, en plus des sorties, les tests permettent aussi de détecter les blocages (en anglais, *quiescence*) grâce à des temporisateurs (*timers*), armés en attente d'une réponse de l'IUT. On suppose que la valeur du temporisateur est suffisamment grande, pour que son expiration soit assimilée à un blocage de l'IUT. Dans le modèle IOLTS, trois types de blocage peuvent apparaître :

1. le *blocage de sortie* (ou *outputlock*) si le système est bloqué en attente d'une entrée provenant de l'environnement (par exemple, l'état q_0 de l'IOLTS de la Fig.32),
2. le *blocage complet* (ou *deadlock*) si le système ne peut plus évoluer (par exemple, l'état q_2 de l'IOLTS de la Fig.32),
3. le *blocage vivant* (ou *livelock*) si le système diverge avec une suite infinie d'actions internes.

Le testeur doit pouvoir distinguer entre un blocage de l'IUT valide (existant dans la spécification) ou non valide. Il faut donc modéliser les blocages possibles sur la spécification. Il suffit de rajouter, dans chaque état de blocage, une boucle étiquetée par une nouvelle action $!\delta$. L'action $!\delta$ est considérée comme une sortie de la spécification, puisqu'elle est observable mais n'est pas contrôlable par l'environnement. L'IOLTS résultant de l'ajout des actions $!\delta$ est appelé l'*automate suspendu* .

Définition 21 (Automate suspendu) *L'automate suspendu associé à l'IOLTS* $S = (Q^S, {q_0}^S, \Sigma^S, \rightarrow_S)$ *est un IOLTS* $S^\delta = ({Q^S}^\delta, {q_0}^{S^\delta}, {\Sigma^S}^\delta, \rightarrow_{S^\delta})$ *tel que :* ${\Sigma^S}^\delta = \Sigma^S \cup \{\delta\}$ *avec* $\delta \in \Sigma_O^{S^\delta}$. $\rightarrow_{S^\delta}$ *est obtenu à partir de* $\rightarrow_S$ *par l'addition d'une boucle* $q \xrightarrow{!\delta}_{S^\delta} q$ *pour chaque état de blocage.*

La modélisation des blocages dans la spécification doit s'accomplir dès la première étape de l'augmentation (Phase 1) parce que l'intégration d'aléas et particulièrement les sorties acceptables, ainsi que la déterminisation du modèle changent les informations de blocage.

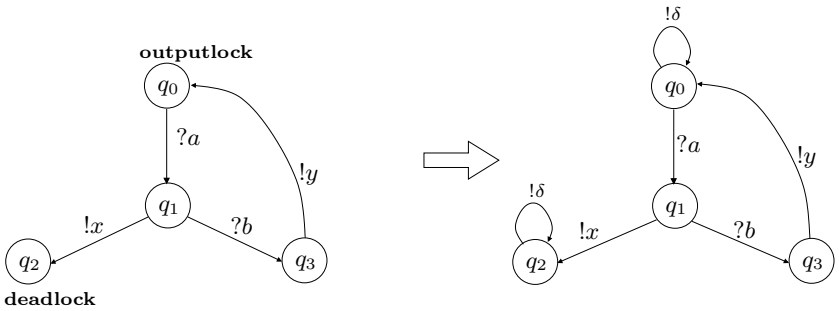


Fig. 32: L'automate suspendu associé à S

Intégration d'entrées invalides

Comme nous l'avons dit auparavant, les testeurs de robustesse prévoient un ensemble d'entrées invalides, décrivant l'influence d'aléas sur les entrées valides, et demandent aux concepteurs qu'ils précisent ce qu'ils attendent du système en présence de ces entrées invalides. On considère que leur réponse est formulée en terme d'un meta-graphe appelé *graphe d'aléas* et noté par HG (*hazard graph*).

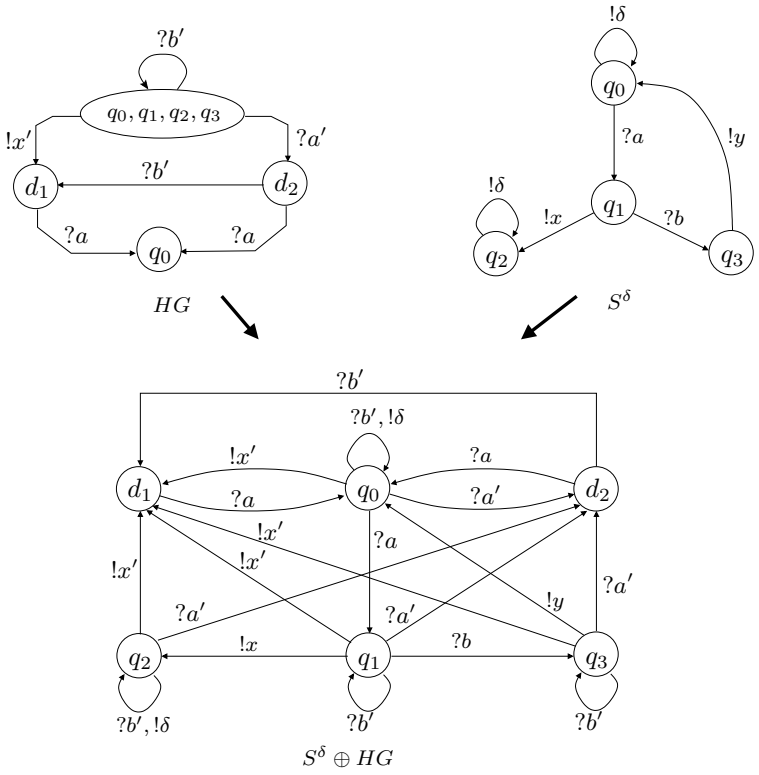


FIG. 33: Intégration d'entrées invalides dans S^δ

L'intégration d'entrées invalides se fait par la composition de HG avec l'IOLTS de la spécification, le résultat est $S^\delta \oplus HG$. La composition d'un meta-graphe avec un IOLTS est définie ci-dessous :

Définition 22 (Composition $IOLTS \oplus G$)
Soient $S = (Q^S, q_0^S, \Sigma^S, \rightarrow_S)$ un IOLTS et $G = (V, E, L)$ un meta-graphe associé à S. Rappelons que $V = V_m \cup V_d$. La composition de S et G, notée $S \oplus G$, est l'IOLTS $(Q^{S\oplus G}, q_0^{S\oplus G}, \Sigma^{S\oplus G}, \rightarrow_{S\oplus G})$ défini par : $Q^{S\oplus G} = Q^S \cup V_d$, $q_0^{S\oplus G} = q_0$ et les règles suivantes :

1. $q \xrightarrow{a}_S q' \implies q \xrightarrow{a}_{S\oplus G} q'$,
2. $(v, a, v') \in E$ *et* $v, v' \in V_d \implies v \xrightarrow{a}_{S\oplus G} v'$,
3. $(v, a, v') \in E$, $v \in V_m$ *et* $v' \in V_d \implies q \xrightarrow{a}_{S\oplus G} v'$ *pour chaque* $q \in v$,

4. $(v, a, v') \in E$, $v \in V_d$ *et* $v' \in V_m$ $\Longrightarrow$ $v \xrightarrow{a}_{S \oplus G} q$ *pour chaque* $q \in v'$,
5. $(v, a, v') \in E$ *et* $v, v' \in V_m$ $\Longrightarrow$ $q \xrightarrow{a}_{S \oplus G} q'$ *pour chaque* $q \in v$ *et* $q' \in v'$,
6. $(v, a, v) \in E$ *et* $v \in V_m$ $\Longrightarrow$ $q \xrightarrow{a}_{S \oplus G} q$ *pour chaque* $q \in v$.

Cette composition consiste à ajouter à S l'ensemble de transitions et états dégradés du meta-graphe G. En effet, pour un état q de S faisant partie d'un meta-état (ensemble d'états) v de G, on ajoute à S l'ensemble de transitions de G partant de v, ainsi que les états dégradés de G. Nous utiliserons cette composition pour intégrer les entrées invalides, entrées inopportunes et sorties acceptables exprimées sous forme de meta-graphes dans la spécification nominale.

Exemple 5.4 *Dans l'exemple de la* FIG.*33, l'application de :*

- *La règle 1 ajoute à* $S \oplus G$*, les transitions de* S^δ *:* ($q_0 \xrightarrow{?a} q_1$, $q_1 \xrightarrow{!x} q_2$, $q_1 \xrightarrow{?b} q_3$, $q_3 \xrightarrow{!y} q_4$, $q_0 \xrightarrow{!\delta} q_0$, $q_2 \xrightarrow{!\delta} q_2$),
- *La règle 2 ajoute à* $S \oplus G$*, la transition* $d_2 \xrightarrow{?b'} d_1$,
- *La règle 3 ajoute à* $S \oplus G$*, les transitions* ($q_0 \xrightarrow{?a'} d_2$, $q_1 \xrightarrow{?a'} d_2$, $q_2 \xrightarrow{?a'} d_2$, $q_3 \xrightarrow{?a'} d_2$, $q_0 \xrightarrow{!x'} d_1$, $q_1 \xrightarrow{!x'} d_1$, $q_2 \xrightarrow{!x'} d_1$, $q_3 \xrightarrow{!x'} d_1$),
- *La règle 4 ajoute à* $S \oplus G$*, les transitions* ($d_1 \xrightarrow{?a} q_0$, $d_2 \xrightarrow{?a} q_0$),
- *La règle 6 ajoute à* $S \oplus G$*, les transitions* ($q_0 \xrightarrow{?b'} q_0$, $q_1 \xrightarrow{?b'} q_1$, $q_2 \xrightarrow{?b'} q_2$, $q_3 \xrightarrow{?b'} q_3$).

La règle 5 n'est pas applicable puisqu'il n'y a pas des transitions entre les meta-états.

Intégration d'entrées inopportunes

Le test de robustesse d'un protocole devrait tenir compte de la réception de toute entrée inopportune ou inattendue dans tout état du système. Formellement, soit S un $IOLTS$, les entrées inopportunes de chaque état q de S sont calculées par la fonction $ref(q)$.

Dans cette étape, le testeur de robustesse calcule d'abord les entrées inopportunes de chaque état de la composition, $S^\delta \oplus HG$, obtenue dans l'étape précédente. Ensuite, il demande aux concepteurs du système qu'ils spécifient le comportement acceptable du système en présence de ses entrées inopportunes. On suppose que leur réponse est formulée par le meta-graphe IIG (*inopportune input graph*).

Définition 23 (Le graphe d'entrées inopportunes) *Le graphe d'entrées inopportunes est un meta-graphe* $IIG = (V, E, L)$ *tel que :*

- $V = V_m \cup V_d$*. L'ensemble d'états est composé d'un sous ensemble de meta-états* (V_m) *et un sous ensemble d'états dégradés* (V_d).
- $L = L_I \cup L_O$*. Son alphabet est composé d'alphabet d'entrée* (L_I) *et un alphabet de sortie* (L_O). $L_I \subseteq {\Sigma_I}^S$*, c'est à dire l'alphabet d'entrée contient que des entrées de la spécification nominale.* L_O *est l'ensemble de sorties acceptables produites par la réception des entrées inopportunes.*

– $\forall v \in V_m, \forall q \in v, \forall a \in ref(q) \Longrightarrow q \xrightarrow{a} \in E$, *c'est à dire chaque état q faisant partie d'un meta-état de IIG doit être complètement spécifié vis-à-vis de l'ensemble d'entéres inopportunes* $ref(q)$.

L'intégration d'entrées inopportunes est obtenue par la composition $S^{\delta} \oplus HG$ avec IIG suivant les mêmes principes de la composition donnés dans la définition 22.

Remarque 1 *Signalons que les meta-graphes HG et IIG peuvent être non connexes. Cette particularité dépend du comportement spécifié.*

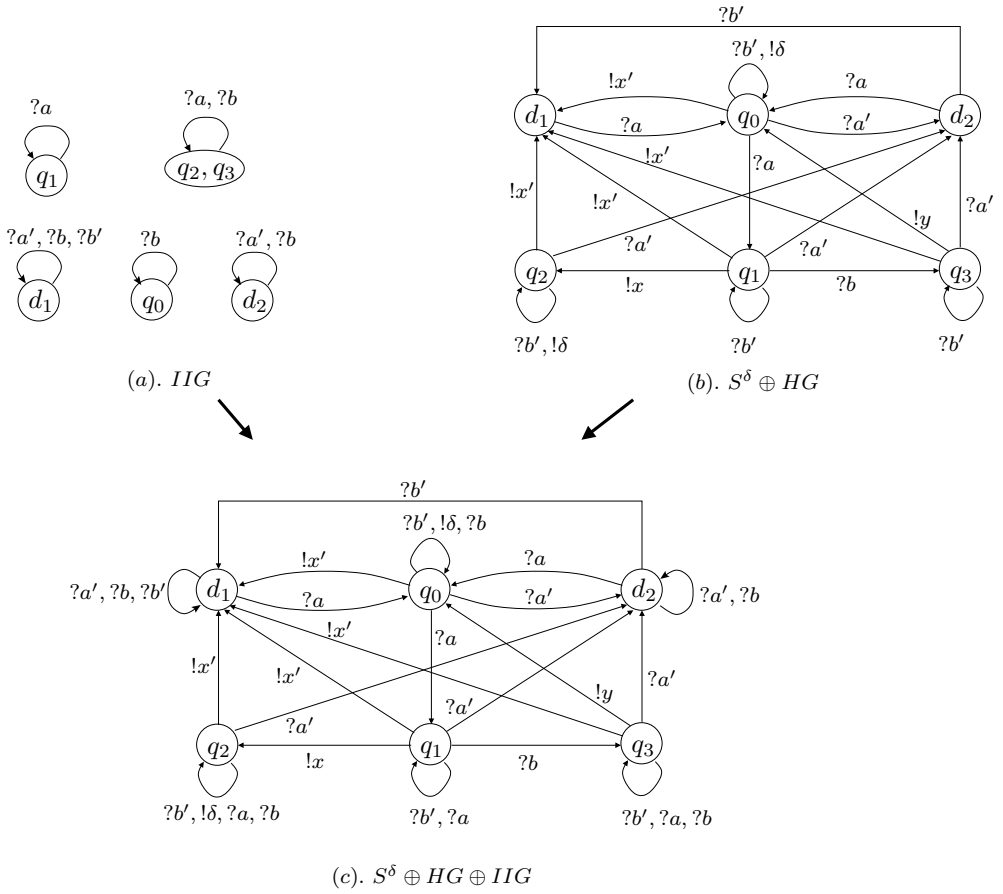


FIG. 34: Intégration d'entrées inopportunes dans $S^{\delta} \oplus HG$

Exemple 5.5 *Les entrées inopportunes de la composition* $S^{\delta} \oplus HG$ *de la* FIG.*34.(b) sont :*
– $ref(q_0) = \{?b\}$,
– $ref(q_1) = \{?a\}$,
– $ref(q_2) = \{?a, ?b\}$,
– $ref(q_3) = \{?a, ?b\}$,
– $ref(d_1) = \{?a', ?b, ?b'\}$,

– $ref(d_2) = \{?a', ?b\}$.

On considère que le comportement acceptable du système est donné par le meta-graphe IIG de la FIG.*34.(a), la composition de* $S^\delta \oplus HG$ *(*FIG.*34.(b)) et IIG est obtenue par l'application des règles 1, 2 et 6 de la composition (*FIG.*34.(c)).*

Vérification de traces de suspension et déterminisation

L'intégration d'entrées invalides, entrées inopportunes et sorties acceptables dans l'automate suspendu associé à la spécification peut modifier les informations de blocages. En effet, les états dégradés peuvent être des états de blocage (dans la FIG.34.(c), les états dégradés d_1 et d_2 sont deux états de blocage de sortie (outputlock)). En revanche, l'ajout de sorties acceptables dans les états de blocage de la spécification nominale ne supprime pas les traces de suspension déjà ajoutées, puisque celles-ci sont des comportements corrects de la spécification nominale (dans l'exemple q_0 et q_2).

La déterminisation du modèle consiste à résoudre les conflits liés à l'identification d'états et l'observation d'actions. La déterminisation permet de supprimer les situations dans lesquelles, à partir d'un état, il existe plus d'une action possible après une suite d'actions non observables (internes). En conséquence, il est indispensable de fournir un ensemble de traces caractérisées par un IOLTS déterministe.

Définition 24 (Déterminisation d'un $IOLTS$**)** *Soit* $S = (Q^S, q_0^S, \Sigma^S, \rightarrow_S)$*, un IOLTS. L'IOLTS déterministe résultant de S est noté* $\Delta(S)$ *tel que* $Traces(S) = Traces(\Delta(S))$*.* $\Delta(S) = (Q^{S^\Delta}, q_0^S\ after\ \varepsilon, \Sigma^{\Delta(S)}, \rightarrow_{\Delta(S)})$ *est défini par :*

– $Q^{S^\Delta} \subseteq 2^{Q^S}$*, c'est à dire les états de* $\Delta(S)$ *sont des parties de* Q^S*,*

– *L'état initial de* $\Delta(S)$ *est l'ensemble d'états de S accessibles depuis* q_0^S *par des actions internes. Formellement,* $q_0^{\Delta(S)} = \{q_0^S\ after\ \varepsilon\}$*,*

– $\Sigma^{\Delta(S)} = \Sigma^S$ *est l'ensemble d'actions observables,*

– $P \xrightarrow{a}_{\Delta(S)} P' \iff P, P' \in 2^{Q^S},\ a \in \Sigma^{\Delta(S)}$ *et* $P' = P\ after\ a$*.*

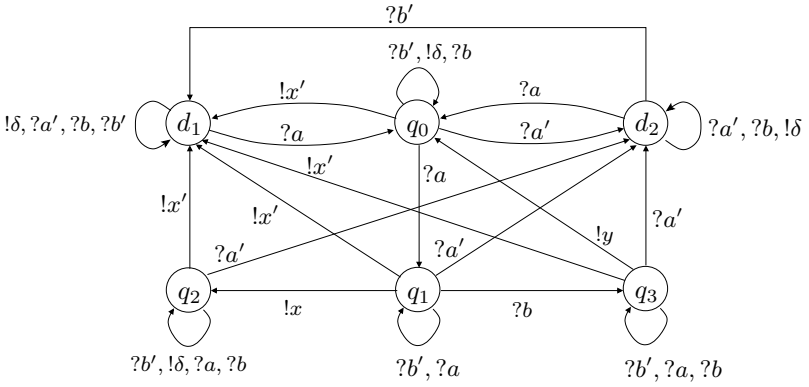


FIG. 35: La spécification augmentée S_A

La spécification augmentée (FIG.35) est donc un IOLTS déterministe, observable et input-complet (voir les notations standards sur les $IOLTSs$ dans le paragraphe 2.3.2). S_A décrit le comportement correct du système dans les conditions normales, ainsi que son comportement acceptable en présence d'entrées invalides, inopportunes et sorties acceptables.

5.6 Méthode TRACON

Naturellement, tester la robustesse en présence d'aléas contrôlables et non représentables consiste à exécuter une implémentation d'un système (IUT) dans son environnement réel, activer les aléas, appliquer des séquences d'entrées et comparer les sorties de l'IUT à celles de la référence. Les séquences d'entrées sont composées uniquement d'entrées nominales, c'est à dire les entrées décrites dans la spécification nominale.

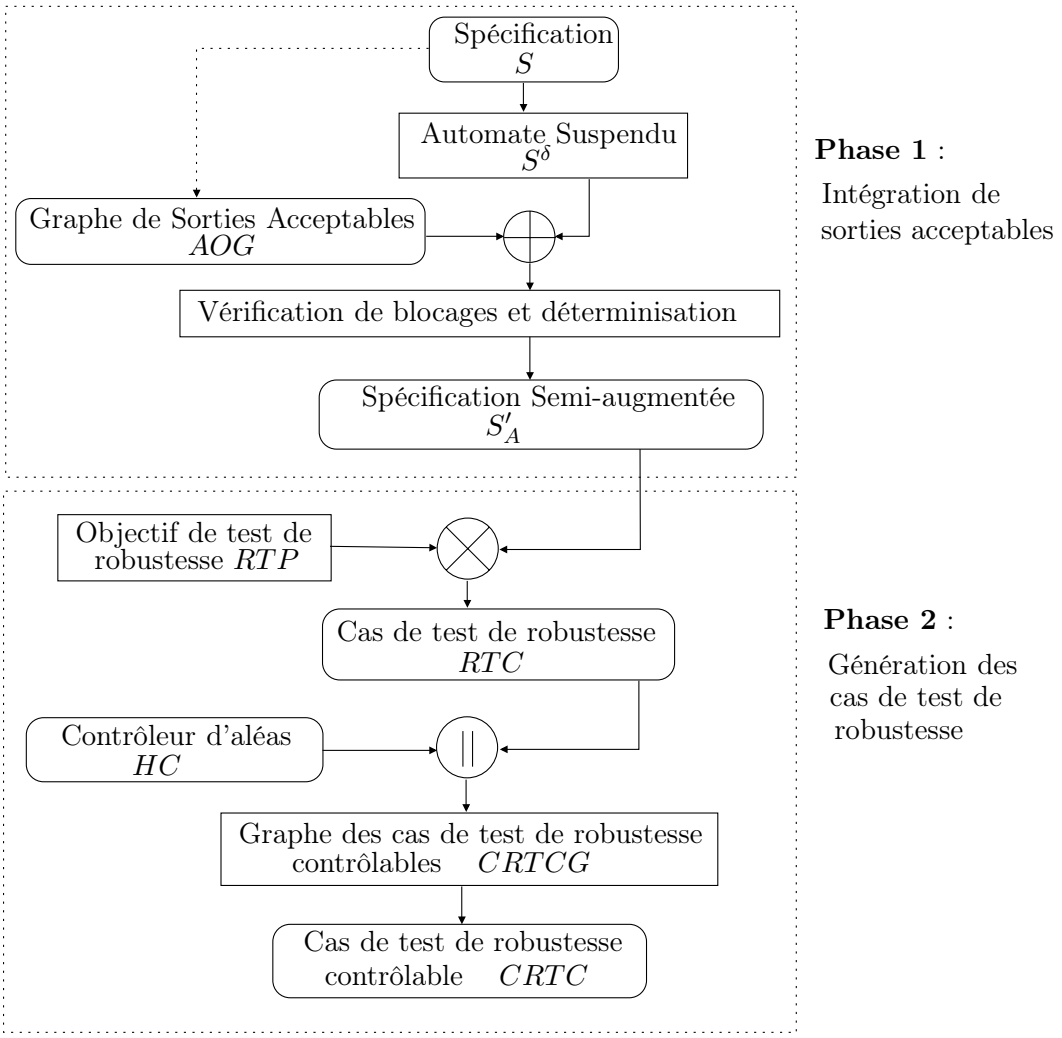


FIG. 36: Plateforme de la Méthode TRACON

Contrairement à la méthode TRACOR, on ne procède à aucune augmentation au niveau des entrées et, seules les sorties acceptables, qui peuvent être observées en présence d'aléas, sont prises en compte. Par conséquent, le domaine de sortie ainsi que la référence du test doivent être modifiés. De plus, les alternances démarrage/arrêt d'un aléa et l'application d'entrées est un aspect important qui doit être pris en compte durant la génération et l'exécution des tests de robustesse.

Afin de répondre à ces besoins, nous proposons la méthode TRACON dont l'architecture est présentée dans le paragraphe 5.4.4. Cette méthode complète la méthode TRACOR. Elle propose un cadre formel pour le test de robustesse en présence d'aléas contrôlables et non représentables (conditions de stress). Cette méthode est résumée en deux phases (voir la Fig.36) :

- **Phase 1** : Intégration de sorties acceptables,
- **Phase 2** : Génération des cas de test de robustesse.

5.6.1 Intégration de sorties acceptables

Dans la première étape de cette phase, on modélise les blocages valides dans l'IOLTS de la spécification nominale. Pour ce faire, nous utilisons l'automate suspendu donné par la définition 21. Parallèlement, les concepteurs du système doivent préciser l'ensemble de sorties acceptables, il s'agit ici de quelques sorties additionnelles, déjà spécifiées ou non (messages déjà décrits ou omis de la spécification nominale), par exemple, un message réinitialisant une connexion ou message exprimant une défaillance. Le comportement fourni par les concepteurs est supposé modélisable par un meta-graphe appelé *graphe de sorties acceptables* ou *AOG* (en anglais, *acceptable output graph* (Fig.37.(b)).

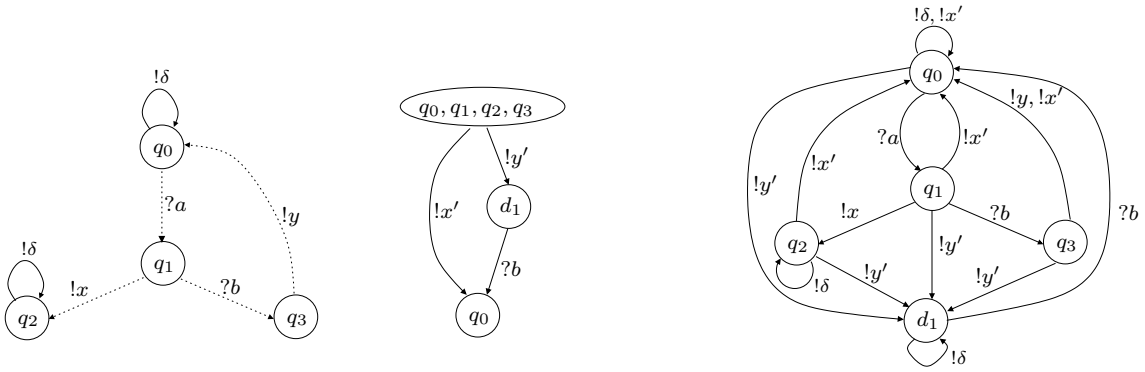
Définition 25 (Graphe de sorties acceptables) *Le graphe de sorties acceptables est un meta-graphe* $AOG = (V, E, L)$. *tel que :*

- $V = V_m \cup V_d$. *L'ensemble d'états contient des meta-états (un ensemble d'états nominaux)* V_m, *et un ensemble d'états dégradés* V_d.
- $L = L_I \cup L_O$. *Son alphabet contient les sorties qui peuvent être considérées comme acceptables en présence d'aléas (*L_O*), et dans certains cas, des entrées nominales pour assurer le retour dans un état de fonctionnement normal (*$L_I \subseteq \Sigma^S$*).*
- $E \subseteq V \times L \times V$ *un ensemble de transitions.*

La composition de l'automate suspendu, associé à la spécification nominale, S^δ et le graphe de sorties acceptables AOG, est calculée suivant les règles de la composition données dans la définition 22.

Vérification de traces de suspension et déterminisation

Après l'intégration de sorties acceptables, nous procédons à la vérification des traces de suspension, puis à la déterminisation de $S^{\delta} \oplus AOG$. Le modèle obtenu est appelé "*la spécification semi-augmentée*" (voir FIG.37.(c)). Elle servira comme une référence pour dériver les cas de test de robustesse en présence d'aléas contrôlables et non représentables.



(*a*). S^{δ} (*b*). *AOG* (*c*). Spécification semi-augmentée S'_A

FIG. 37: L'intégration de sorties acceptables dans la méthode TRACON

Exemple 5.6 *La spécification semi-augmentée de la* FIG.*.37.(c) est obtenue comme suit :*

1. *Construction de l'automate suspendu associé la spécification nominale S (S est dessiné par des traits discontinus dans la* FIG.*37.(a)),*
2. *La composition de l'automate suspendu S^{δ} (*FIG.*37.(a)) et le graphe de sorties acceptables AOG (*FIG.*37.(b)),*
3. *La mise à jour de traces de suspension dans l'état dégradé d_1 (d_1 est un état de blocage complet (deadlock)), puis la déterminisation du modèle obtenu.*

La **Phase 2** (génération des cas de test de robustesse) consiste à dériver de la spécification semi-augmentée, les traces satisfaisant un besoin spécifique appelé "objectif de test de robustesse". Cette phase se termine par le calcul d'un cas de test de robustesse RTC (*robustness test case*) décrivant seulement les interactions entre le testeur de robustesse et l'IUT. Dans la méthode TRACON, un cas de test de robustesse doit décrire aussi les interactions entre le testeur et le contrôleur d'aléas (pour activer ou désactiver les aléas pendant l'exécution du test). En conséquence, nous devons élargir le cas de test de robustesse RTC par l'ajout de commandes de contrôle (*start/stop*) de HC. Le résultat de cette opération est le *graphe des cas de test de robustesse contrôlables* CRTCG (en anglais, *controllable robustness test cases graph*). Cette phase sera détaillée dans le chapitre 6.

5.7 Relation de robustesse

Dans cette section, nous formalisons, à travers une relation binaire, la notion de la robustesse entre une implémentation sous test (IUT) et la spécification augmentée (ou la spécification semi-augmentée). Nous rappelons que dans l'approche du test boîte noire, le code de l'IUT est inconnu. Mais pour pouvoir raisonner formellement sur la robustesse d'une implémentation vis-à-vis de la spécification augmentée (ou la spécification semi-augmentée), nous considérons les hypothèses du test suivantes :

1. Les comportements possibles de l'IUT sont modélisables par un IOLTS noté $IUT = (Q^{IUT}, q_0^{IUT}, \Sigma^{IUT}, \rightarrow_{IUT})$ tel que $\Sigma_I^{S_A} \subseteq \Sigma_I^{IUT}$ et $\Sigma_O^{S_A} \subseteq \Sigma_O^{IUT}$. Ceci reflète le fait que l'alphabet de l'IUT est inconnu, mais qu'à priori ses entrées contiennent celles de la spécification augmentée.
2. L'implémentation est supposée complète en entrées sur l'alphabet d'entrée augmentée ($\Sigma_I^{S_A}$), c'est à dire dans chaque état, l'IUT ne peut refuser aucune entrée émise par le testeur. Formellement, $\forall q \in Q^{IUT}, \forall a \in \Sigma_I^{S_A}, q \xrightarrow{a}$. Cette hypothèse est nécessaire pour garantir que chaque blocage de l'interaction $Testeur/IUT$ produit un verdict. Notons que cette hypothèse n'implique pas forcement la robustesse de l'IUT puisqu'il est possible que l'IUT répond négativement en cas de réception d'une entrée inopportune ou invalide, et justement nous cherchons à comparer ses réponses (sorties) à celles de la spécification augmentée.
3. L'IUT est conforme à la spécification nominale S. Dans ce but, nous utilisons la relation de conformité **ioco** de *Tretmans* [190, 191] pour s'assurer de la conformité de l'IUT vis-à-vis de S (voir paragraphe 3.2.3). Cette relation considère qu'une IUT est conforme à S si après chaque trace de S, les sorties et les blocages de IUT sont inclus dans ceux de S. Ceci est donné par la relation :

$$IUT \textbf{ ioco } S \equiv_{def} \forall \sigma \in Traces(S^{\delta}) \implies Out(IUT^{\delta}, \sigma) \subseteq Out(S^{\delta}, \sigma).$$

Soit S une spécification nominale, la spécification augmentée obtenue à partir de S est un IOLTS $S_A = (Q^{S_A}, q_0^{S_A}, \Sigma^{S_A}, \rightarrow_{S_A})$ tel que :

- $Q^{S_A} = Q^{S_A} \cup D$ l'ensemble d'état est composé de l'ensemble d'états de la spécification nominale S et un ensemble d'états dégradés D. $q_0^{S_A} = q_0^S$ l'état initial de S_A est l'état initial de S ;
- $\Sigma^{S_A} = \Sigma^S \cup \theta$ l'alphabet de la spécification augmentée est composé de l'alphabet de la spécification nominale ainsi qu'un ensemble d'aléas (entrées invalides et sorties inattendue), cet ensemble est noté par θ,
- $\rightarrow_{S_A} = \rightarrow_S \cup \rightarrow_\theta$ l'ensemble de transitions augmentées. Cet ensemble est composé de l'ensemble de transitions de S et un ensemble de transitions décrivant les aléas.

La relation ***ioco*** peut être élargie pour tester la robustesse d'une implémentation IUT vis à vis de la spécification augmentée S_A. La relation de robustesse ***Robust*** est définie par :

$$IUT \quad \textbf{\textit{Robust}} \quad S_A \equiv_{def} \forall \sigma \in Traces(S_A) \backslash Trace(S^\delta) \Longrightarrow Out(IUT^\delta, \sigma) \subseteq Out(S_A, \sigma).$$

Seuls les comportements augmentés sont visés par le test de robustesse car les comportements nominaux sont déjà vérifiés par le test de conformité.

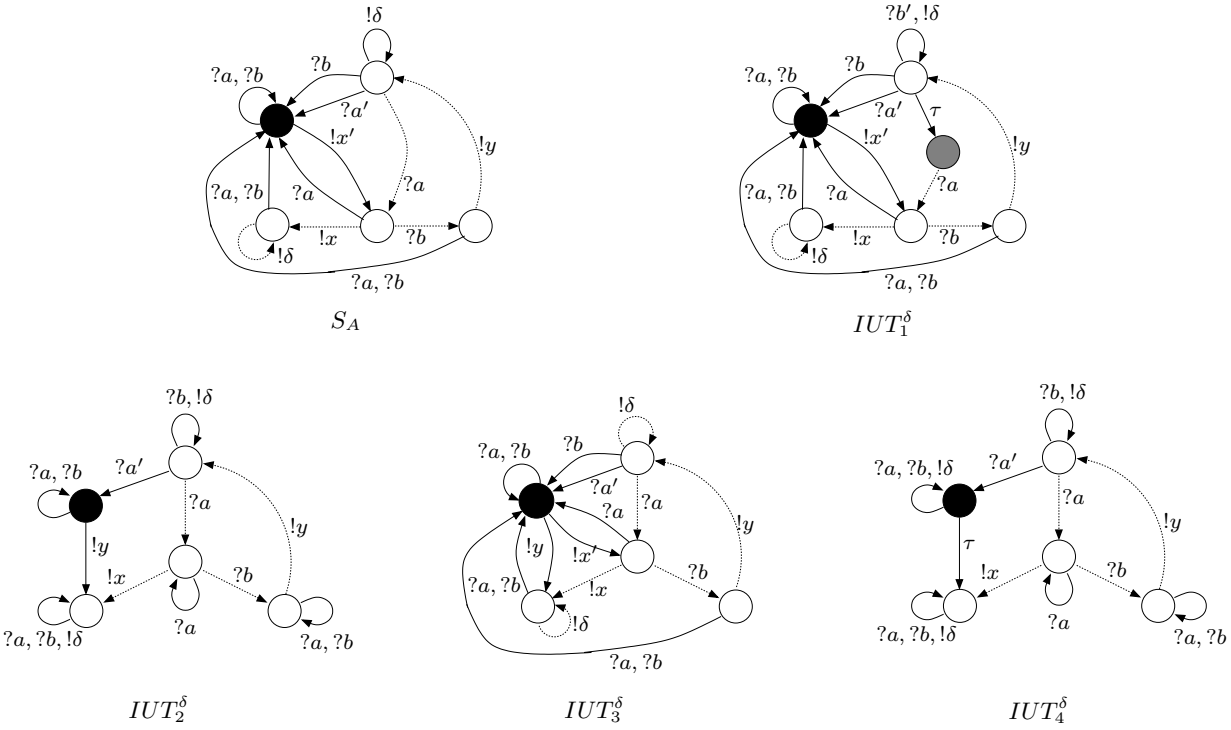


FIG. 38: Relation de robustesse

La relation "***Robust***" permet de détecter les implémentations non robustes malgré leur conformité à la spécification nominale. Dans la FIG.38 toutes les implémentations (IUT_1,...,IUT_4) sont conformes, selon la relation ***ioco***, à la spécification S (sur les exemples, elle est dessinée avec des traits discontinus). En revanche, seulement IUT_1 est robuste vis-à-vis de la spécification augmentée S_A. Dans cette implémentation (IUT_1), l'implémentateur a ajouté un état additionnel accessible à partir de l'état initial par une action interne τ. Ceci n'a pas d'impact sur l'observation de traces. De plus, il a ajouté une boucle dans l'état initial étiquetée par une entrée invalide $?b'$, cette boucle ne sera pas vérifiée par le test de robustesse car $?b' \notin \Sigma^{S_A}$.

Les types de fautes (ou modèle de fautes) qui peuvent être identifiés par la relation ***Robust*** sont les fautes de sorties et de blocage. Ci-dessous, nous donnons quelques exemples :

Sortie manquante

C'est le type de fautes le plus attendu. En effet, si l'implémentateur s'appuie seulement sur la spécification nominale, qu'elle est probablement incomplète, pour développer l'implémentation, alors il se peut que celle-ci se comporte négativement en présence d'aléas. Dans la Fig.38, si on applique la trace $?a, !x, ?b$ sur l'IUT_2 on obtient la sortie $!\delta$. En revanche, l'application de la même trace dans S_A donne la sortie $!x'$. En conséquence, IUT_2 **not** ***Robust*** S_A .

Sortie additionnelle

En plus des sorties manquantes, ***Robust*** permet de détecter les sorties qui ne sont pas autorisées dans un état donné de la spécification augmentée. Dans l'exemple de la Fig.38, IUT_3 **not** ***Robust*** S_A car $IUT_3\ after\ ?a = \{!x', !y\}$. En revanche, $S_A\ after\ ?a = \{!x'\}$. $!y$ est une sortie additionnelle.

Blocage additionnel

Ce type de faute est un cas particulier du modèle de fautes "sortie manquante". En fait, l'oubli de toutes les sorties dans un état donné transforme cet état en état de blocage. Dans l'exemple de la Fig.38, IUT_4 **not** ***Robust*** S_A car $IUT_4\ after\ ?a = \{!\delta\}$. En revanche, $S_A\ after\ ?a = \{!x'\}$. L'état $IUT_4\ after\ ?a$ est un état de blocage.

5.8 Conclusion

Nous venons de présenter un cadre formel pour le test de robustesse fondé sur une définition spécifique de la robustesse et une classification formelle d'aléas. Ce cadre est composé d'une approche et une relation de robustesse.

Notre définition considère le test de robustesse comme un sur-ensemble du test de conformité. En effet, les systèmes visés par le test de robustesse sont préalablement conformes à leurs spécifications nominales.

Les aléas désignent tout événement omis de la spécification nominale du système. Ils peuvent être classifiés selon plusieurs points de vue : la situation vis-à-vis des frontières du système, leur observabilité par le testeur, contrôlabilité par le testeur et représentabilité dans un formalisme de description.

Notre approche est dédiée aux protocoles de communication. Dans ce domaine, les influences d'aléas sur les entrées/sorties d'une entité protocolaire sont modélisables à travers

des entrées invalides, entrées inopportunes et sorties acceptables. A ce stade, nous avons proposé deux méthodes, l'une complète l'autre, pour le test de robustesse.

La **Méthode TRACOR** est fondée sur l'intégration d'entrées invalides, entrées inopportunes, sorties inattendue et traces de suspension dans le modèle de la spécification nominale. Le modèle obtenu, après la déterminisation, est appelé la spécification augmentée. Elle servira comme un modèle de référence pour la génération et l'exécution des tests de robustesse.

La **Méthode TRACON** est focalisée sur le test de robustesse en présence d'aléas contrôlables et non représentables dont l'influence sur les entrées/sorties d'une entité protocolaire est non représentable, voire imprévisible. Cette méthode intègre les sorties acceptables dans le modèle de la spécification nominale. Le nouveau modèle, obtenu après la déterminisation, est appelé la spécification semi-augmentée. Elle servira comme un modèle de référence pour la génération et l'exécution des tests de robustesse en présence de cette classe d'aléas.

Afin de formaliser la robustesse d'une implémentation sous test vis-à-vis de la spécification augmentée (ou la spécification semi augmentée), nous avons proposé la relation binaire ***Robust***. Elle complète la relation de conformité ***ioco*** de *Tretmans* [190, 191], mais elle est basée sur l'observation des blocages et sorties d'une implémentation sous test après l'exécution des traces contenant les aléas.

La *phase 2* de chaque méthode est dédiée à la génération des cas de test de robustesse à l'aide d'objectifs de test de robustesse. Cette phase sera détaillée dans le chapitre suivant.

Chapitre 6

Génération des tests de robustesse

Nous avons exposé dans le chapitre 3 un état de l'art concernant les méthodes de génération des tests de conformité. Ceci nous a permis d'explorer les principes, les avantages et les inconvénients de chaque méthode afin d'en dégager des pistes servant pour développer une méthode dédiée à la génération des tests de robustesse.

Avant tout, on écarte les méthodes basées sur le modèle $EFSM$ et les graphes de flux de données, présentées dans la section 3.2.2, car tout simplement nous avons choisi d'utiliser le modèle IOLTS. Celui-ci n'autorise pas la représentation de flux de données. En parallèle, l'application des méthodes basées sur les $FSMs$, présentées dans la section 3.2.1, pour tester des systèmes complexes modélisés par des machines de taille importante est très coûteuse. De plus, l'application de ces méthodes nécessite un ensemble d'hypothèses portant d'une part sur la spécification mais surtout sur l'implémentation (complétude, déterminisme, forte connexité, etc). Ceci est moins réaliste dans le domaine des protocoles.

Dans notre approche pour le test de robustesse, l'intégration d'aléas augmente la taille du modèle de la spécification et par conséquent, le processus de la génération des tests de robustesse devient de plus en plus coûteux. Afin de réduire les efforts, nous avons besoin de nous concentrer sur le test de certaines propriétés ou fonctionnalités précises. Ce besoin nous a poussé vers les méthodes utilisant des *objectifs de test* (voir paragraphe 3.2.3). Parmi plusieurs, nous avons décidé de prendre comme point de départ les travaux fondés en collaboration entre IRISA et Verimag [63, 109, 107].

Ce chapitre est organisé comme suit : La section 6.1 définit la notion d'objectif du test de robustesse. La section 6.2 explique les différentes étapes de la méthode TRACOR permettant de générer les cas de test de robustesse. La section 6.3 détaille les étapes de génération des cas de test de robustesse dans la méthode TRACON, et enfin, la section 6.4 conclut le chapitre. Les résultats de ce chapitre sont publiés dans [164].

6.1 Objectifs de test de robustesse

Un objectif de test de robustesse RTP (en anglais, *robustness test purpose*) correspond à une suite d'entrées/sorties qui doivent figurer dans les séquences du test pour vérifier la propriété voulue. De plus, un RTP peut permettre d'élaguer certaines séquences de la spécification augmentée (ou semi-augmentée). Dans notre approche, un objectif de test de robustesse est défini comme suit :

Définition 26 (RTP) *Un objectif de test de robustesse RTP est un IOLTS déterministe équipé obligatoirement d'un état final "**Accept**" décrivant le comportement accepté par RTP et, optionnellement d'un autre état final "**Reject**" décrivant le comportement rejeté par RTP. Formellement,* $RTP = (Q^{RTP}, q_0^{RTP}, \Sigma^{RTP}, \rightarrow_{RTP})$ *tel que :*

- $Q^{RTP} \subset Q^{S_A} \cup \{\textbf{\textit{Accept}}\} \cup \{\textbf{\textit{Reject}}\}$.
- Σ^{RTP} *est l'alphabet de RTP, il consiste en un sous-ensemble d'actions visibles de la spécification augmentée* S_A*, c'est à dire* $\Sigma^{RTP} \subseteq \Sigma^{S_A}$*. De plus, l'alphabet de RTP peut être un sous ensemble de l'alphabet* Σ^S*, c'est à dire* $\Sigma^{RTP} \subseteq \Sigma^S$.

Dans le cadre du test de robustesse, un objectif de test a pour but de décrire certains aspects de fonctionnement en présence d'aléas. En particulier, un RTP peut décrire :

La conservation d'une fonctionnalité. Certaines fonctionnalités de la spécification nominale doivent être réalisables en présence d'aléas. Intuitivement, les $RTPs$ doivent permettre de trouver les traces de la spécification augmentée contenant les transitions nominales vérifiant la fonctionnalité voulue, mais aussi des transitions décrivant les aléas (voir RTP_4 de la FIG.39).

La réalisation d'une propriété de robustesse. Il s'agit de certaines propriétés exprimant un comportement particulier en présence d'aléas. Par exemple, l'observation d'une sortie acceptable après la réception d'une entrée invalide (voir RTP_1, RTP_2, RTP_3 de la FIG.39).

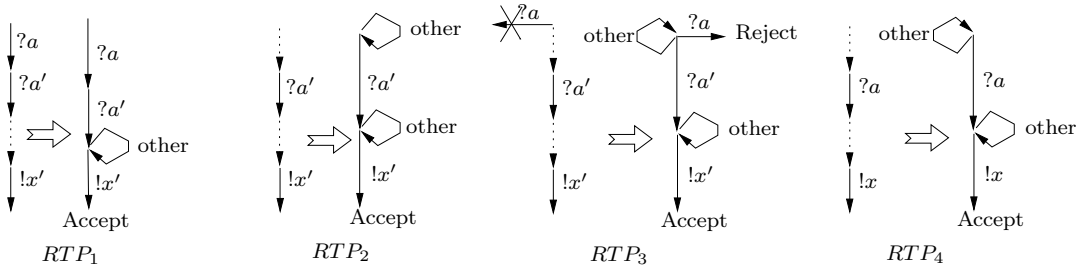


FIG. 39: Exemple d'objectifs de test de robustesse.

Exemple 6.1 *Dans la* FIG.*39 :*

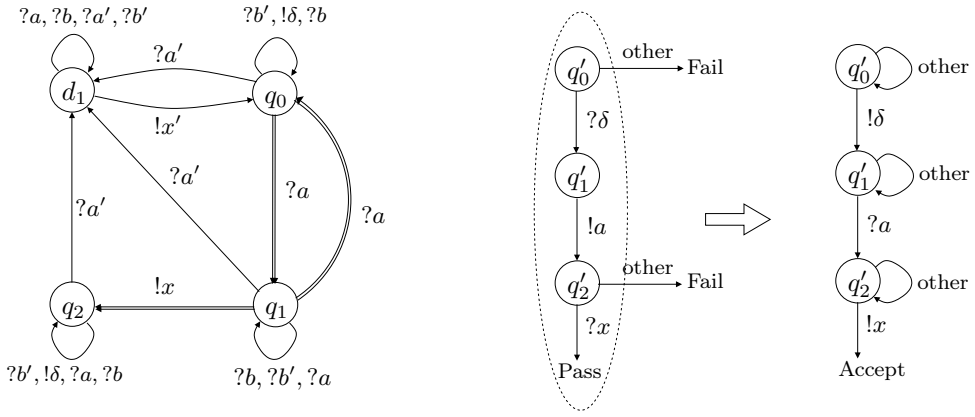
- RTP_1 *décrit toute trace de la spécification augmentée commençant par deux réceptions successives d'une entrée valide* $?a$ *et une entrée invalide* $?a'$*, suivies par une émission de la sortie acceptable* $!x'$.

- *RTP_2 décrit toute trace de la spécification augmentée passant par une réception de l'entrée invalide $?a'$ et une émission de la sortie acceptable $!x'$.*
- *RTP_3 décrit toute trace de la spécification augmentée passant par une réception de l'entrée invalide $?a'$ et une émission de la sortie acceptable $!x'$, sans aucun passage par des transitions etiquetées par $?a$.*
- *RTP_4 décrit la fonctionnalité nominale : émission de $!x$ après réception de $?a$.*

Remarque : On emploie l'étiquette "*other*" pour décrire toutes les actions de l'alphabet qui ne sont pas spécifiées dans l'état courant. Ceci implique l'acceptation de n'importe quelle action de l'alphabet hors les entrées/sorties spécifiées dans cet état, c'est à dire pour un état q, $other = \Sigma^{S_A} \setminus (Out(q) \cup In(q))$.

6.1.1 Transformation des cas de test de conformité en objectifs de test de robustesse

Les objectifs de test de robustesse, visant à réaliser une fonctionnalité précise, peuvent être calculés directement à partir d'un cas de test de conformité (voir Fig.40.*b*) obtenu à partir de la spécification nominale (elle est dessinée par des traits doubles dans la Fig.40.*a*). Pour obtenir un objectif de test de robustesse, à partir d'un cas de test de conformité, on



(*a*). S_A (*b*). Cas de test de conformité (*c*). RTP

Fig. 40: Un RTP calculé à partir d'un cas de test de conformité.

procède aux étapes suivantes :

1. Supprimer les verdicts de la conformité,
2. Inverser, par image miroir, le cas de de test de conformité (les entrées deviennent entrées et inversement),
3. Ajouter des boucles, dans chaque état, permettant de considérer les aléas. Pour ce faire, il suffit seulement d'ajouter des boucles étiquetées par *other* dans chaque état. *other* sera remplacée, par la suite, par l'ensemble d'aléas spécifiés dans cet état.

L'avantage de cette idée consiste à faciliter l'écriture d'objectifs de test de robustesse, mais aussi à donner la possibilité de tester, à la fois, la conformité et la robustesse d'une implémentation vis-à-vis de la même propriété. De plus, l'utilisation d'objectifs de test de robustesse calculés à partir d'une trace de spécification nominale permet de réduire l'effort de la génération des cas des test de robustesse.

6.2 Principes de génération dans la méthode TRACOR

Nous rappelons que le cadre formel que nous avons présenté dans le chapitre 5 propose deux méthodes pour le test de robustesse, la méthode TRACOR est focalisée sur les aléas contrôlables et représentables (entrées invalides, entrées inopportunes et sorties acceptables) et, la méthode TRACON sur les aléas contrôlables non représentables. La **Phase 2** de chaque méthode intègre une technique pour la génération des cas de test de robustesse respectant un objectif de test RTP. Cette phase peut s'organiser selon les étapes suivantes (FIG.41) :

1. Choix d'un objectif de test de robustesse RTP,
2. Synchronisation de RTP avec la spécification augmentée S_A afin d'extraire les comportements de S_A satisfaisant RTP,
3. Inverser le produit synchrone, par image miroir, pour obtenir le graphe du test de robustesse RTG. Ensuite, supprimer les traces rejetées par RTP pour obtenir le graphe réduit du test de robustesse RRTG,
4. Sélection des cas de test de robustesse.

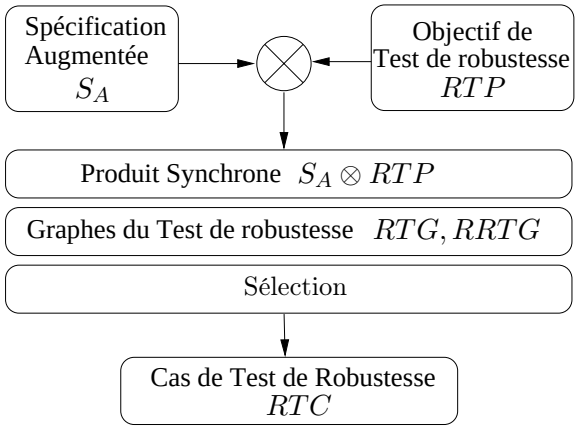


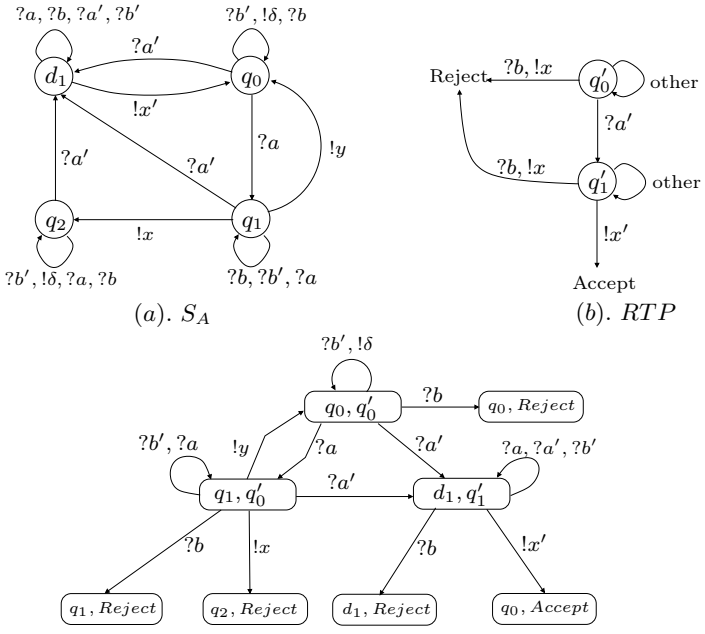
FIG. 41: Organigramme de génération des cas de test de robustesse

6.2.1 Produit synchrone

La démarche pour obtenir les traces de la spécification augmentée S_A, satisfaisant un objectif de test de robustesse RTP, consiste à parcourir en parallèle le RTP et la spécification S_A. Le produit synchrone est défini comme suit :

Définition 27 (Produit synchrone) *Soient $S_A = (Q^{S_A}, q_0^{S_A}, \Sigma^{S_A}, \rightarrow_{S_A})$ une spécification augmentée et $RTP = (Q^{RTP}, q_0^{RTP}, \Sigma^{RTP}, \rightarrow_{RTP})$ un objectif de test de robustesse. Le produit synchrone de S_A et RTP, noté $S_A \otimes RTP$, est un $IOLTS$ déterministe $S_A \otimes RTP = (Q^{S_A \otimes RTP}, q_0^{S_A \otimes RTP}, \Sigma^{S_A \otimes RTP}, \rightarrow_{S_A \otimes RTP})$ défini par :*

1. $q_0^{S_A \otimes RTP} = (q_0^{S_A}, q_0^{RTP})$,
2. $Q^{S_A \otimes RTP} = \{(q_1, q_2) \mid q_1 \in Q^{S_A}, q_2 \in Q^{RTP}\}$,
3. $\Sigma^{S_A \otimes RTP} \subseteq \Sigma^{S_A} \cup \Sigma^{RTP} = \Sigma^{S_A}$ *(car nous avons supposé que $\Sigma^{RTP} \subseteq \Sigma^{S_A}$ dans la définition 26)*,
4. $\rightarrow_{S_A \otimes RTP}$ *est définie par :* $(q, q') \in Q^{S_A \otimes RTP}$, $q \xrightarrow{a}_{S_A} q_1 \wedge\ q' \xrightarrow{a}_{RTP} q'_1$ $\iff (q, q') \xrightarrow{a}_{S_A \otimes RTP} (q_1, q'_1)$. *Une transition de $(q, q') \xrightarrow{a}_{S_A \otimes RTP} (q_1, q'_1)$ est obtenue s'ils existent deux transitions : $q \xrightarrow{a}_{S_A} q_1$ et $q' \xrightarrow{a}_{RTP} q'_1$.*



(*a*). S_A (*b*). RTP

(*c*). $S_A \otimes RTP$

Fig. 42: Le produit synchrone.

Exemple 6.2 *Le produit synchrone de la* Fig.*42.c est obtenu à partir de la spécification augmentée S_A (*Fig.*42.a) et l'objectif de test de robustesse RTP (*Fig.*42.b) qui vise à chercher toute trace, de la spécification augmentée, contenant une réception de l'entrée*

invalide $?a'$ *suivie par une émission de la sortie acceptable* $!x'$*, sans aucun passage par des transitions étiquetées par* $!x$ *ou* $?b$*.*

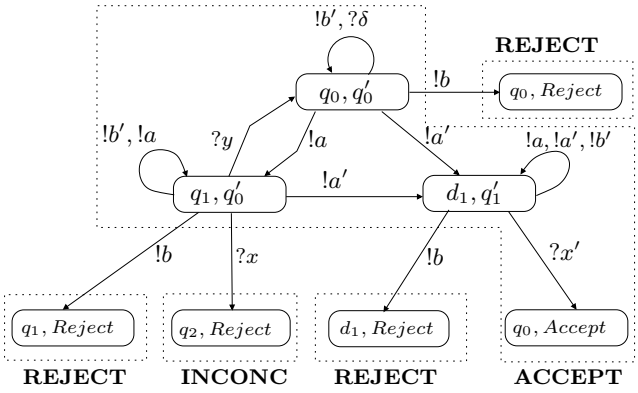
6.2.2 Graphes du test de robustesse

Le graphe du test de robustesse RTG (en anglais, *robustness test graph*) décrit l'ensemble des tests correspondant à un objectif de test de robustesse RTP. Formellement, RTG est un IOLTS déterministe $RTG = (Q^{RTG}, q_0^{RTG}, \Sigma^{RTG}, \rightarrow_{RTG})$ tel que :

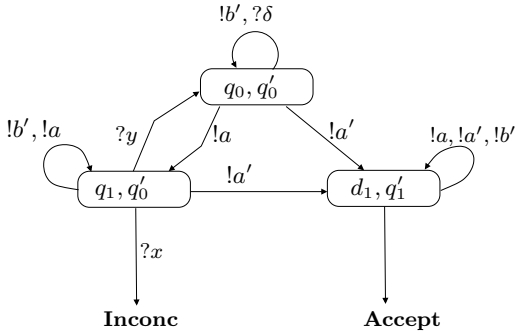
- $\Sigma^{RTG} = \Sigma_O^{RTG} \cup \Sigma_I^{RTG}$ avec $\Sigma_O^{RTG} = \Sigma_I^{S_A \otimes RTP}$ et $\Sigma_I^{RTG} = \Sigma_O^{S_A \otimes RTP}$, c'est à dire l'alphabet de RTG est l'image miroir de l'alphabet de $S_A \otimes RTP$ (les entrées deviennent sorties et inversement). L'image miroir décrit le comportement du testeur. Ceci implique que chaque réception dans $S_A \otimes RTP$ devrait être une émission dans RTG.
- $Q^{RTG} =$ **ACCEPT** $\cup$ **INCONC** $\cup$ **REJECT**. Q^{RTG} est composé de trois sous ensembles d'états :
  1. **ACCEPT** est l'ensemble d'états de $S_A \otimes RTP$ depuis lesquels l'état **Accept** est accessible. Formellement :
     ACCEPT $= \{q \in Q^{S_A \otimes RTP} | \ \exists \sigma \in \Sigma^{S_A \otimes RTP *}, \ q \xrightarrow{\sigma}_{S_A \otimes RTP}$ **Accept**$\}$.
  2. **INCONC** est l'ensemble d'états de $S_A \otimes RTP$ n'appartenant pas à l'ensemble **ACCEPT**, mais qui sont successeurs immédiats par une émission d'un état de l'ensemble **ACCEPT**. Formellement :
     INCONC $= \{q' \in Q^{S_A \otimes RTP} \mid \exists q' \notin$ **ACCEPT**, $\exists q \in$ **ACCEPT**, $a \in \Sigma_O^{S_A \otimes RTP} \mid q \xrightarrow{a}_{S_A \otimes RTP} q'\}$.
  3. **REJECT** est l'ensemble d'états qui n'appartient ni à **ACCEPT**, ni à **INCONC**.
- Si l'état initial de $S_A \otimes RTP$ est dans l'ensemble **ACCEPT**, c'est aussi l'état initial de RTG, sinon RTG est vide.

Le graphe du test de robustesse est très intéressant puisqu'il représente tous les tests correspondant à un objectif de test. En revanche, le fait de calculer un trop grand graphe de test pourrait limiter l'intérêt pratique. Par conséquent, nous avons besoin de réduire ce graphe en se concentrant uniquement sur les comportements acceptés par l'objectif de test de robustesse. Le graphe obtenu est appelé le *graphe réduit du test de robustesse* et, noté RRTG (en anglais, *reduced robustness test graph*).

Dans le test, les sorties sont incontrôlables, c'est à dire le testeur ne peut pas empêcher l'implémentation d'émettre ses sorties. C'est pour cette raison que nous devrons les conserver dans le graphe réduit du test de robustesse. En conséquence, Le RRTG ne contient que l'ensemble d'états de RTG depuis lesquels l'état **Accept** est accessible, ainsi que toute réception partant de cette ensemble (on parle ici de réception car la réduction est basée sur le graphe du test de robustesse RTG qui est basé lui-même sur l'image miroir de $S_A \otimes RTP$).



(*d*). *RTG*



(*e*). *RRTG*

Fig. 43: Les graphes du test de robustesse

Le graphe réduit du test de robustesse (RRTG) est obtenu comme suit :

1. supprimer les états de l'ensemble **REJECT**, ainsi que toute transition dont l'état d'arrivée fait partie de **REJECT**,
2. remplacer les états de l'ensemble **INCONC** par un état sans successeur étiqueté par **Inconc**,
3. remplacer tout état $(q, \textbf{Accept})$ par un état sans successeur étiqueté par **Accept**,
4. renuméroter les autres états de **ACCEPT**.

Exemple 6.3 *Le graphe du test de robustesse RTG da la* Fig.*43.*(*a*) *présente l'image miroir du produit synchrone de la* Fig.*42.*(*c*). *RTG est composé de trois ensembles d'états :* $\boldsymbol{INCONC} = \{(q_2, \textbf{\textit{Reject}})\}$, $\boldsymbol{REJECT} = \{(d_1, \textbf{\textit{Reject}})$, $(q_1, \textbf{\textit{Reject}}$, $(q_0, \textbf{\textit{Reject}}\}$ *et* $\boldsymbol{ACCEPT} = \{(q_0, q'_0)$, (d_1, q'_1), (q_1, q'_0), $(q_0, \textbf{\textit{Accept}})\}$.

Le graphe réduit du test de robustesse (RRTG) est obtenu comme suit :

1. *supprimer les états* $(d_1, \textbf{\textit{Reject}})$, $(q_1, \textbf{\textit{Reject}})$, $(q_0, \textbf{\textit{Reject}})$ *ainsi que les transitions* $\{(q_0, q'_0) \xrightarrow{!b} (q_0, \textbf{\textit{Reject}})$, $(q_1, q'_0) \xrightarrow{!b} (q_1, \textbf{\textit{Reject}})\}$, $(d_1, q'_1) \xrightarrow{!b} (d_1, \textbf{\textit{Reject}})\}$,
2. *remplacer l'état* $(q_2, \textbf{\textit{Reject}})$ *par l'état* ***Inconc***,
3. *remplacer l'état* $(q_0, \textbf{\textit{Accept}})$ *par l'état* ***Accept***.

6.2.3 Séquences, cas de test de robustesse

Une séquence de test de robustesse est composée d'un ensemble de cas de test de robustesse (RTC, *Robustness Test Case*) permettant de vérifier la robustesse d'une implémentation vis à vis de la spécification augmentée.

Un cas de test de robustesse est un test élémentaire qui correspond à un objectif particulier. Il décrit les interactions entre un testeur et une implémentation. Il n'aura donc que

des actions observables. Formellement, Un cas de test de robustesse RTC est modélisé par un IOLTS déterministe $RTC = (Q^{RTC}, q_0^{RTC}, \Sigma^{RTC}, \rightarrow_{RTC})$ muni de trois sous-ensembles distincts d'états sans successeurs **Accept**, **Fail** et **Inconc** de Q^{RTC} caractérisant les verdicts. L'alphabet de RTC est $\Sigma^{RTC} = \Sigma_I^{RTC} \cup \Sigma_O^{RTC}$ avec $\Sigma_O^{RTC} \in \Sigma_I^{S_A}$. Ceci implique qu'un RTC n'émet que des entrées de la spécification augmentée, c'est à dire des entrés invalides, entrées inopportunes et entrées valides. De plus, $\Sigma_I^{RTC} = \Sigma_O^{IUT} \cup \{\delta\}$. Ceci implique qu'un RTC doit prévoir la réception de toute sortie et blocage de l'IUT. Un cas de test de robustesse possède les caractéristiques suivantes :

1. Les ensembles d'états **Accept**, **Fail** et **Inconc** ne sont accessibles, par une seule transition, que par des réceptions (entrées). Autrement dit, si le testeur reçoit une entrée, il émet alors un verdict,
2. A tout état, un verdict est accessible. Formellement, $\forall q, \in Q^{RTC} \ \exists \sigma \in \Sigma^{RTC^*}, \ \exists q' \in \{\mathbf{Accept}, \mathbf{Inconc}, \mathbf{Fail}\}$ tel que $q \xrightarrow{\sigma} q'$,
3. Dans tout état d'un RTC, il n'y a jamais un choix entre deux sorties ou entre sorties et entrées. Formellement, $\forall q \in Q^{RTC}$, $(\exists a \in \Sigma_O^{RTC}, \ q \xrightarrow{a}_{RTC} \implies \ \forall b \in \Sigma^{RTC}, \ b \neq a, \ q \not\xrightarrow{b}_{RTC})$.
4. Tout état de réception devrait être complet en entrée. Formellement, $\forall q \in Q^{RTC}$, $(\exists a \in \Sigma_I^{RTC}, \ q \xrightarrow{a}_{RTC} \implies \ \forall b \in \Sigma_I^{RTC}, \ q \xrightarrow{b}_{RTC})$. Ceci implique que dans un état de réception, RTC doit prévoir toutes les sorties de l'IUT.

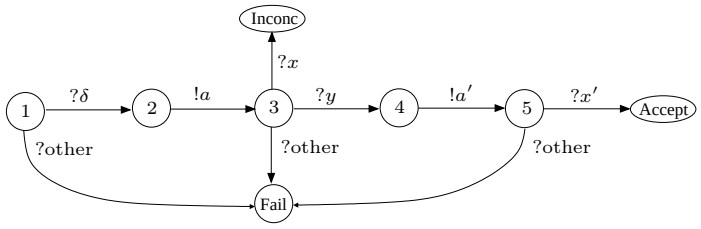
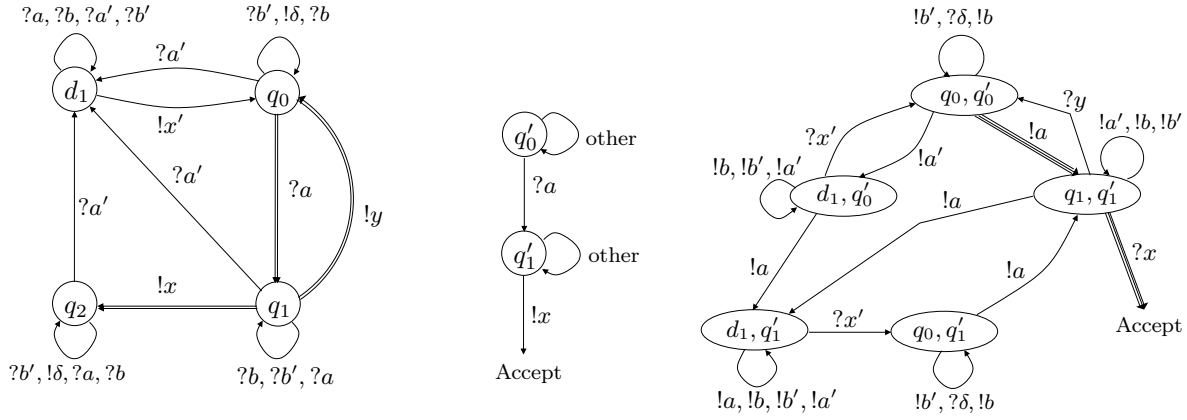


FIG. 44: Un exemple de cas de test de robustesse

6.2.4 Algorithme de sélection

Nous rappelons que la relation ***Robust***, définie dans la section 5.7, est fondée sur l'exclusion de traces de la spécification nominale. En conséquence, les cas de test de robustesse, supportés par cette relation, doivent viser les comportements assimilant les aléas , c'est à dire des traces n'appartenant pas à la spécification nominale.

La sélection aléatoire d'un RTC ne pose aucun problème, si on utilise des objectifs de test décrivant des propriétés de robustesse en présence d'entrées invalides, car toute trace, dérivée à partir du graphe réduit du test de robustesse RRTG, contient forcement les aléas décrits dans le RTP. De plus, les entrées invalides sont faciles à identifiées (hors de Σ^S).
En revanche, si on utilise des objectifs de test de robustesse visant à vérifier une propriété en présence d'entrées inopportunes, ou même des objectifs de test visant à conserver une



(*a*). S_A (*b*). *RTP* (*c*). *RRTG*

Fig. 45: Problématique de sélection des cas de test de robustesse

fonctionnalité nominale (voir Fig.45.*b*), le graphe réduit du test de robustesse peut contenir certaines traces entièrement spécifiées dans la spécification nominale (elle est dessinée en traits doubles dans la Fig.45.(*a*)). Par exemple, la trace $(q_0, q'_0) \xrightarrow{!a} (q_1, q'_1) \xrightarrow{?x} Accept$. De plus, les entrées inopportunes ne sont pas directement identifiables à partir de S_A car celles-ci appartiennent aussi à l'alphabet de la spécification nominale.

Pour résoudre les problèmes liés à la sélection des traces visées par le test de robustesse, nous proposons de faire un coloriage pendant la construction de la spécification augmentée (ou semi-augmentée). Dans ce but, nous utilisons deux couleurs pour distinguer les transitions de la spécification nominale et celles qui sont ajoutées pendant la phase 1 de chaque méthode. Un cas de test de robustesse peut être obtenu par l'algorithme 1.

Cet algorithme consiste à visiter les états du graphe réduit du test de robustesse RRTG à partir de l'état initial. Nous rappelons que RRTG est obtenu par la suppression de l'ensemble d'états **REJECT**. Dans chaque état visité, nous choisissons soit une émission soit toutes les réceptions et ainsi de suite jusqu'à l'arrivée à l'état **Accept**. Nous signalons qu'il existe deux types d'états dans le RRTG : états de réception ou états d'émission/réception. Les états d'émission sont impossibles car l'ajout de traces de suspension supprime cette possibilité. Enfin, tout RTC porte la même couleur que la spécification nominale sera rejeté. Pour connaître la couleur de RTC, on applique la règle suivante :

- Si toutes les transitions de RTC ont la même couleur que la spécification nominale alors RTC est de la même couleur de la spécification nominale,
- Sinon RTC est de couleur différente.

Algorithme 1 Sélection d'un RTC

ENTRÉES: Le graphe réduit du test de robustesse $RRTG$
SORTIES: Un cas de test de robustesse RTC

répéter
  $Q^{RTC} := q_0$;
  pour tout état q de Q^{RTC} n'est pas encore visité **faire**
    si q est un état de réception **alors**
      Ajouter toutes les transitions partant de q à RTC ;
      Ajouter tout les état d'arrivée à Q^{RTC} ;
      Ajouter une transition $q \xrightarrow{other} Fail$;
    finsi
    si q est un état d'émission/réception **alors**
      Choisir aléatoirement entre l'émission ou la réception ;
      si émission **alors**
        Choisir une émission aléatoire ;
        Ajouter l'état d'arrivée à Q^{RTC} ;
      sinon
        Ajouter toutes les réceptions partant de q à RTC ;
        Ajouter tout les état d'arrivée à Q^{RTC} ;
        Ajouter une transition $q \xrightarrow{other} Fail$;
      finsi
    finsi
  fin pour
jusqu'à la couleur de RTC est différente de celle de la spécification nominale.

6.2.5 Exécution de cas de test de robustesse

Dans cette méthode, les cas de test de robustesse décrivent un ensemble d'interactions à exécuter sur une implémentation sous test, certaines interactions sont des émissions d'entrées invalides ou inopportunes. Le testeur de robustesse émet alors les sorties d'un RTC (les entrées de l'implémentation) et, compare les observations (sorties de l'implémentation) à celles de RTC. Les verdicts sont les résultats de cette comparaison. Trois verdicts sont envisageables :

- ***Accept*** : traduit la robustesse du comportement de l'IUT.
- ***Inconc*** : le verdict *"Inconc"* ne traduit pas systématiquement la non robustesse de l'IUT. En effet, le comportement du testeur qui mène à cet état correspond à un comportement de la spécification S_A, mais qui ne satisfait pas l'objectif de test.
- ***Fail*** : si l'exécution d'un cas de test produit *"Fail"* alors le comportement de l'implémentation, observé par le testeur, n'est pas un comportement robuste vis-à-vis de la spécification augmentée S_A.

Exemple 6.4 *Le cas de test de robustesse donné dans la* FIG.*44 est généré à partir du graphe réduit du test de robustesse donné dans la* FIG.*43.(b). Ce RTC se compose de deux*

états d'émission (les états 2 et 4) et trois états de réception (les état 1, 3 et 5). L'exécution de ce RTC est détaillée ci-dessous :

- *Dans l'état initial (état 1), le testeur ne fait rien pendant une durée δ (blocage valide de l'IUT). En revanche, s'il reçoit n'importe quelle entrée (sortie de l'IUT), il émet alors un verdict **Fail** car aucune réception n'est autorisée dans cet état. Après l'expiration de la durée δ, le testeur se déplace à l'état 2.*
- *Dans l'état 2, le testeur émet l'entrée valide $!a$ et il se déplace à l'état 3.*
- *Dans l'état 3, si le testeur reçoit l'entrée $?x$, il émet alors le verdict **Inconc** car cette réception est bien spécifiée dans cet état, mais elle n'est pas visée par l'objectif de test. Si le testeur reçoit une autre entrée différente de $?x$, il émet alors le verdict **Fail**. En revanche, si le testeur reçoit l'entrée $?y$, il se déplace dans l'état 4.*
- *Dans l'état 4, le testeur émet l'entrée invalide $!a'$ et il se déplace à l'état 5.*
- *Dans l'état 5, si le testeur reçoit l'entrée $?x'$, il produit alors le verdict **Accept** et termine l'exécution du RTC. En revanche, s'il reçoit une autre entrée différente de $?x'$, il produit le verdict **Fail**.*

6.3 Principes de génération dans la méthode TRACON

Nous rappelons que la méthode TRACON est fondée sur l'exécution réelle d'une IUT en présence d'aléas contrôlables et non représentables. Les aléas considérés dans cette méthode sont supposés contrôlables ou commandables par un contrôleur d'aléas HC, c'est à dire une entité matérielle ou logicielle capable d'activer ou désactiver l'exécution d'un aléa physique (voir Fig.46.(b)). Du point de vue formel, nous avons donc besoin de construire des cas de test de robustesse décrivant à la fois les intéractions testeur/IUT et testeur/HC.

6.3.1 Génération d'un cas de tes de robustesse non contrôlable

Partant de la spécification semi-augmentée S'_A et, d'un objectif de test de robustesse RTP, on applique les mêmes principes de génération de la méthode TRACOR jusqu'à l'obtention d'un cas de test de robustesse RTC. Nous rappelons qu'un RTC décrit seulement les intéractions entre le testeur de robustesse et une IUT. En conséquence, nous avons besoin d'ajouter les intéractions testeur/HC dans un cas de test ordinaire afin qu'il soit contrôlable par le testeur.

6.3.2 Composition parallèle de RTC avec HC

Un contrôleur d'aléa HC est considéré comme une boîte noire (on ne s'intéresse pas à sa structure). Les intéractions entre le testeur et HC peuvent être modélisées par un IOLTS (voir Fig.46.(b)).

Cette étape consiste à composer parallèlement le cas de test de robustesse RTC avec le contrôleur d'aléas HC. La composition parallèle, notée par $\|$, est définie par :

Définition 28 (Composition parallèle) *Soient* $S_1 = (Q^{S'_A}, q_0^{S'_A}, \Sigma^{S'_A}, \rightarrow_{S'_A})$ *et* $HC = (Q^{HC}, q_0^{HC}, \Sigma^{HC}, \rightarrow_{HC})$ *deux IOLTS. La composition parallèle de* S'_A *et* HC*, notée* $S'_A \| HC$*, est un IOLTS* $S'_A \| HC = (Q^{S'_A \| HC}, q_0^{S'_A \| HC}, \Sigma^{S'_A \| HC}, \rightarrow_{S'_A \| HC})$ *défini par :*

1. $q_0^{S'_A \| HC} = (q_0^{S'_A}, q_0^{HC})$,
2. $Q^{S'_A \| HC} = \{(q_1, q_2) \mid q_1 \in Q^{S'_A}, q_2 \in Q^{HC}\}$,
3. $\Sigma^{S'_A \| HC} \subseteq \Sigma^{S'_A} \cup \Sigma^{HC}$,
4. $\rightarrow_{S'_A \| HC}$ *est obtenue par l'application de l'une des règles suivantes :*

   (a) $q_1 \xrightarrow{\mu}_{S'_A} q'_1$, $q_2 \xrightarrow{\mu}_{HC} q'_2$, $\mu \in \Sigma^{S'_A} \cap \Sigma^{HC} \implies (q_1, q_2) \xrightarrow{\mu}_{S'_A \| HC} (q'_1, q'_2)$

   (b) $q_1 \xrightarrow{\mu}_{S'_A} q'_1$, $\mu \in \Sigma_\tau^{S'_A}$, $\mu \notin \Sigma^{HC} \implies (q_1, q_2) \xrightarrow{\mu}_{S'_A \| HC} (q'_1, q_2)$

   (c) $q_2 \xrightarrow{\mu}_{HC} q'_2$, $\mu \in \Sigma_\tau^{HC}$, $\mu \notin \Sigma^{S'_A} \implies (q_1, q_2) \xrightarrow{\mu}_{S'_A \| HC} (q_1, q'_2)$.

Dans la première règle, S'_A et HC exécutent simultanément l'action de synchronisation. Les deux autres règles correspondent à une évolution indépendante de S'_A et de HC.

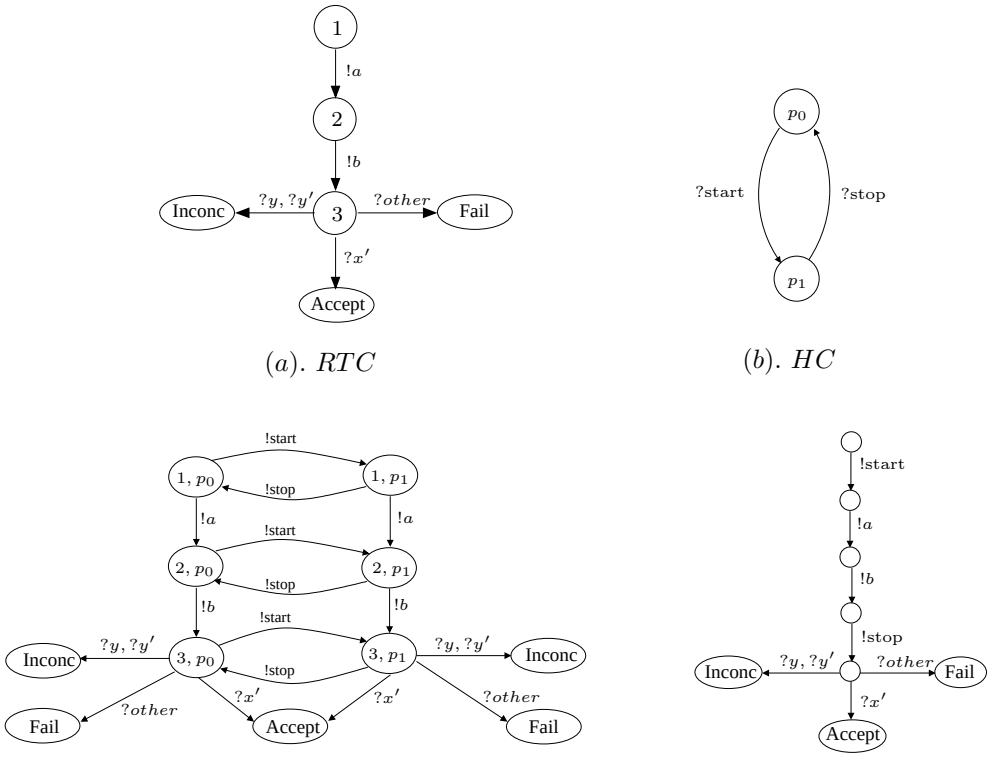
Signalons que la composition parallèle de S'_A et HC est obtenue par l'application des règles (a) et (b) de la définition 28 car $\Sigma^{S'_A} \cap \Sigma^{HC} = \emptyset$.

L'IOLTS, obtenu par la composition parallèle de S'_A et HC, est appelé le *graphe des cas de test de robustesse contrôlables* et noté par $CRTCG = S'_A \| HC$ (en anglais, *controllable robustness test cases graph*). CRTCG décrit toutes les intéractions possibles entre le testeur et le contrôleur d'aléas pour exécuter le cas de test RTC (voir FIG.46.(c)).

6.3.3 Sélection d'un cas de test de robustesse contrôlable

Cette étape consiste à extraire un cas de test de robustesse contrôlable CRTC (en anglais, *controllable robustness test case*) à partir de CRTCG. Dans ce but, on applique un algorithme de parcours en arrière partant de l'état **Accept** et on choisit un CRTC contenant au moins une transition étiquetée par !*start* (une transition !*stop* n'est accessible qu'après le passage par une transition !*start*). Les verdicts ne changent pas.

Exemple 6.5 *Pour exécuter le RTC de la* FIG.*46.(d), le testeur active d'abord l'aléa par l'émission de* !*start. Ensuite, il émet les sorties* !*a puis* !*b et, il désactive l'aléa par l'émission de* !*stop. Si, le testeur reçoit l'entrée* $?x'$ *(sortie de l'IUT), il produit le verdict* ***Accept****. S'il reçoit les entrées* $?y$ *ou* $?y'$*, il produit le verdict* ***Inconc****. En revanche, si le testeur reçoit une autre entrée différente de* $?x'$*,* $?y$ *ou* $?y'$ *il produit le verdict* ***Fail****.*



(a). *RTC* (b). *HC*

(c). *CRTG* (d). *CRTC*

Fig. 46: Dérivation d'un cas de test de robustesse de la méthode TRACON

6.4 Conclusion

Nous venons de décrire une méthode, fondée sur l'utilisation d'objectifs de test, pour la génération des cas de test de robustesse.

Dans la méthode TRACOR, le processus de la génération des tests de robustesse peut s'organiser selon les étapes suivantes : 1) Choix d'un objectif de test de robustesse. 2) Synchronisation de cet objectif avec la spécification augmentée afin d'en déduire les comportements qui satisfont cet objectif. 3) Inversion du produit synchrone, par image miroir afin d'obtenir le graphe du test de robustesse. 4) Réduction du graphe du test de robustesse. 5) Extraction des cas de test de robustesse. La définition d'un objectif de test de robustesse RTP doit tenir compte des aspects de fonctionnement en présence d'aléa.

Dans la méthode TRACON, en plus d'étapes précédentes (de la méthode TRACOR), on intègre les interactions entre le testeur et le contrôleur d'aléas, le résultat est le graphe des cas de test de robustesse contrôlables CRTCG. Enfin, on dérive, à partir de CRTCG, un cas de test de robustesse contrôlable.

Chapitre 7

Mise en œuvre et étude de cas

L'état de l'art exposé dans la section 3.3 projette la lumière sur un ensemble d'outils automatisant la génération des tests de conformité. Certains entre eux utilisent la notion d'objectif de test pour dériver les cas de test de conformité à partir d'une spécification formelle d'un système. Cette particularité aborde une interrogation fondamentale concernant l'éventuelle réutilisation de ces outils pour la génération des cas de test de robustesse. Á ce sujet, nous allons explorer quelques pistes permettant d'adapter ces outils à notre approche afin de réussir la génération des cas de test de robustesse. Ensuite, nous allons présenter un outil entièrement dédié à la génération des cas de test de robustesse. L'outil RTCG (*robustness test cases generator*) est basé sur les fondements théoriques présentés dans les chapitres 5 et 6. RTCG permet la génération des cas de test de robustesse selon les méthodes TRACOR et TRACON. Enfin, nous présentons une évaluation de notre outil à travers deux études de cas sur les protocoles SSL handshake et TCP. Une partie de cette étude de cas est publiée dans [165, 166, 163].

Ce chapitre est organisé comme suit : La section 7.1 explique comment adapter certains outils du test de conformité à notre approche. La section 7.2 présente les principaux algorithmes implantés dans l'outil RTCG. La section 7.3 propose une application de notre approche sur les protocoles SSL handshake et TCP. Enfin, la section 7.4 conclut le chapitre.

7.1 Utilisation d'outils du test de conformité

Notre approche et particulièrement la phase de génération de cas de test de robustesse est très proche de celle utilisée dans le prototype TGV [63, 109, 107] dont les fondements de base sont présentés dans les sections 3.2.3 et 3.3. Brièvement, cet outil applique une stratégie de génération à la volée (on-the-fly) pour dériver un cas de test de conformité à partir d'une spécification nominale et un objectif de test. Pour adapter cet outil à la méthode TRACOR, il est nécessaire de modifier les primitives suivantes :

Spécification. C'est plutôt la spécification augmentée S_A qui sera utilisée comme un

modèle de base pour le processus de génération. En revanche, cet outil n'offre aucun moyen pour automatiser l'augmentation de la spécification. Il faut donc voir comment intégrer un module externe permettant d'automatiser cette tâche.

Objectifs de test. En plus de la spécification, les objectifs de test présentent un handicap majeur pour la génération des cas de test de robustesse. En effet, l'utilisation d'objectifs visant à vérifier une fonctionnalité nominale en présence d'aléas ou, même visant à vérifier des propriétés en présence d'entrées inopportunes seulement (c'est à dire des objectifs dont l'alphabet est un sous ensemble de l'alphabet de la spécification nominale) risque de produire des cas de test de conformité et non pas de robustesse. Seuls les objectifs utilisant les entrées invalides (entrées hors l'alphabet de la spécification nominale) permettent de générer des cas de test de robustesse. En conséquence, pour adapter cet outil à notre approche, nous devons ajouter d'autres algorithmes de sélection plus adaptés à la robustesse. Par ailleurs, les outils existant n'offrent aucun moyen pour automatiser les différents besoins de la méthode TRACON.

7.2 Outil RTCG

Afin de mieux répondre aux différents besoins du test de robustesse présentés auparavant, nous avons développé l'outil RTCG (591 Ko). Cette application offre trois interfaces différentes : la première permet de construire une spécification augmentée à partir d'une spécification nominale, un graphe d'aléas et, optionnellement, un graphe d'entrées inopportunes. La deuxième interface est focalisée sur la génération automatique des cas de test de robustesse selon la méthode TRACOR, et la troisième permet d'automatiser la méthode TRACON.

Dans sa version actuelle, RTCG fonctionne uniquement dans l'environnement Windows©. Les spécifications sont formatés en SDL, les objectifs de test de robustesse sont écrits dans un langage formel très simple (DOT Language) supportant les différentes caractéristiques du modèle IOLTS, et les cas de test de robustesse sont formatés en TTCN-3 [58, 106], XML selon une DDT Calife et DOT. Enfin, RTCG permet, grâce à l'outil *GraphViz* [53], de représenter graphiquement les spécifications et les cas de test de robustesse en formats de sortie très variés.

7.2.1 Interface 1 : Augmentation de la spécification

Comme nous l'avons dit auparavant, cette interface automatise l'augmentation de la spécification nominale suivant les fondements théoriques de la méthode TRACOR. Les différentes étapes de cette interface sont détaillées dans la FIG.47. Les ovales décrivent les fichiers d'entrées (spécification nominale, graphe d'aléas et graphe d'entrées inopportunes) et le fichier de sortie (spécification augmentée). Les rectangles décrivent les étapes de l'augmentation (nos algorithmes) et les rectangles à angles arrondis décrivent les données

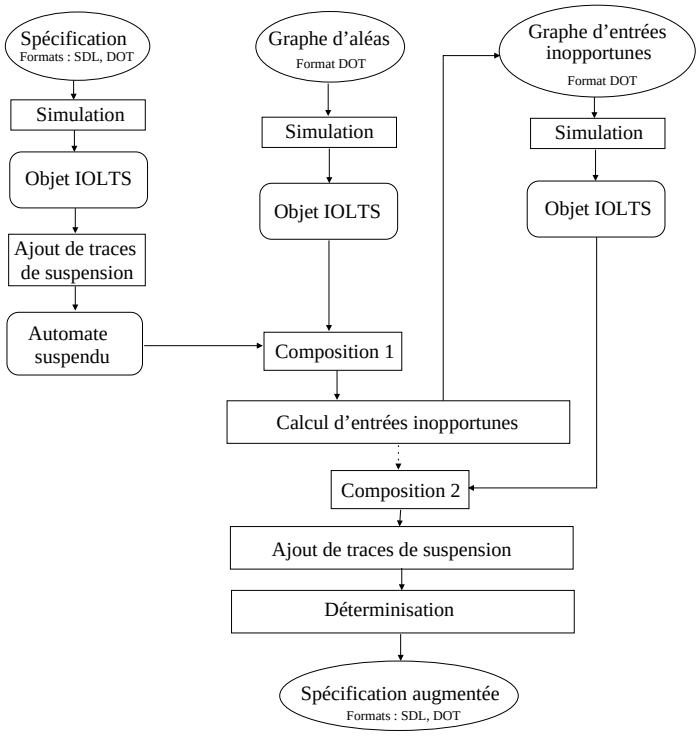


Fig. 47: Architecture de l'interface 1

intermédiaires (automate suspendu, etc).

La classe IOLTS

L'implémentation de notre outil est fondée sur les principes de la programmation objet. À ce stade, nous avons défini la classe IOLTS comme suit (voir Fig.48) :

```
type
    Transition = record
      Etat_source,Etat_cible,Action : string;
       Color : integer;
    end;
    IOLTS = class
      private
          Init_State : string;
          Transitions_Tab : array of Transition;
Public
    Constructor Create (q : string);
    Destructor Destroy;
    Procedure AddTransition (T : Transition);
    Function Determinize : IOLTS;
    Function SuspensionTraces : IOLTS;
    Function Outputs (q : string) : TListeString;
    Function Inputs (q : string) : TListeString;
    Function Composition (s : IOLTS) : IOLTS;
    .....
end;
```

Fig. 48: Déclaration de la classe IOLTS

Une transition est considérée comme un enregistrement contenant les champs : $Etat\_source$, $Etat\_cible$, $Action$ et $Color$. Une instance de la classe IOLTS est composée de variables privées $Init\_State$ et $Transitions\_Tab$ décrivant l'état initial et l'ensemble de transitions d'un IOLTS. Le champ $Color$ est utilisé pour distinguer les transitions de la spécification nominale de celles ajoutées pendant l'augmentation. La méthode $AddTransition$ ajoute une transition dans la table $Transistions\_Tab$ et, elle lui affecte une couleur (0 : nominale, 1 : augmentée). Par ailleurs, toutes les notations standards sur les $IOLTSs$ sont implémentées sous forme de méthodes de la classe IOLTS (procédures et fonctions). Pour simplifier la lecture et la compréhension d'algorithmes, nous utilisons les mêmes notations standards définies dans les paragraphes 5.4.1, 2.3.1 et 2.3.2.

```
// Exemple d'une spécification SDL
PROCESS Emitter-Proc (1,1)
START  NEXTSTATE Closed;
STATE Closed; INPUT CallReq;
  OUTPUT DoCall;
  NEXTSTATE WackCall;
STATE WackCall;
  INPUT AckCall;
  NEXTSTATE Connected;
........
END PROCESS Emitter-Proc.
```

```
// Exemple d'une spécification DOT
Digraph IOLTS
{
....
1 -> 2 [label = "?open"];
2 -> 1 [label = "?close"];
.......
1 [label = "Closed-State];
2 [label = "Closed-State];
..........
}
```

FIG. 49: Exemple de spécifications SDL et DOT

Les fichiers d'entrées que nous utilisons sont formatés en SDL ou en DOT (voir FIG.49). Pour initialiser un objet IOLTS à partir des fichiers SDL ou DOT, nous avons implémenté les deux méthodes $LoadFromSDLFile$ et $LoadFromDOTFile$. Ces deux méthodes sont basées sur une analyse syntaxique permettant d'identifier les transitions d'un IOLTS dérivé à partir de ces spécifications. Ensuite, les deux méthodes remplissent la table de transitions par les données provenant de ces fichiers.

Pour les spécifications SDL contenant plusieurs processus, la méthode $LoadFromSDLFile$ localise et puis sauvegarde tous les processus du système SDL. Ensuite, chaque processus sera utilisé, de manière indépendante, comme une spécification nominale partielle.

Ajout de traces de suspension

Cette étape consiste à identifier les états de blocage (outputlock, deadlock et livelock) dans l'IOLTS da la spécification (voir paragraphe 5.5.1).

L'algorithme 2 parcourt l'IOLTS et vérifie les sorties, entrées et cycles d'actions internes de tout état. Si un état de blocage est détecté, il signale le type de blocage et ajoute une boucle $q \xrightarrow{!\delta} q$. Nous rappelons que $In(q)$ est l'ensemble d'entrées spécifiées dans l'état q. $Out(q)$ est l'ensemble de sorties dans cet état.

Algorithme 2 Ajout de traces de suspension

ENTRÉES: $IOLTS\ S = (Q^S, q_0^S, \Sigma^S, \rightarrow_S)$.
SORTIES: Automate suspendu S^δ.
pour tout $q \in Q^S$ **faire**
si $Out(q) = \emptyset \ \wedge \ In(q) \neq \emptyset \ \wedge \ q \overset{\varepsilon}{\not\Rightarrow} q$ **alors**
Ajouter une boucle $q \xrightarrow{!\delta} q$ {état de blocage de sortie (outputlock)} ;
finsi
si $Out(q) \neq \emptyset \ \vee \ In(q) \neq \emptyset \ \wedge \ q \overset{\varepsilon}{\Rightarrow} q$ **alors**
Ajouter une boucle $q \xrightarrow{!\delta} q$; {état de blocage vivant (livelock)} ;
finsi
si $\forall a \in \Sigma^S \cup \{\tau\},\ q \overset{a}{\not\rightarrow}$ **alors**
Ajouter une boucle $q \xrightarrow{!\delta} q$; {état de blocage de type deadlock} ;
finsi
fin pour

Composition

Après l'exécution de l'algorithme 2, RTCG demande à l'utilisateur d'insérer le fichier d'aléas HG. Ce fichier décrit le comportement du système en présence de certaines entrées invalides. La composition consiste à ajouter les transitions et états de S^δ et HG à l'IOLTS de la composition. Ceci est donné par l'algorithme 3.

Algorithme 3 Composition

ENTRÉES: Deux $IOLTS\ S^\delta = (Q^{S^\delta}, q_0{}^{S^\delta}, \Sigma^{S^\delta}, \rightarrow_{S^\delta})$ et $HG = (Q^{HG}, q_0^{HG}, \Sigma^{HG}, \rightarrow_{HG})$.
SORTIES: IOLTS de la composition $S^\delta \otimes HG$.
$S^\delta \otimes HG := \emptyset$; {création des transitions et états de S^δ}
pour tout $q \xrightarrow{a}_{S^\delta} q'$ **faire**
Ajouter la transition $(q \xrightarrow{a} q')$ à $S^\delta \otimes HG$;
fin pour{Création des transitions et états de HG}
pour tout $q \xrightarrow{a}_{HG} q'$ **faire**
Ajouter la transition $(q \xrightarrow{a} q')$ à $S^\delta \otimes HG$;
fin pour

Calcul et ajout d'entrées inopportunes

Cette étape consiste à verifier les réceptions inattendues ou inopportunes de chaque état de la composition $S^\delta \otimes HG$. Ceci est effectué à l'aide de l'algorithme 4.

Algorithme 4 Calcul d'entrées inopportunes

ENTRÉES: La composition $S^\delta \otimes HG$.
SORTIES: Ensemble d'entrées inopportunes $ref(q)$ de chaque état $q \in S^\delta \otimes HG$.
pour tout $q \in Q^{S^\delta \otimes HG}$ **faire**
$ref(q) = \Sigma_I^{S^\delta \otimes HG} - In(q)$;
fin pour

Après le calcul de la composition $S^\delta \otimes HG$, RTCG retourne une liste d'entrées inopportunes et, demande à l'utilisateur d'insérer le fichier IIG décrivant le comportement du

système en présence de ses entrées inopportunes. En conséquence, RTCG applique à nouveau l'algorithme 3 et, ajoute tous les états et transitions de IIG à la nouvelle composition $S^{\delta} \otimes HG \otimes IIG$. Toutefois, l'utilisateur peut choisir l'augmentation par défaut. Dans ce cas, RTCG ajoute à chaque état des boucles étiquetées par les entrées inopportunes calculées dans l'algorithme 4.

Vérification de traces de suspension et déterminisation

Naturellement, l'ajout d'états dégradés et de sorties acceptables dans la spécification change les informations de blocage. En conséquence, nous devons procéder à une deuxième vérification de traces de suspension. Dans ce but, nous utilisons à nouveau l'algorithme 2. La dernière étape dans cette interface consiste à déterminiser le modèle. Pour ce faire, nous définissons la méthode ε-fermeture.

Définition 29 *on appelle ε-fermeture de l'ensemble d'états $T = \{q_0, q_1, ..., q_n\}$ l'ensemble des états accessibles depuis un état q_i de T par des ε-transitions.*

Le calcul de ε-fermeture de l'ensemble d'états $T = \{q_0, q_1, ..., q_n\}$ est donné par l'algorithme 5 :

Algorithme 5 Calcul de l'ε-fermeture de $T = \{q_0, q_1, ..., q_n\}$

ENTRÉES: $T = \{q_0, q_1, ..., q_n\}$.
SORTIES: ε-fermeture(T).
tantque $Empty(P) = false$ **faire**
  Soit $Top(p)$ le sommet de la pile, $Depiler(P)$
  pour tout État q tel qu'il y a une ε-transition entre $Top(P)$ et q **faire**
    si $q \notin$ ε-fermeture(T) **alors**
      Ajouter q à ε-fermeture(T) ;
      $Empiler(P, q)$
    finsi
  fin pour
fin tantque

Cet algorithme utilise une pile P, les méthodes (fonctions et procédures) standards $Top(P)$, $Empiler(P, x)$ et $Depiler(P)$ ont pour but de retourner le sommet de la pile, mettre une valeur x dans le sommet de la pile et supprimer la valeur en sommet de la pile. $Empty(P)$ est une fonction logique permettant de vérifier l'état de la pile (vide ou non). L'algorithme 6 calcule l'IOLTS déterministe à partir d'un IOLTS non déterministe.

Algorithme 6 Déterminisation d'un IOLTS

ENTRÉES: $IOLTS\ S = (Q^S, q_0^S, \Sigma^S, \rightarrow_S)$ non déterministe.
SORTIES: IOLTS $\Delta(S) = (Q^{\Delta(S)}, q_0^{\Delta(S)}, \Sigma^{\Delta(S)}, \rightarrow_{\Delta(S)})$ déterministe.
Calculer ε- fermeture(q_0^S)
répéter
  Ajouter dans la table de transitions toutes les $\varepsilon - fermeture$ des nouveaux états produits, avec leurs transitions
jusqu'à ce qu'il n'y ait plus de nouvel état.

7.2.2 Interface 2 : Génération des cas de test de robustesse

Cette interface permet de générer les cas de test de robustesse à partir de la spécification augmentée S_A et un objectif de test de robustesse RTP. D'abord, l'utilisateur est interrogé pour insérer deux fichiers, l'un de la spécification augmentée S_A et l'autre de la spécification nominale S. RTCG procède à l'ajout de traces de suspension, puis à la déterminisation de la spécification nominale. Ensuite, il compare, au fur et à mesure, les transitions de la spécification augmentée à celles de $\Delta(S^\delta)$. Pendant cette comparaison, RTCG attribue une couleur à l'ensemble de transitions de $\Delta(S^\delta)$ et une autre couleur à l'ensemble de transitions hors de $\Delta(S^\delta)$. La coloration de transitions sera utilisée comme un critère de sélection des cas de test de robustesse.

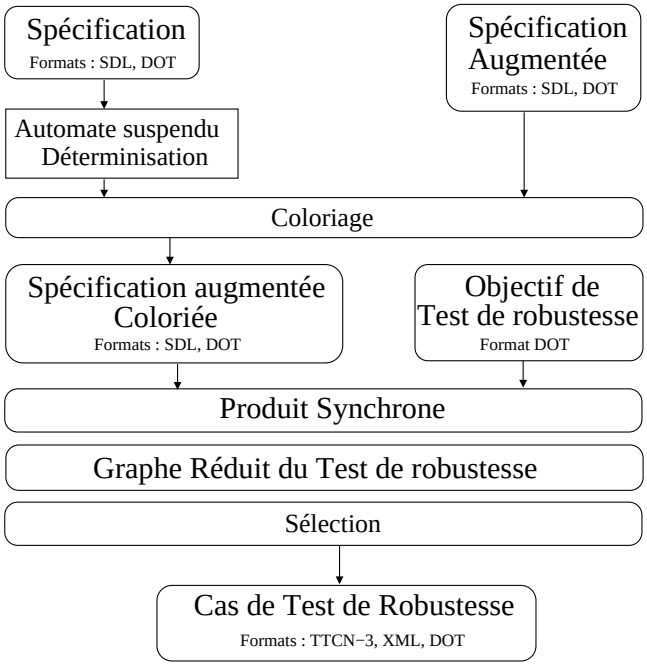


Fig. 50: Architecture de l'interface 2

L'utilisateur est ensuite interrogé pour insérer un autre fichier décrivant l'objectif du test de robustesse et fixer les états initiaux de chaque IOLTS, ainsi que l'état *Accept* de RTP. Les objectifs utilisés dans la version actuelle de RTCG utilisent seulement les deux étiquettes *Accept* et *other* (*Reject* n'est pas encore utilisée). Les différents algorithmes de cette interface sont donnés ci-dessous :

Interprétation de l'objectif du test de robustesse

D'abord, RTCG procède à l'interprétation de l'objectif de test de robustesse. Celle-ci consiste à remplacer chaque transition étiquetée par *other* par l'ensemble de transitions

correspondant. Ceci est calculé par l'algorithme 7.

Algorithme 7 Interprétation de RTP

ENTRÉES: RTP abstrait
SORTIES: $IOLTS$ de RTP
pour tout état q dans RTP tel que $q \xrightarrow{other}_{RTP} q'$ **faire**
  pour tout action a dans $\Sigma^{S_A} - IO(q)$ **faire**
    Ajouter une transition $q \xrightarrow{a}_{RTP} q'$;
  fin pour
  Supprimer $q \xrightarrow{other}_{RTP} q'$;
fin pour

Pour toute transition T de RTP étiquetée par *other*. Pour chaque action a non spécifiée dans l'état source de T, on ajoute une transition étiquetée par a.

Construction des graphes du test de robustesse

Afin de calculer toutes les traces, de la spécification augmentée, satisfaisant l'objectif de test de robustesse, nous avons besoin de calculer leur produit synchrone. Partant de la définition 27, nous proposons l'algorithme suivant :

Algorithme 8 Le produit synchrone

ENTRÉES: Deux IOLTSs, $S_A = (Q^{S_A}, q_0^{S_A}, \Sigma^{S_A}, \rightarrow_{S_A})$ et $RTP = (Q^{RTP}, q_0^{RTP}, \Sigma^{RTP}, \rightarrow_{RTP})$
SORTIES: IOLTS $S_A \otimes RTP = (Q^{S_A \otimes RTP}, q_0^{S_A \otimes RTP}, \Sigma^{S_A \otimes RTP}, \rightarrow_{S_A \otimes RTP})$
$Q^{S_A \otimes RTP} := \emptyset$;
$q_0^{S_A \otimes RTP} = (q_0^{S_A}, q_0^{RTP})$;
$Q^{S_A \otimes RTP} := \{q_0^{S_A \otimes RTP}\}$;
pour tout $q^{S_A \otimes RTP} = (q_j^{S_A}, q_k^{RTP}) \in Q^{S_A \otimes RTP}$ **faire**
  pour tout $a \in IO(q_j^{S_A}) \cup IO(q_k^{RTP})$ **faire**
    si $(q, q') \in Q^{S_A \otimes RTP}$, $q' \neq$ **Accept**, $q \xrightarrow{a}_{S_A} q_1$, $q' \xrightarrow{a}_{RTP} q'_1$ **alors**
      Ajouter la transition $(q, q') \xrightarrow{a}_{S_A \otimes RTP} (q_1, q'_1)$;
    finsi
    si $(q, q') \in Q^{S_A \otimes RTP}$, $q' \neq$ **Accept**, $q \xrightarrow{a}_{S_A} q_1$, $q' \not\xrightarrow{a}_{RTP}$ **alors**
      Ajouter la transition $(q, q') \xrightarrow{a}_{S_A \otimes RTP} (q_1, q')$;
    finsi
    si $(q, q') \in Q^{S_A \otimes RTP}$, $q' \neq$ **Accept**, $q \not\xrightarrow{a}_{S_A}$, $q' \xrightarrow{a}_{RTP} q'_1$ **alors**
      Ajouter la transition $(q, q') \xrightarrow{a}_{S_A \otimes RTP} (q, q'_1)$;
    finsi
  fin pour
fin pour

L'algorithme 8 utilise une liste d'états pour enregistrer les états créés de $S_A \otimes RTP$ et une table dynamique pour stocker les transitions de $S_A \otimes RTP$. $IO(q)$ décrit l'ensemble d'actions (entrées et sorties) tirables dans l'état q. c'est à dire $IO(q) = In(q) \cup Out(q)$. Nous rappelons que les règles de synchronisation ne sont pas applicables sur l'état **Accept**.

Le graphe du test de robustesse RTG est l'image miroir du produit synchrone $S_A \otimes RTP$ (les entrées deviennent sorties et inversement). Pour ce faire, il suffit de parcourir toutes les transitions de $S_A \otimes RTP$ en remplaçant le symbole ? par ! et inversement.

Algorithme 9 Réduction du graphe de test de robustesse *RTC*

Entrées: Le graphe de test de robustesse *RTG*
Sorties: Le graphe réduit de test de robustesse *RRTG*
Étape 1 : récupération de l'ensemble d'état depuis lesquels *Accept* est accessible.
{$VisitedStates$ est la liste d'états à visiter}
$VisitedStates := \emptyset$;
{$CurrentState$ est l'état actuel}
$CurrentState := Accept$;
répéter
$VisitedStates := VisitedStates \cup Predecessors(CurrentState)$;
Passer à l'état suivant dans la liste $VisitedStates$;
jusqu'à ce qu'il n'y a aucun état à ajouté ;
Étape 2 : Construction de *RRTG*.
$RRTG := \emptyset$;
pour tout transition $q \xrightarrow{a} q'$ dans *RTG* **faire**
si $(q \in VisitedStates) \wedge (q' \in VisitedStates)$ **alors**
Ajouter la transition $q \xrightarrow{a} q'$ dans *RRTG* ;
finsi
si $(q \in VisitedStates) \wedge (q' \notin VisitedStates) \wedge (\exists a \in \Sigma_I^{RTG} \mid q \xrightarrow{a} q')$ **alors**
Ajouter la transition $q \xrightarrow{a} Inconc$ dans *RRTG* ;
finsi
fin pour

Le graphe du test de robustesse RTG contient certaines traces de la spécification augmentée qui ne satisfont pas l'objectif RTP. En conséquence, il est nécessaire de réduire ce graphe en se concentrant uniquement sur l'ensemble d'états/transitions depuis lesquels l'état *Accept* est accessible, ainsi que toute réception partant de cet ensemble. L'algorithme 9 assure cette réduction. Cet algorithme collecte d'abord une liste d'états ($VisitedStates$) depuis lesquels l'état *Accept* est accessible. Pour ce faire, il enregistre l'ensemble de prédécesseurs ($predecessors(q)$) de chaque état q dans la liste $VisitedStates$. Ensuite, l'algorithme parcourt RTG et ajoute toute transition dont la source et la cible font partie de $VisitedStates$, ainsi que toute transition sortant de $VisitedStates$ par une réception.

Sélection d'un cas de test de robustesse

Pour dériver un cas de test de robustesse RTC, nous avons implémenté un algorithme de génération à la volée (on-the fly) avec quelques règles supplémentaires privilégiant les entrées invalides et les entrées inopportunes. Pour ce faire, nous avons implémenté l'algorithme 1 présenté dans le chapitre 6. Cet algorithme parcourt le graphe du test de robustesse RTG à partir de son état initial. Dans chaque état visité, s'il s'agit d'un :

1. État de réception, on prend alors toutes les réceptions spécifiées dans cet état ;
2. État d'émission et réception, on fait un choix aléatoire entre l'une ou l'autre. Selon le choix, s'il s'agit d'un état de.
   (a) Émission, on choisit uniquement une seule transition (on donne l'avantage aux émissions coloriées différemment de celles de la spécification nominale) ;
   (b) Réception, on prend alors toutes les réceptions spécifiées dans cet état ;

Dans chaque état de réception, on ajoute une transition, etiquetée par " *?other*", vers un état sans successeur représentant le verdict **Fail**.

Description des cas de test en langage TTCN-3

TTCN (*Tree and Tabular Combined Notation*) est un langage normalisé dans les recommandations Z.140-146 de ITU-T [38] et ISO 9646-3 [91]. Ce langage est un moyen pour représenter les séquences de test en distinguant les symboles que le testeur doit émettre et recevoir. TTCN (versions 1 et 2) permet de décrire des séquences de test abstraites en deux formats : TTCN.GR (Graphic) qui modélise les séquences de test par des tableaux, et TTCN.MP (Machine Processable) qui traduit les séquences de test en format supporté par les testeurs automatiques. La représentation tabulaire (TTCN.GR) d'une suite de test comporte deux parties :

1. une partie statique ou déclarative décrivant les points de contrôle et d'observation (PCOs), les unités de données du protocole (PDUs) ainsi que les horloges ;
2. une partie dynamique décrivant la séquence du test, c'est à dire les émissions et les réceptions, l'interrogation et la réinitialisation d'horloges ainsi que les verdicts.

TTCN-3 (*Testing and Test Control Notation*) [58] est la version actuelle de TTCN. Le but de TTCN-3 est d'élargir le domaine d'utilisation aux autres types du test. La norme TTCN-3 inclut plusieurs formats (facultatifs) de présentation : format textuel avec une syntaxe proche de celle des langages de programmation, format tabulaire (TFT) qui est bien connu aux utilisateurs des versions plus anciennes de TTCN, et un format graphique (GFT) qui exprime les séquences de test sous une forme graphique.

Dans notre outil RTCG, nous utilisons le format textuel pour décrire les cas de test de robustesse. Dans TTCN-3, un cas de test comporte trois parties :

1. partie interface (runs on) qui décrit le type de MTC (Main Test Component) ;
2. partie système (system) décrit le type de l'interface du système ;
3. partie comportement (behaviour part) décrivant le comportement du MTC.

En général, un cas de test possède la syntaxe suivante :

```
testcase <nom de cas de test> runs on <partie interface> system
<partie système>
 timer ReponseTimer := temps de réponse;
// partie préambule
L1.send(Message0)
 ...........
// test
L1.send(Message1) ReponseTimer.start
alt {
    [contrainte 1]    L1.receive(Message1)
                      {faire quelque chose
                      }
    [contrainte 2]    L1.receive(Message2)
                      {...........}
    [contrainte 3]    L1.receive(Mesage3)
                      {...........}
    [else]            {faire quelque chose}
    }
...........
```

après la réception d'un message, le testeur exécute une affectation d'un verdict ($setverdict(pass)$, $setverdict(inconc)$, $setverdict(fail)$), démarrage ou arrêt d'un timer

(*ReponseTimer.start*, *ReponseTimer.stop*) ou un appel à une fonction particulière.

7.2.3 Autres fonctionnalités

En ce qui concerne la méthode TRACON de notre approche, RTCG offre une autre interface basée sur les mêmes algorithmes précédents (ajout de traces de suspension, composition, déterminisation) pour calculer la spécification semi-augmentée. Ensuite, il réutilise les algorithmes 8 et 9 pour construire le graphe réduit du test de robustesse, puis l'algorithme 1 afin d'obtenir un cas de test de robustesse ordinaire. RTCG procède à une deuxième synchronisation avec un contrôleur standard d'aléas (un IOLTS de 2 états et 2 transitions *start* et *stop*) pour obtenir le graphe des cas de test de robustesse non contrôlables. Enfin, RTCG applique un algorithme de sélection aléatoire (parcours en arrière) pour calculer un cas de test de robustesse contrôlable (CRTC).

En plus des deux méthodes de notre approche, RTCG permet de faire l'aspect du test passif sur le test de robustesse. Cette fonctionnalité consiste à vérifier l'appartenance d'une trace d'exécution T, complètement spécifiée, à la spécification augmentée. Pour ce faire, nous appliquons un algorithme très similaire à l'algorithme 8, seulement avant de faire la synchronisation entre un état q' de T et un état q de S_A, nous vérifions d'abord l'inclusion de toutes les entrées/sorties tirables dans l'état q' de T dans celles de l'état q de S_A. L'algorithme s'arrête une fois que l'inclusion est non satisfaite.

7.3 Études de cas

Cette section présente deux études de cas visant à la fois l'évaluation théorique de notre approche, ainsi qu'une évaluation quantitative de l'outil *RTCG*. Á travers l'évaluation théorique, nous montrons quelques exemples d'aléas (entrées invalides, entrées inopportunes et sorties acceptables). En ce qui concerne l'évaluation de notre outil, nous construisons deux spécifications, en termes d'$IOLTS$, pour les protocoles SSL handshake et TCP. Ensuite, nous adoptons quelques hypothèses permettant d'augmenter ces spécifications. Enfin, nous montrons quelques résultats démonstratifs concernant la génération des cas de test de robustesse.

7.3.1 Le protocole SSL Handshake

Le protocole SSL (*Secure Socket Layer*) est un protocole de sécurisation des échanges, développé par Netscape. Il a été conçu pour assurer la sécurité des transactions sur Internet (notamment entre un client et un serveur). SSL est intégré depuis 1994 dans les navigateurs. Il existe plusieurs versions : la version 2.0 développée par Netscape ; la version 3.0 qui est actuellement la plus répandue, et la version 3.1 baptisée TLS (*transport Layer Security*) et standardisée par l'IETF (*Internet Engineering Task Force*). SSL fonctionne de manière

indépendante par rapport aux applications qui l'utilisent ; il est obligatoirement au dessus de la couche TCP et certains le considèrent comme un protocole de niveau 5 (couche session). Plusieurs implémentations du protocoles SSL/TLS sont mises en œuvre (Open SSL, SSLeay, BSAFF 3.0, SSL Plus, SSL Ref 3.0, etc). Les spécifications standards du protocole (version 1.1) sont données dans la RFC 2246 [81, 52]. Le protocole SSL/TLS est constitué de quatre sous-protocoles : SSL *Handshake*, SSL *Changes Cipher Spec*, SSL *Alert* et SSL *Record*.

Notre étude de cas est centrée seulement sur le protocole SSL handshake, celui-ci permet l'authentification mutuelle du serveur et du client et la négociation pour choisir les suites de chiffrement qui seront utilisées lors de la session. L'authentification du serveur est obligatoire et celle du client est optionnelle. Le principe du fonctionnement est illustré dans la FIG.51.

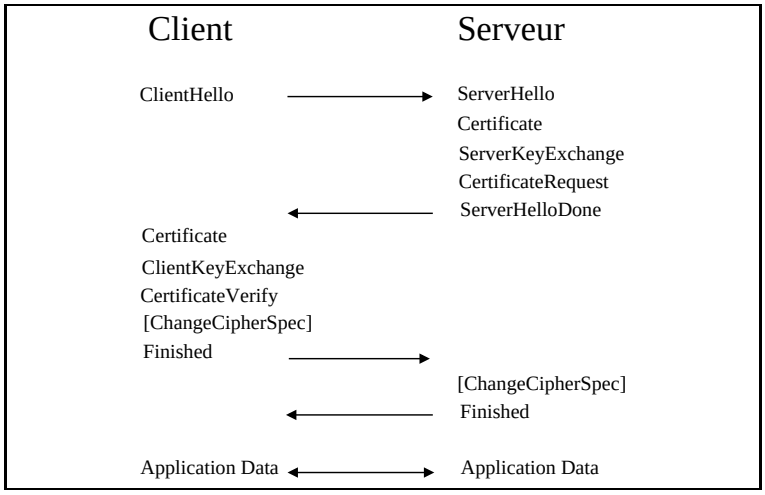


FIG. 51: Négociation d'une connexion entre le client et le serveur

- `ClientHello`, `ServerHello` : chaque message contient la version du protocole SSL, un nombre aléatoire, un identificateur de session, une liste des suites de chiffrement choisies et une liste des algorithmes de compression. `ClientHello` $= <$ *version*, *client_random*, *session_id*, *ciphersuite*, *algodcompression*$>$ et `ServerHello`$=<$ *laversion*, *Server_random*, *session_id*, *ciphersuite*, *algodcompression* $>$;
- `Certificate` contient le certificat du serveur ;
- `ServerKeyExchange` contient le certificat de signature ;
- `CertificateRequest` : le serveur réclame un certificat au client ;
- `ServerHelloDone` : la fin de l'envoi de message ;
- `ClientKeyExchange` contient le `Pre-Master-secret` crypté à l'aide la clé publique du serveur ;
- `CertificateVerify` : vérification explicite du certificat du client ;
- `Finished` : fin du protocole Handshake et le début de l'émission des données.

Spécification nominale

Les standards précédents décrivent essentiellement trois modes pour négocier une connexion sécurisée entre le client et le serveur (voir TAB.1) :

Cas	Séquences
Session non identifiée / Client non authentifié	`!Client-Hello`, `?Server-Hello`, `?Client-Master-Key`, `?Client-Finished`, `!Server-Verify`, `!Server-Finished`
Session identifiée / Client non authentifié	`!Client-Hello`, `?Server-Hello`, `?Client-Finished`, `!Server-Verify`, `!Server-Finished`
Session identifiée / Client authentifié	`!Client-Hello`, `?Server-Hello`, `?Client-Master-Key`, `?Client-Finished`, `!Server-Verify`, `?Request-Certificate`, `!Client-Certificate`, `!Server-Finished`

Tab. 1: Les traces nominales du protocole SSL Handshake

En plus des trois traces nominales présentées dans la TAB.1, les standards du protocole ont traité quatres modèles de fautes (messages d'erreur) : l'absence du certificat (`No-Certificate-Error`), mauvais certificat (`Bad-Certificate-Error`), certificat non supporté (`Unsupported-Certificate-Type-Error`) et l'absence d'algorithme du chiffrement (`No-Cipher-Error`).

Spécification augmentée

Dans la littérature concernant l'analyse du protocole SSL/TLS , les auteurs de [25] ont identifié deux messages d'erreur qui ne sont pas signalés dans les standards du protocole. Il s'agit de `Unsupported-Authentification-Type-Error` et `Unexpected-Message-Error`. Le premier est un message d'erreur qui peut se produire pour empêcher le protocole d'utiliser certaines méthodes pour authentifier un client. Le deuxième a pour but de signaler la réception d'un message inconnu. Ceci provoque la fermeture exceptionnelle d'une connexion.

Le test de conformité, basé sur la spécification nominale (le standard du protocole), est incapable de donner des verdicts concernant les comportements d'une implémentation contenant l'un des deux messages précédents. Du point de vue du test de robustesse, les deux messages précédents sont considérés comme des *entrées invalides* car ils ne sont pas spécifiés dans les standards du protocole. En conséquence, il semble nécessaire d'augmenter la spécification nominale afin d'évaluer tout comportement incluant ces messages.

Pour appliquer notre approche sur le protocole SSL handshake, nous avons construit un IOLTS (voir Fig.52) décrivant la spécification nominale du protocole SSL handshake. Cet IOLTS est inspiré de la TAB.1.

Pour construire la spécification augmentée, nous avons choisi les aléas suivants :

1. **Entrées invalides**
   Nous avons utilisé les deux messages `Unsupported-Authentification-Type-Error` et `Unexpected-Message-Error`. Bien entendu, on peut ajouter également d'autres classes d'équivalence modélisant les erreurs envisageables dans chaque message spécifié.
2. **Entrées inopportunes**
   Dans chaque état q de l'IOLTS de la spécification, nous avons calculé automatiquement les entrées (messages) qui peuvent être inopportunes ou imprévues dans cet état. Ce calcul est basé sur l'alphabet de la spécification nominale, (les messages décrites dans TAB.1 et les 4 messages d'erreur décrits dans la RFC), ainsi que les entrées invalides citées ci-dessus.
3. **Sorties acceptables**
   Pour compléter la spécification du protocole, nous supposons que le protocole ferme la connexion après la réception de `Unexpected-Message-Error` ou `Unsupported-Authentification-Type-Error`. En revanche, le système conserve son état de départ en cas de réception d'une entrée inopportune.

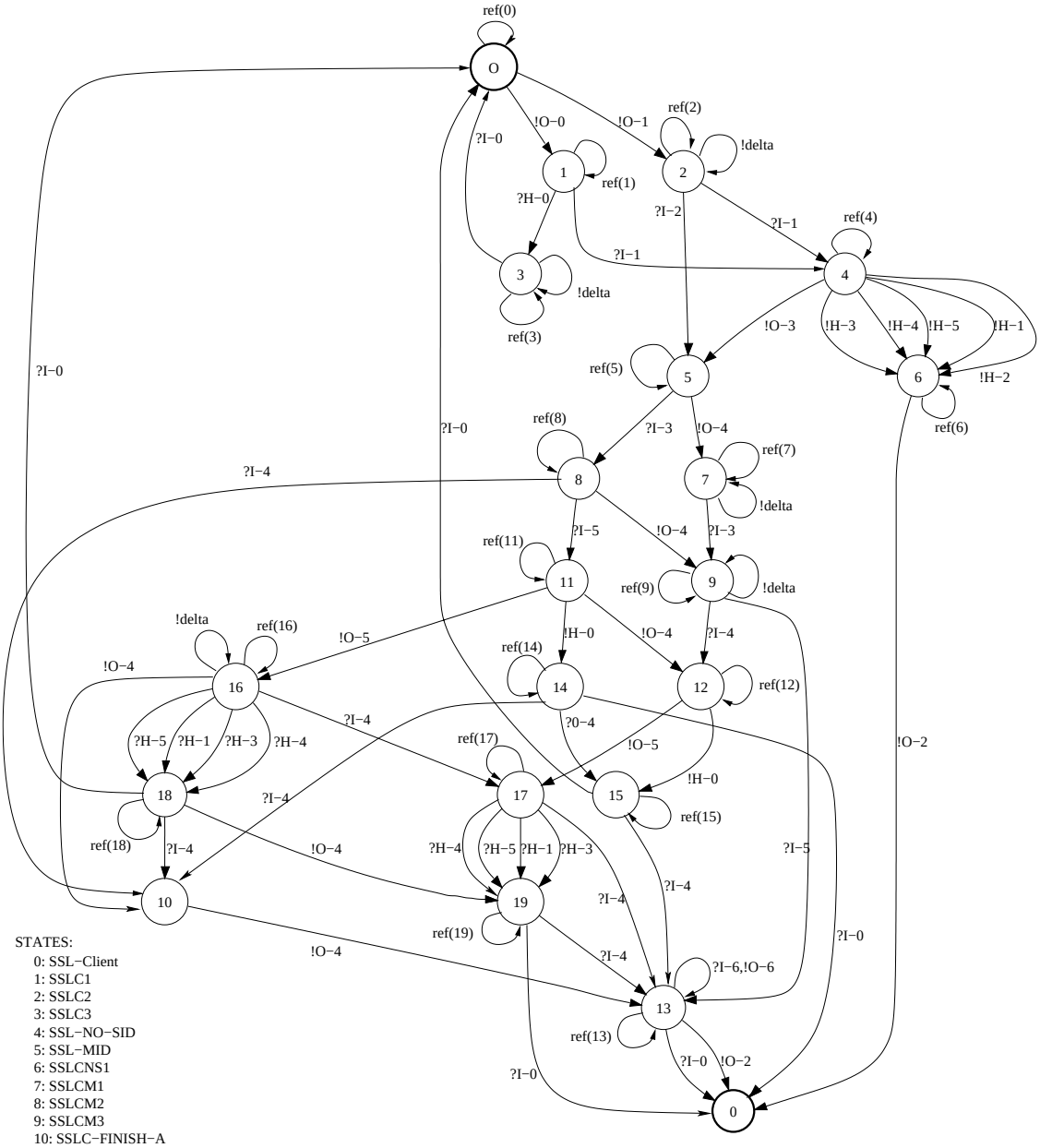
L'IOLTS de la spécification augmentée est donné par la FIG.52. Pour des raisons de lisibilité, nous avons utilisé l'étiquette $ref()$ sur les boucles, mais dans l'étude de cas (voir annexe A), cette étiquette est remplacée par les entrées inopportunes de tout état.

Résultats avec un outil du test de conformité

Afin de tester la robustesse du protocole SSL Handshake en tenant compte des différents aléas introduits auparavant, nous avons défini un ensemble d'objectifs de test de robustesse visant à tester le comportement de l'implémentation en présence d'aléas au niveau du certificat (`No-Certificate-Error`, `Bad-Certificate-Error`, `Unsupported-Certificate-Type-Error`), au niveau de l'algorithme de chiffrement (`No-Cipher-Error`) et au niveau des aléas `Unexpected-Message-Error` et `Unsupported-Authentification-Type-Error` :

- **RTP1** : Le maintien de la connexion entre le serveur et le client après la détection d'un certificat non approprié.
- **RTP2** : Fermeture de la connexion après la détection d'un problème au niveau de l'algorithme de chiffrement.
- **RTP3** : Le maintien de la connexion entre le serveur lors d'un message d'erreur non attendu.

Notre première étude de cas a été focalisée sur l'utilisation d'un outil de génération des tests de conformité. Pour ce faire, nous avons utilisé l'outil TGSE [17] développé au LaBRI. Le tableau 3 résume les différents résultats obtenus avec une implémentation de TGSE sur une station Linux Fedora 3 (Intel Pentium 4 CPU 1.80GHz, mémoire 128Mo). "Taille RTC" représente la taille moyenne d'un cas de test vérifiant une propriété et, "Temps CPU" le temps CPU moyen pour générer un cas de test.



STATES:
0: SSL−Client
1: SSLC1
2: SSLC2
3: SSLC3
4: SSL−NO−SID
5: SSL−MID
6: SSLCNS1
7: SSLCM1
8: SSLCM2
9: SSLCM3
10: SSLC−FINISH−A
11 SSLC−CERT−A
12: SSLC−CERT−B
13: SSLC−FINISH−B
14: SSLCC1
15: SSLCC2
16: SSLCC3
17: SSLCC4
18: SSLCC5
19: SSLCC6

INPUTS
?I−0: ?Close−Connection
?I−1: ?Server−Hello(No−Sid)
?I−2: ?Server−Hello(Hit)
?I−3: ?Server−Verify
?I−4: ?Server−Finished
?I−5: ?Server−Request−Certificate
?I−6: ?SSL−Data−Record

OUTPUTS
!O−0: !Client−Hello(No−SId)
!O−1: !Client−Hello(Sid)
!O−2: !Close−Connection
!O−3: !Client−Master−Key
!O−4: !Client−Finished
!O−5: !Client−Certificate
!O−6: !SSL−Data−Record

HAZARDS
H−0: No−Certificate−Error
H−1: Bad−Certificate−Error
H−2: No−Cipher−Error
H−3: Unsupported−Certificate−Type−Error
H−4: Unsupported−Authentication−Type−Error
H−5: Unexpected−Message−Error

Fig. 52: La spécification augmentée du protocol SSL Handshake

Propriété	Taille RTC	Temps CPU (ms)
RTP1	53	961,8
RTP2	10	359,9
RTP3	20	157,97

TAB. 2: Expérimentation avec TGSE

Signalons que dans cette étude de cas, la spécification augmentée a été construite à la main car cet outil est entièrement dédié au test de conformité et par conséquent, il n'offre aucun moyen pour augmenter la spécification. De plus, le choix d'objectifs de test de robustesse était basé sur les entrées invalides (les deux messages d'erreur précédents) afin de garantir que les cas de test générés décrivent la robustesse et non pas la conformité.

Les résultats obtenus montre que les cas de test de robustesse sont relativement volumineux par rapport à ceux du test de conformité en raison de la considération d'aléas.

Expérimentations avec RTGC

Méthode TRACOR

En ce qui concerne l'utilisation de RTCG pour la génération des cas de test de robustesse. D'abord, la spécification augmentée est calculée automatiquement à partir de la spécification nominale du protocole, un graphe d'aléas et une augmentation par défaut sur l'ensemble d'entrées inopportunes (voir ANNEXE A.1).

Nous avons défini ensuite un ensemble d'objectifs de test de robustesse (voir ANNEXE A.1) visant à vérifier les trois traces nominales, présentées dans la TAB.1, en présence d'entrées invalides (`Unexpected-Message-Error` et `Unsupported-Authentification-Type-Error`), ainsi que toutes les entrées inopportunes de tout état de l'IOLTS de la spécification :

1. **RTP1** couvre la première trace de la TAB.1 en présence d'aléas précédents ;
2. **RTP2** couvre la deuxième trace de la TAB.1 en présence d'aléas précédents ;
3. **RTP3** couvre la troisième trace de la TAB.1 en présence d'aléas précédents.

Le tableau suivant récapitule les différents résultats obtenus avec une implémentation de RTCG sur une station (WindowsXP©, Pentium(R)4 CPU 2.80 GHz, RAM 256 Mo). "Propriété" décrit le RTP utilisé. "Taille RTC" la taille moyenne d'un cas de test de robustesse.

Propriété	Taille RTC	Temps CPU (ms)
RTP1	11	1.7965
RTP2	14	2.6807
RTP3	19	4.4749

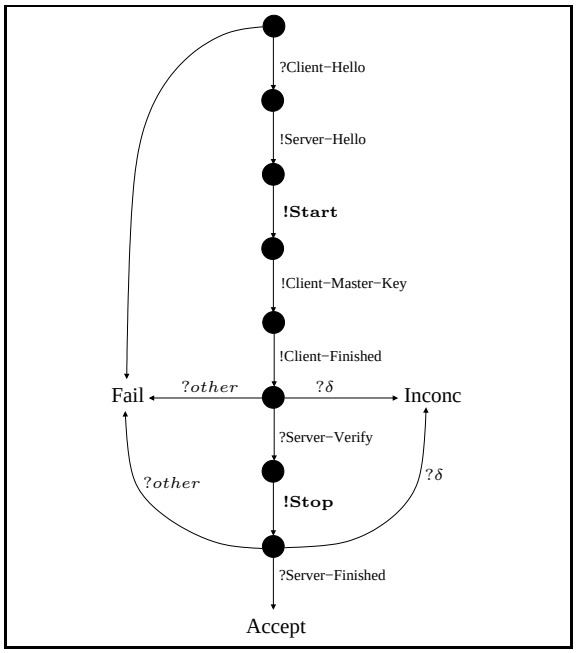
TAB. 3: Résultats obtenus avec RTCG sur le protocole SSL handshake

Les cas de test de robustesse obtenus avec l'outil RTCG sont moins volumineux par rapport aux résultats obtenus avec TGSE. Précisément, dans l'outil RTCG, les objectifs de test utilisés décrivent des fonctionnalités (les traces de la TAB.1) en présence d'aléas. Ceci réduit énormément la taille du graphe réduit du test de robustesse et par conséquent la taille de cas de test de robustesse.

Méthode TRACON.
En ce qui concerne l'application de la Méthode TRACON, nous avons réalisé l'étude de cas suivante (voir ANNEXE A.2) :

Spécification semi-augmentée. Nous avons fait l'hypothèse qu'une implémentation robuste ne devrait émettre aucune sortie additionnelle en présence d'aléas contrôlables et non représentables. De plus, les traces dérivées de la spécification nominale doivent être exécutables en dépit d'aléas. En conséquence, nous avons utilisé l'automate suspendu de la spécification nominale comme référence pour le test de robustesse.

Le processus de la génération de cas de test de robustesse se fait en suivant les mêmes étapes que la Méthode TRACOR. De plus, les cas de test de robustesse obtenus sont composés avec un contrôleur d'aléas standard HC (2 états, 2 transitions `?start`, `?stop`). La FIG.53 montre un exemple de cas de test de robustesse contrôlable, généré avec RTCG, visant à vérifier l'exécutabilité de la première trace de la spécification nominale présentée dans la TAB.1.




```
tescase SSL-CRTC() runs on SSL-IUT {timer ReponseTimer := 100E-3 ;
  ReponseTimer.start
  alt{ [] ReponseTimer.timeout
         {  setverdict(fail);
            stop }
       [] Tester.receive(Client-Hello);
         {  setverdict(pass);
            ReponseTimer.stop
          Tester.send(Server-Hello);
          Tester.send(Start);
          Tester.send(Client-Master-Key);
          Tester.send(Client-Finished);
          ReponseTimer.start
          alt { [] ReponseTimer.timeout
                   {  setverdict(inconc);
                     stop }
                [] Tester.receive(Server-Verify);
                   {  setverdict(pass);
                     ReponseTimer.stop
                     Tester.send(Stop);
                     alt { [] ReponseTimer.timeout
                              {  setverdict(inconcn);
                                stop}
                           [] Tester.receive(Server-Finished);
                              {  setverdict(pass);
                                 ReponseTimer.stop}
                          [else] { setverdict(fail);
                                   stop }
                         }
                [else]  { setverdict(fail);
                          stop
                         }
           }}
}}}
```

FIG. 53: Exemple de CRTC généré avec RTCG

Pour exécuter le cas de test de robustesse présenté dans la FIG.53, le testeur de robustesse émet, après la réception de l'entrée `?Client-Hello`, la sortie `!Server-Hello` et, envoie la commande `!Start` au contrôleur d'aléa pour activer l'exécution d'aléa. Ensuite, le testeur de robustesse émet respectivement les sorties `!Client-Master-Key` et `!Client-Finished`. S'il reçoit l'entrée `?Server-Verify`, il arrête l'exécution d'aléa par l'émission de la commande `!Stop`. Enfin, si le testeur reçoit `?Server-Finished`, il émet le verdict *Accept* et, arrête l'exécution du cas de test de robustesse. Dans les deux états de réception, si le testeur reçoit d'autres entrées respectivement différentes à `?Client-Hello` et `?Server-Verify`, il émet un verdict *Fail* et arrête l'exécution.

7.3.2 Le protocole TCP

TCP est un protocole de transport fonctionnant en mode connecté [150]. Avant qu'une connexion soit établie, les deux entités désirant communiquer jouent des rôles dissymétriques. L'un des programmes d'application, appelé communément serveur, à l'une des deux extrémités, exécute une fonction d'ouverture passive de communication. Pour ce faire, il prend contact avec son système d'exploitation afin de lui indiquer qu'il est susceptible d'accepter une demande de connexion entrante. Le système d'exploitation lui fournit alors un port de communication désigné par un numéro. A l'autre extrémité, le programme d'application, appelé client, doit exécuter une demande d'ouverture active pour demander l'établissement d'une connexion.

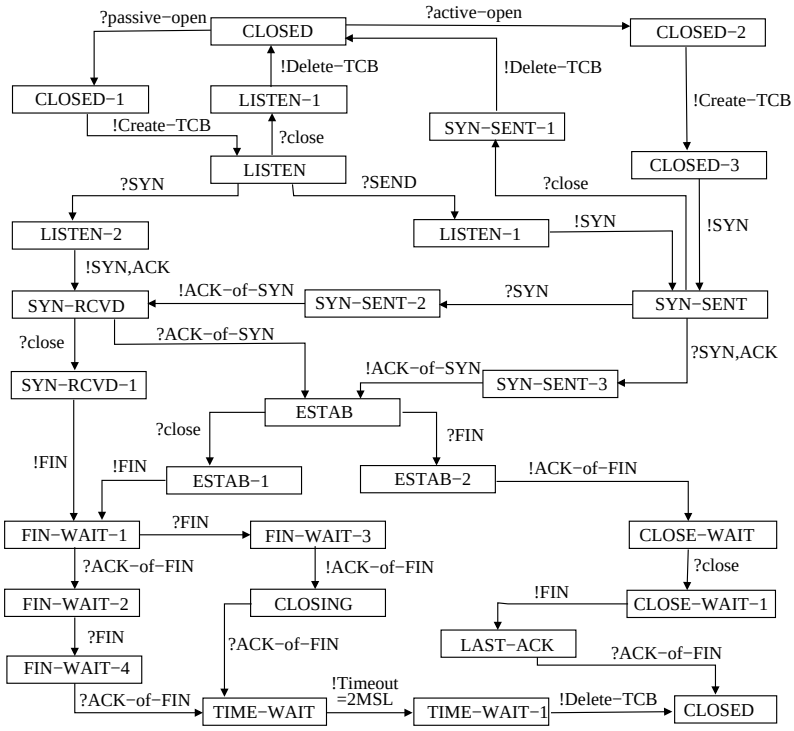


FIG. 54: IOLTS d'une connexion TCP

La gestion des connexions de TCP dans les conditions nominales est décrite, de manière simplifiée, en SDL (voir ANNEXE B.1), et à l'aide d'un système de transitions étiquetées IOLTS (voir Fig.54). L'un des buts principaux de l'ouverture d'une connexion TCP est de fixer, de manière non équivoque, les numéros de séquences utilisés pour la numérotation des messages, appelés segments, qui sont échangés par les deux extrémités de la connexion pendant la durée de la communication. La Fig.54 illustre une partie de la spécification décrivant l'ouverture et fermeture d'une connexion TCP.

Scénarii d'aléas

Voici deux scénarii d'aléas concernant le protocole TCP, ils se distinguent les uns des autres par leur influence sur le comportement du système.

Le système change l'état de son fonctionnement. Un retour à un état dit "état initial" est un cas particulier de ce type de fonctionnement. Beaucoup de systèmes utilisent un retour à l'état initial, ou réinitialisation, en cas de problème. Les protocoles réseau, en général, et TCP, en particulier, ne font pas exception à cette règle. Ainsi, le RFC décrivant TCP précise, de manière informelle, qu'une séquence de réinitialisation doit être entreprise à chaque fois que l'un des hôtes impliqués dans une connexion a acquis la conviction que les numéros de séquence des segments échangés sont incohérents. Plusieurs exemples d'aléas provoquent ce type de fonctionnement : c'est l'un d'entre eux qui est utilisé dans cette étude de cas et qui est rappelé ci-dessous.
Soit S, un hôte serveur d'adresse IP ip_S, en attente passive de connexion TCP sur un port de numéro $port\_S$ et soit C, d'adresse ip_C, un hôte client de S. Afin d'établir une connexion avec S, C émet un segment de demande de connexion, CNX_C1, en provenance de son port $port\_C$ et à destination du port $port\_S$ de S ; ce segment porte le numéro de séquence initial seq_C_1. La réponse de S ne parvient pas de suite à C à cause d'une congestion du réseau conduisant à un retard de CNX_C_1 ou de la réponse de S. Ceci entraîne la retransmission de CNX_C_1 par C. Le paquet ainsi réémis est noté $CNX\_C'_1$; il est en tout point identique à CNX_C_1. Finalement, CNX_C_1 est acquitté par S grâce au segment ACK_S_1, portant l'acquittement $ack\_S_1 = seq\_C_1 + 1$, et la connexion est normalement établie en suivant les règles de l'ouverture d'une connexion TCP en trois temps. Après un éventuel échange de données, cette connexion se termine normalement mais avant que $CNX\_C'_1$ n'atteigne S. Un peu plus tard, cette connexion est réincarnée, c'est à dire C est à l'origine d'une nouvelle connexion possédant les mêmes caractéristiques de source et de destination, tant du point de vue des adresses IP que des numéros de ports ; seuls diffèrent les numéros de séquence initiaux. Pour ce faire, C émet un segment de demande de connexion, CNX_C_2, en provenance du numéro de port $port\_C$ et à destination du port de numéro $port\_S$ de S ; il porte le numéro de séquence initial seq_C_2, strictement postérieur à seq_C_1. Normalement, la réponse de S à cette demande devrait être un paquet d'acquittement de demande de connexion, ACK_S_2, portant l'acquittement $ack\_S_2 = seq\_C_2 + 1$. Toujours à cause de problèmes de congestion dans le réseau, le

segment $CNX\_C'_1$ précédemment généré parvient à S avant CNX_C_2. S ne répond donc pas à C par le segment ACK_S_2 mais par un segment $ACK\_S'_1$ portant un acquittement $ack\_S'_1$, avec $ack\_S'_1 = ack\_S_1$. Malgré cette égalité, il faut souligner que le segment $ACK\_S'_1$ n'est, a priori, pas identique à ACK_S_1 car la demande de connexion qu'il porte en retour ne possède pas nécessairement le même numéro de séquence initial que celui porté par ACK_S_1. Puisque $ack\_S'_1 \neq ack\_S_2$, C s'aperçoit que S ne répond pas à la bonne demande de connexion et que les numéros de séquences sont manifestement incohérents. En fonctionnement robuste, C doit envoyer à S un segment RST_C de demande de réinitialisation de connexion, avec le numéro de séquence $ack\_S1 = seq\_C_1 + 1$ attendu par S afin que ce dernier l'accepte sans problème. Conformément au rôle de la commande `RST` de TCP, S doit alors retourner à l'état d'attente passive (`LISTEN`). C peut alors réitérer sa demande de connexion en utilisant encore le numéro de séquence initial seq_C_2. Il faut remarquer que C ne doit surtout pas utiliser un nouveau numéro de séquence initial car si le paquet CNX_C_2 était reçu par S, le scénario précédent risquerait de se reproduire. Ce comportement est illustré par la FIG.55.

A posteriori, il est important de remarquer que le prétexte (faute) présidant au scéna-

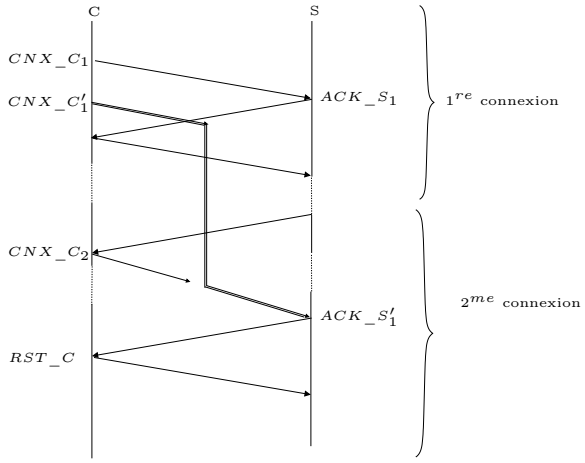


FIG. 55: Exemple de scénario provoquant un changement d'état (initialisation)

rio tel qu'il est décrit ci-dessus est une congestion du réseau entraînant des retards dans l'acheminement des paquets. Si cette hypothèse (faute), accompagnée de l'absence de garantie de la conservation de la séquence des datagrammes par IP, était la seule à engendrer des incohérences de numéros de séquences, elle pourrait probablement être introduite, de manière plus ou moins simple, dans le graphe de contrôle de TCP. Toutefois, il est possible d'imaginer bon nombre de scénarii voisins du précédent en faisant varier le moment précis des réémissions, le retard pris par chaque paquet, la durée de certains échanges, etc. Il est même possible d'ajouter à cette liste d'hypothèses le fonctionnement erratique d'éléments d'interconnexion, perdant ou dupliquant des paquets, ainsi que la génération malicieuse de paquets par des entités mal intentionnées.

Le système ignore l'aléa et ne change pas son état de fonctionnement. Il est quelquefois possible d'assurer certaines caractéristiques de la robustesse d'un système en ignorant certains événements. Ainsi, dans le cas du protocole TCP, la réception de segments dupliqués ne doit entraîner aucun changement d'état par rapport à l'IOLTS décrivant son fonctionnement simplifié. Par exemple, lorsqu'un module TCP se trouve dans l'état `LISTEN` suite à une mise en état d'attente passive, la réception d'un segment de demande d'ouverture de connexion entraîne le passage à l'état `SYN-RCVD`. Si le même segment arrive de nouveau avant que `SYN-RCVD` n'ait été quitté, il doit être simplement ignoré. De la même manière que dans le paragraphe précédent, il faut constater que cette technique fait aussi partie de celles qui sont utilisées en tolérance aux fautes mais que son utilisation est, encore une fois, différente. En effet, il peut aussi être difficile de faire un inventaire complet des états devant ainsi être rebouclés et des conditions associées.

Méthode TRACOR

Spécification augmentée. Pour pouvoir vérifier le comportement du protocole en présence d'aléas, nous avons procédé une augmentation de la spécification nominale donnée dans la Fig.54. Cette augmentation est fondée sur les aléas suivants :

1. **Entrées invalides**. Elles sont représentées par les classes d'équivalence suivantes : `?close-failed`, `?SYN-failed`, `?SEND-failed`, `?SYN,ACK-failed`, `?ACK-of-SYN-failed`, `FIN-failed` et `?ACK-of-FIN-failed`. Ces classes représentent toutes les faute susceptibles d'atteindre les entrées valides : `?close`, `?SYN`, `?SEND`, `?SYN,ACK`, `?ACK-of-SYN`, `?FIN` et `?ACK-of-FIN`. Dans ce cas, on considère que le comportement acceptable du protocole, en présence de ces classes d'aléas, consiste à ajouter des boucles, étiquetées par les entrées invalides précédentes, dans tout état de réception, c'est à dire le protocole ignore la réception d'un message déformé.
2. **Entrées inopportunes**. On suppose que dans chaque état de l'IOLTS de la spécification, la réception de `?SYN` ou `?Close` provoque une réinitialisation du protocole. En revanche, toute autre entrée inopportune sera ignorée. Ceci est modélisable par des boucles dans tout état de l'IOLTS associé.
3. **Sorties acceptables**. Dans cette étude de cas, on ne considère qu'une seule sortie acceptable : il s'agit d'un message `!RST` émis après la réception d'un message `?SYN` ou `?Close` inopportun.

Résultats. Afin de tester la robustesse du processus d'ouverture de connexion du protocole TCP, nous avons développé deux objectifs de test de robustesse (voir ANNEXE B.1) :

1. **RTP1** : Ouverture d'une connexion passive en présence d'aléas.
2. **RTP2** : Ouverture d'une connexion active en présence d'aléas.

Á l'aide de l'outil RTCG, nous avons calculé la spécification augmentée associée à la spécification nominale présentée dans la Fig.54 (22 états et 357 transitions). Nous rappelons

que les cas de test de robustesse sont générés aléatoirement à partir du graphe réduit du test de robustesse. La FIG.56 (à gauche) présente un exemple d'un RTC généré par notre outil.

Méthode TRACON

Afin de générer des cas de test de robustesse visant à vérifier certaines fonctionnalités ou propriétés en présence d'aléas contrôlables et non représentables, nous avons fait l'hypothèse de robustesse suivante : une implémentation robuste devrait émettre une demande de réinitialisation `!RST` après chaque occurrence d'aléas. La spécification semi-augmentée (voir ANNEXE B.2), construite à partir de l'hypothèse de robustesse précédente, est obtenue par l'ajout d'une transition `!RST` partant de tout état de l'IOLTS de la FIG.54 vers l'état initial. La FIG.56 (à droite) présente un exemple d'un CRTC généré par notre outil.

```
tescase RTC() runs on TCP-IUT
{timer ReponseTimer := 100E-3 ;
  Tester.send(close);
  Tester.send(passive-open);
  Tester.send(ACK-of-FIN-failed);
  Tester.send(SEND);
  Tester.send(SYN,ACK);
  ReponseTimer.start
  alt
    { [] ReponseTimer.timeout
          {  setverdict(fail);
             stop
          }
      [] Tester.receive(SYN);
          {  setverdict(pass);
             ReponseTimer.stop
             Tester.send(SYN);
             Tester.send(SYN,ACK);
             Tester.send(SYN,ACK);
             ReponseTimer.start
             alt
               {  [] ReponseTimer.timeout
                       {  setverdict(fail);
                          stop
                       }
                  [] Tester.receive(ACK-of-SYN);
                      {  setverdict(pass);
                         ReponseTimer.stop}
                  [else] { setverdict(fail);
                           stop
                         }
               }
      [else] { setverdict(fail);
               stop
             }
     }}
control
  {
     execute (TCP-Tester());
  }
}
```

```
tescase CRTC-TCP runs on TCP-IUT {timer ReponseTimer := 100E-3 ;
  Tester.send(close);
  Tester.send(passive-open);
  Tester.send(FIN);
  Tester.send(SEND);
  Tester.send(FIN);
  ReponseTimer.start
  alt
    { [] ReponseTimer.timeout
          {  setverdict(fail);
             stop
          }
      [] Tester.receive(SYN);
          {  setverdict(pass);
             ReponseTimer.stop
             Tester.send(SYN);
             Tester.send(Start);    // démarrage d'aléa.
             Tester.send(SYN,ACK);
             Tester.send(SYN);
             Tester.send(Stop);     // arrêt d'aléa.
             ReponseTimer.start
             alt
              {  [] ReponseTimer.timeout
                       {  setverdict(fail);
                          stop
                       }
                  [] Tester.receive(ACK-of-SYN);
                      {  setverdict(pass);
                         ReponseTimer.stop }
                  [else] { setverdict(fail);
                           stop
                             }
             }
      [else] { setverdict(fail);
               stop
              }}}
  control
  {
     execute (CRTC-TCP());
  }
}
```

FIG. 56: Exemple de RTC et CRTC générés par l'outil RTCG

Remarque

Dans un premier temps, les objectifs de test de robustesse utilisés peuvent être cycliques ce qui permet de générer un nombre infini des cas de test de robustesse. En conséquence, l'outil RTCG permet de générer que des cas de test de robustesse aléatoires (selon la

demande de l'utilisateur).

7.4 Conclusion

Nous venons de présenter une mise en œuvre de notre approche suivie par une étude de cas sur les protocoles SSL Hanshake et SSL.

Tout d'abord, nous avons abordé les possibilités et limites concernant la réutilisation de certains outils du test de conformité dans le cadre du test de robustesse. Ensuite, nous avons présenté les bases algorithmiques de notre outil RTCG. Cette outil permet d'automatiser les méthodes TRACOR et TRACON présentées dans le chapitre 5.

RTCG fournit deux interfaces pour automatiser la méthode TRACOR.
La première permet d'assister l'augmentation de la spécification d'un système à partir d'un graphe d'aléas et, optionnellement, d'un autre graphe d'entrées inopportunes.
La deuxième interface permet, grâce à une stratégie de coloriage, de générer des cas de test de robustesse.

RTCG fournit aussi une troisième interface pour automatiser la méthode TRACON. Dans ce cas, RTCG construit une spécification semi-augmentée à partir de l'automate suspendu de la spécification nominale et d'un graphe de sorties acceptables. Ensuite, il calcule le produit synchrone, le graphe réduit du test de robustesse, et dérive un cas de test de robustesse non contrôlable. Enfin, RTCG procède à une synchronisation du cas de test avec un contrôleur d'aléas. Le résultat est un cas de test de robustesse contrôlable.

Dans les trois interfaces, les spécifications sont décrites en SDL ou en DOT , et les cas de test de robustesse sont formatés en TTCN-3, XML ou DOT.

Enfin, nous avons présenté une étude de cas sur les protocoles SSL handshake et TCP afin de montrer l'intérêt pratique de notre approche.

Chapitre 8

Conclusion et perspectives

L'objectif principal de cette thèse était de définir des nouvelles approches basées sur les modèles formels pour automatiser le test de robustesse. Pour ce faire, nous sommes partis d'une situation de référence, le test de conformité basé sur les modèles formels ne prenant pas en compte les aléas, et avons étudié les écarts induits par la considération d'aléas. Le modèle utilisé à travers ce document est le système de transitions étiquetées à entrée-sortie (IOLTS). Nos principales contributions sont résumées ci-dessous :

Définition du test de robustesse. Pour écarter toute contradiction issue de définitions données par l'IEEE [88] ou l'AS 23 [33], nous avons décidé de considérer le test de robustesse comme un sur-ensemble du test de conformité. Par conséquent, le test de robustesse que nous avons défini ne devrait s'appliquer qu'à des systèmes conformes à leurs spécifications nominales.

Modélisation du comportement acceptable en présence d'aléa. Notre méthodologie considère comme un aléa : tout événement absent de la spécification nominale (existante) du système. Les aléas peuvent être classés par rapport aux frontières du système [33]. Selon ce point de vue, nous distinguons des aléas internes, externes ou au-delà des frontières du système. Notre classification d'aléas dépend de deux points : 1) la contrôlabilité et l'observabilité par le testeur, et 2) la représentabilité dans un modèle formel. Selon ce point de vue, nous distinguons deux classes principales d'aléas. Les aléas contrôlables et représentables, ainsi que les aléas contrôlables et non représentables (observables ou non). Dans le domaine des protocoles, certains aléas contrôlables peuvent être et représentés par des entrées inopportunes, entrées invalides ou sorties inattendues.

Afin d'éviter les problèmes liés à l'oracle, les concepteurs du système doivent spécifiés le comportment acceptable en présence d'un ensemble d'aléas (ou d'influence d'aléas) définis par les testeurs de robustesse. Pour ce faire, nous avons proposé d'utiliser les meta-graphes afin de minimizer la taille du modèle fourni.

Pour formaliser le test de robustesse, nous avons proposé deux méthodes formelles, chacune s'occupe d'une classe d'aléas.

Méthode TRACOR. Cette méthode est focalisée sur la formalisation du test de robustesse en présence d'aléas observables, contrôlables et représentables à travers des entrées invalides, entrées inopportunes et sorties acceptables. Dans cette méthode, une spécification augmentée est construite à partir de la spécification nominale du système, un graphe d'aléas et un graphe d'entrées inopportunes. La spécification augmentée est utilisée comme un modèle de référence pour la génération des cas de test de robustesse.

Méthode TRACON. Cette méthode est focalisée sur la formalisation du test de robustesse en présence d'aléas observables, contrôlables non représentables. Contrairement à la méthode TRACOR, on ne procède à aucune augmentation au niveau des entrées et, seules les sorties acceptables qui peuvent être observées en présence d'aléas sont prises en compte. Par conséquent, nous avons proposé de construire une spécification semi-augmentée. Celle-ci sera utilisée comme une référence pour la génération des cas de test de robustesse. De plus, les alternances démarrage/arrêt d'un aléa et l'application d'entrées est un aspect important qui doit être pris en compte durant la génération et l'exécution des tests de robustesse. En conséquence, les cas de test de robustesse dérivés à partir de la spécification semi-augmentée doivent être composés avec un contrôleur d'aléas afin d'obtenir des cas de test de robustesse contrôlables.

Génération des tests de robustesse. Afin de mesurer la robustesse d'une implémentation sous test vis-à-vis de la spécification augmentée (ou la spécification semi-augmentée), nous avons proposé une relation binaire appelée ***Robust***. Cette relation étend la relation ***ioco*** de Tretmans [190, 191]. ***Robust*** est centrée sur la partie augmentée de la spécification et élimine par conséquent toutes les traces nominales.

Partant de la spécification augmentée et de la relation ***Robust***, nous avons adapté l'approche de génération développée par l'IRISA et Verimag [63, 109, 107]. En plus du produit synchrone, notre nouvelle approche est basée sur un coloriage spécial de la spécification augmentée, un algorithme de réduction très simple (parcours en arrière) et un algorithme de sélection à la volée privilégiant les traces augmentées.

Pour ce qui concerne la génération de cas de test de robustesse en présence d'aléas contrôlables et non représentables, nous procédons à une deuxième composition avec un contrôleur d'aléas (une entité logicielle ou matérielle permettant d'activer ou désactiver l'exécution d'aléas). Le résultat est le graphe des cas de test de robustesse contrôlables CRTCG. Enfin, un cas de test de robustesse contrôlable CRTC est généré à partir de CRTCG.

La construction de la spécification augmentée et la spécification semi-augmentée préserve toute les traces de la spécification nominale, y compris les traces de blocage. Cette

particularité permet de réutiliser ces deux modèles comme références pour le test de conformité. Seule la relation de conformité utilisée doit éliminer les traces augmentées.

Mise en œuvre et études de cas. Nous avons implémenté les fondements théoriques de notre approche dans l'outil RTCG. Cet outil fournit deux interfaces pour automatiser la méthode TRACOR.
La première permet d'assister l'augmentation de la spécification d'un système à partir d'un graphe d'aléas et, optionnellement, d'un autre graphe d'entrées inopportunes.
La deuxième interface permet, grâce à une stratégie de coloriage, de générer des cas de test de robustesse.

RTCG fournit aussi une troisième interface pour automatiser la méthode TRACON. Tout d'abord, RTCG construit une spécification semi-augmentée à partir de l'automate suspendu de la spécification nominale et d'un graphe de sorties acceptables. Ensuite, il calcule le produit synchrone, le graphe réduit du test de robustesse, et dérive un cas de test de robustesse non contrôlable. Enfin, RTCG procède à une synchronisation du cas de test avec un contrôleur d'aléas. Le résultat est un cas de test de robustesse contrôlable.

Dans RTCG, les spécifications sont décrites en SDL ou en DOT , et les cas de test de robustesse sont formatés en TTCN-3, XML ou DOT.

Enfin, les protocoles SSL handshake et TCP sont utilisés comme des exemples concrets afin d'évaluer notre approche. A travers cette évaluation, nous avons montré d'abord quelques exemples sur les aléas prévisibles dans le domaine des protocoles. Ensuite, nous avons donné quelques résultats démonstratifs concernant l'application de l'outil RTCG.

Nouvelles perspectives

Pour conclure cette thèse, nous présentons certaines perspectives de recherche et améliorations envisageables dans le domaine du test de robustesse.

Application sur les systèmes temps-réel et embarqués

A court terme, il serait nécessaire d'adapter notre approche aux systèmes temps-réel modélisables à travers des formalismes complexes (contrôles, horloges, données, etc). Dans ce contexte, la considération du temps et/ou des données change complètement la notion d'aléas (le temps de réponse) et probablement la relation de robustesse. Plus précisément, nous adopterons le modèle des machines IOSTS (Input/Output Symbolic Transition System). Ce modèle est basé sur la sémantique des IOLTSs, mais étendu par des constantes symboliques , des variables, des paramètres de communication, des gardes et des affectations.

Une autre perspective consiste à adapter notre approche aux systèmes embarqués qui

commencent à trouver, de plus en plus, leur place dans les nouvelles technologies et dans le domaine d'usage public.

Modélisation Uniforme du test

Nous constatons que les approches formelles appliquées aux différents types de test sont très similaires dans leurs ingrédients de base. Les points de différenciation possibles peuvent être résumés en :

- Les spécifications sont légèrement différentes selon le degré de détail considéré (prise en compte ou non des aléas),
- Les objectifs de test se diffèrent seulement dans leur sémantique (état et/ou transition visité ou non, couverture, etc),
- Les algorithmes de générations diffèrent, à leur tour, dans la façons utilisée pour parcourir la totalité ou une partie de la spécification (parcours en avant ou en arrière), ainsi que dans les critères adoptés pour choisir les états et/ou les transitions à visiter (couleurs, priorité, etc),
- Les relations utilisées pour tester l'IUT.

Partant de contraintes précédentes, il s'avère possible de se baser sur une seule spécification de base susceptible de subir à quelques changements ou transformations pour bien répondre aux différents aspects visés par chaque type de test. Selon le type du test, on choisira la relation du test équivalente, puis on appliquera l'algorithme de génération approprié pour obtenir les cas des tests visés.

Bibliographie

[1] B. Alcalde, A. Cavalli, D. Chen, D. Khuu and D. Lee, "Network protocol system passive testing for fault management : a backward checking approach", *Proceeding of FORTE/PSTV'2004*, Madrid, Spain, LNCS 3235, September 2004.

[2] B. Algayres, Y. Lejeune and F. Hugonnet (Verilog), "GOAL : Observing SDL behaviors with ObjectGeode", *7th SDL Forum*, Oslo, Norway, 26-29 September 1995.

[3] R. Anido, "Guaranteeing Full Fault Coverage for UIO-Based Testing Methods", *International Workshop on Protocol Test Systems (IWPTS)*, Evry, 4-6 September, 1995.

[4] R. Anido, A. Cavalli, T. Macavei, L. Paula Lima, M. Clatin and M. Phalippou, "Engendrer des tests pour un vrai protocole grâce à des techniques éprouvées de vérification", *in Proceeding of CFIP'96 (Cinquième Colloque Francophone sur L'Ingénierie des Protocoles*), ENSIAS, Rabat, (Maroc) 14-17 octobre 1996.

[5] L. Apvrille, J.-P. Courtiat, C. Lohr and P. de Saqui-Sannes, "TURTLE : A Real-Time UML Profile Supported by a Formal Validation Toolkit", *IEEE Trans. Software Eng.* 30(7) : 473-487 (2004).

[6] N. Arakawa, M. Phalippou, N. Risser and T. Soneoka, "Combination of conformance and interoperability testing", *Formal Description Techniques*, V(C-10) M. Diaz and R. Gruz (Eds.) Elsevier Science Publishers, 1993.

[7] J. Arlat, J.-C. Fabre, M. Rodriguez and F. Salles, "Dependability of COTS Microkernel-Based Systems", *IEEE Transactions on Computers*, Volume 51(2), pp.138-163, Fevrier, 2002.

[8] A. Arnold. "Systèmes de transitions finis et sémantique des processus communicants", *Masson* 1992.

[9] A. Arnold, "Finite Transition Systems : Semantics of Communicating Systems", *International Series in Computer Science*. Prentice-Hall International, Englewood Cliffs, NJ, 1994

[10] A.AVIZIENIS, J.C.LAPRIE, B.RANDELL, C.LANDWEHR. "Basic concepts and taxonomy of dependable and secure computing", *IEEE Transactions on Dependable and Secure Computing*, Vol.1, N°1, pp.11-33, 2004

[11] S. Barbin, L. Tanguy and C. Viho, "Towards a formal framework for interoperability testing", *Proceedings FORTE01*, Korea, August 28-31, 2001.

[12] F. Belina and D. Hogrefe, "The CCITT-Specification and Desciption Language SDL", *Computer Networks and ISDN Systems*, Vol. 16, 1989.

[13] F. Belina, D. Hogrefe and A. Sarma, "SDL with Applications from Protocol Specification", *Prentice Hall International*, 1991.

[14] B. Beizer, "Software testing techniques", *2nd Edition*, New York, Van Nostrand Reinhold, 1990.

[15] A. Belinfante, J. Feenstra, R.-G. de Vries, J. Tretmans, N. Goga, L. Feijs, S. Mauw and L. Heerink, "Formal test automation, a simple experiment", *in Proceedings of 12th Int. Workshop on Testing of Communicating Systems*, pp.179-196, 1999.

[16] J.-A. Bergstra and J.-W. Klop, "Process Algebra for Synchronous Communication", *Information and Computation*, n°60, pp.109-137, 1984.

[17] I. Berrada and P. Felix, "TGSE : Un outil générique pour le test", *In proceedings CFIP'2005 (Colloque Francophone sur l'Ingénérie des Protocoles)*, Hermès-Lavoisier, pp 67-84. 29 Mars 2005.

[18] M.-v.-d. Bijl, A. Rensink and J. Tretmans, "Compositional testing with ioco". *in Proceedings of the 3rd International Workshop on Formal Approaches to Testing of Software, FATES2003*, Vol.2931, pp 86-100, Montreal, Canada, 2003.

[19] T. Bolognesi and E. Briskma, "Introduction to the ISO Specification Language LOTOS". *Computer Networks and ISDN Systems*, Vol.14(1), pp.25-29, janvier 1988.

[20] D. Bosnacki and D. Dams, "Discrete-Time Promela and Spin", *In Lecture Notes in Computer Science*, LNCS 1486, pp.307. 1998.

[21] M. Bozga, J.-C. Fernandez, L. Ghirvu, S. Graf, J.-P. Krimm and L. Mounier, "IF : A validation environment for timed asynchronous systems", *In Computer Aided Verification*, pp.543-547, 2000.

[22] M. Bozga, J-C. Fernandez, L. Ghirvu, S. Graf, J-P. Krimm, L. Mounier and J. Sifakis, "IF : An intermediate representation for SDL and its applications", *in SDL Forum 1999*, pp.423-440, 1999.

[23] M. Bozga, J.-C. Fernandez, L. Ghirvu, S. Graf, J.-P. Krimm and L. Mounier, "IF : An Intermediate Representation and Validation Environment for Timed Asynchronous Systems". *World Congress on Formal Methods*, pp.307-327, 1999.

[24] M. Bozga, S. Graf, I. Ober and J. Sifakis, "The IF Toolset", *In Marco Bernardo and Flavio Corradini. Formal Methods for the Design of Real-Time Systems, International School on Formal Methods for the Design of Computer, Communication and Software Systems, SFM-RT 2004*, Springer, Lecture Notes in Computer Science, Vol.3185, pp.237-267, 2004.

[25] J. Bradley and N. Davies, "Analysis of The SSL Protocol", *Technical Report CSTR-95-021, Communications Research Group, Department of Computer Science, University of Bristol*, July 1995.

[26] . E. Brinksma, G. Scollo, and C. Steenbergen, "LOTOS Specifications, their Implementations, and their Tests", *in Proceedings of the sixth international workshop on protocol specification, testing and verification, ed. Behcet Sarikaya*, pp.349-360, North-Holland, Amsterdam, 1987.

[27] E. Brinksma, "On the design of extended LOTOS - A specification language for open distributed system", *PhD thesis, Department of Computer Science, University of Twente*, 1988.

[28] E. Brinksma, "A theory for the derivation of test", S. Aggarwal and Sabnani, eds, *In Protocol Specification, Testing and Verification VIII*, pp.63-74, North-Holland, 1988.

[29] E. Brinksma and J. Tretmans, "Testing transition systems : an annotated Bibliography", *In the proceeding of Modelling and Verification of Parallel Processes (MOVEP2k)*, LNCS 2067, pp.187-195. Springer-Verlag, 2000.

[30] L. Bromstrup and D. Hogrefe, "TESDL : Experience with Generating Test Cases from SDL Specifications", *Fourth Proc. SDL Forum*, pp.267-279, 1989.

[31] W.Chung and P.Amer, "Improved on UIO Sequence Generation and Partial UIO Sequences", *in Protocol Specification, Testing, and Verification*, XII, Lake Buena Vista, Florida, USA, R.J. Linn and M.U. Uyar, Eds, North-Holland, June 1992.

[32] J. Carreira, H. Madeira and J.-G. Silva, "Xception : A Technique for the Experimental Evaluation of Dependability". *in Modern Computers. IEEE Trans. Software Eng.* Vol.24(2), pp.125-136 (1998).

[33] R. Castanet and H. Weaselynk, "Techniques avancées de test des systèmes complexes : test de robustesse ", *Action spécifique N°23 du CNRS*, Laboratoires participants : IRISA, LAAS, LaBRI, LRI, Verimag. 11/2002-11/2003.

[34] S.-T. Chanson and J. Zhu, "A Unified Approach to Protocol Test Sequence Generation", *in Proc. IEEE INFOCOM*, pp.106-114. San Francisco, March 1993.

[35] S.-T. Chanson and J. Zhu, "Automatic Protocol Test Suite Derivation", *in Proc. IEEE INFOCOM*, pp.792-799, 1994.

[36] J. Chailloux and P. Couronné, "AGEL : un atelier de développemnt de systèmes réactifs synchrones", *Génie Logiciel et Système Expert*, Vol.25, 1991.

[37] G.-J. Carrette, "CRASHME : Random Input Testing", http ://people.delphi.com/gjc/crashme.html, 1996.

[38] CCITT Recommendation X.292, "OSI conformance testing methodology and framework for protocol Recommendations for CCITT applications : The Tree and Tabular Combined Notation (TTCN)", 1992.

[39] P. Chevochot and I. Puaut, "Experimental Evaluation of Fail-Silent Behavior of Distributed Real-Time Run-Time Support Built from COTS Components". *In Proc. Int. Conference on Dependable Systems and Networks (DSN-2001)*, pp.304-313, IEEE CS Press, Juillet 2001.

[40] D. Clark, T. Jeron, V. Rusu and E. Zinovieva, "STG : A Symbolic Test Generation Tool", *Lecture Notes in Computer Science*, LNCS 2280, pp.470, 2002.

[41] D. Clarke, T. Jéron, V. Rusu, and E. Zinovieva, "Automated test and oracle generation for Smart-Card applications", *In International Conference on Research in Smart Cards (e-Smart'01)*, LNCS 2140, pp.58-70, 2001.

[42] T.-S. Chow, "Testing software design by finite state machine", *IEEE Transactions on Software Engineering*, Vol.SE-4(3), May, 1978.

[43] M. Clatin, R. Groz, M. Phalippou and R. Thummel, "Two Approaches Linking a Test Generation Tool with Verification Techniques", *International Workshop on Protocol Test Systems (IWPTS)*, Evry, 4-6 September, 1995.

[44] L. Clarke, "A System to Generate Test data and Symbolically Execute Programs", *IEEE Transactions on Software Engineering*, Vol.2(3), Septembre 1976.

[45] J. Crow and B. Di Vito, "Formalizing space shuttle software requirements : Four case studies", *ACM Transactions on Software Engineering and Methodology*, vol.7(3), pp.296-332, 1998.

[46] C. Csallner and Y. Smaragdakis, "JCrasher : an automatic robustness terter for Java", *In College of Computing, Georgia Institute of Technologiy*, Atlanta-USA, 2004.

[47] J.-D. Day and H. Zimmermann, "The OSI Reference Model", *In Proceedings of the IEEE*, Vol.71, pp.1334-1340, 1983.

[48] R.-A. DeMillo, W.M. McCraken, R.-J. Martin and J.-F. Passafiume, "Software Testing and Evaluation", *Menlo Park, CA, USA, The Benjamin/Cummings Publishing Company*, Inc., 1987.

[49] R. De Nicola and M. C. B Hennessy, "Testing equivalences for processes", *Theoretical Computer Science*, Vol.34, pp.83-133, 1984

[50] R. De Nicola and R. Segala, "A process algebraic view of I/O automata", *In Theoretical Computer Science*, Vol.138, pp.391-423, March 1995.

[51] R. de Simone. "Higher-level synchronising devices in MEIJE-SCCS". *Theoretical Computer Science*, Vol.37, pp.245-267, 1985.

[52] T. Dierks and C. Allen, "The TLS Protocol Version 1.0", *Internet Engineering Task Force (IETF), RFC 2246*, 1999.

[53] The DOT Language, http ://www.graphviz.org/doc/info/lang.html.

[54] D. Drusinsky and D. Harel, "Using StateCharts for Hardware Description and Synthesis", *in IEEE Transactions on Computer-Aided Dessign*, 1989.

[55] H. Ehrig and B. Mahr, "Fundamentals of Algebraic Specification 1, Equations and Initial Semantics", *In W. Brauer, B. Rozenberg, A. Salomaa, eds., EATCS , Monographs on Theoretical Computer Science*, Springer Verlag, 1985.

[56] M. H. V. Emden. "Structured Inspections of Code", *In J. of Software Testing, Verification and Reliability*, Vol.2, pp.133-153, 1992.

[57] ETSI ETR 130 (ETSI Technical Report), "Methods for Testing and Specification (MTS) : Interoperability and Conformance Testing. A Classification Scheme", April 1994.

[58] ETSI European Standard (ES) 201 873 (2002/2003) : "The Testing and Test Control Notation", Version 3 (TTCN-3), Part 1 : TTCN-3 Core Language, Part 2 : Tabular Presentation Format for TTCN-3 (TFT), Part 3 : Graphical Presentation Format for TTCN-3 (GFT), Part 4 : Operational Semantics, Part 5 : The TTCN-3 Runtime Interface (TRI), Part 6 : The TTCN-3 Control Interfaces (TCI). *European Telecommunications Standards Institute (ETSI)*, Sophia-Antipolis (France).

[59] M. Rodriguez, F. Salles, J.-C Fabre and J. Arlat, "MAFALDA : Microkernel Assessment by Fault Injection and Design Aid.", *in Proceedings Dependable Computing (EDCC-3), Third European Dependable Computing Conference*, Prague, Czech Republic, pp.143-160, September 15-17, 1999,

[60] M.-E. Fagan, "Design and Code Inspections to Reduce Errors in Program Development", *IBM Systems Journal*, VOL.15(3), pp.219-248, 1976.

[61] J.-C. Fernandez, H. Gravel, L. Mounier, A. Rasse, C. Rodriguez and J. Sifakis, "A tool Box for verification of lotos programs", *in Proceedings of the 14th International Conference on Software Engineering, May 11-15, 1992, Melbourne, Australia. ACM Press, 1992*, pp.246-259, Melbourne, Australia, May 1992.

[62] J.-C. Fernandez, H. Garavel, A. Kerbrat, L. Mounier, R. Mateescu and M. Sighireanu. "CADP - A Protocol Validation and Verification Toolbox", *In proceedings of the 8th International Conference on Computer Aided Verification CAV '96*. Springer-Verlag 3-540-61474-5. Vol.1102. pp 437-440. 1996.

[63] J.-C. Fernandez, C. Jard, T. Jéron and C. Viho, "An experiment in automatic generation of test suites for protocols with verification technology". *Science of Computer Programming*. Vol.29(1-2) : pp.123-146, 1997.

[64] J.-C. Fernandez, L. Mounier, C. Pachon. "A Model-Based Approach for Robustness Testing", *in Testing of Communication Systems (TESTCOM'05)*, ifip, LNCS, 3502, pp.333-348, june 2005.

[65] J.-E. Forrester et B.-P. Miller, "An Empirical Study of the Robustness of Windows NT Application Using Random Testing", *In Proc. 4th USENIX Windoxws Systems Symposium*, Seatle-USA, Août 2000.

[66] H. Fouchal, A. Rollet, and A. Tarhini, "Robustness of Composed Timed Systems". *In 31st Annual Conference on Current Trends in Theory and Practice of Informatics (SOFSEM'05)*, January 22 - 28, 2005, Liptovsky Jan, Slovak Republic, Europe, Vol.3381 of Lecture Notes in Computer Science, pp.155-164. Springer-Verlag, jan 2005.

[67] S. Fujiwara, G.-V. Bochmann, F. Khendek, M. Amalou and A. Ghedamsi, "Test Selection Based on Finite State Models, *Publication départementale 716*, DIRO, Université de Montréal, Février 1990.

[68] J. Gadre, C. Rohrer, C. Summers and S. Stmington, "A COS study of OSI interoperability", *Computer Standards and Interfaces*, Vol.9(3), pp.217-237, 1990.

[69] H. Garavel, "Compilation et Vérification de Programmes LOTOS", *Thèse de Doctorat*, Université Josef Fourier, Grenoble, 1989.

[70] A. Gill, "Introduction to the theory of finite-state machines", *Mc Graw-Hill*, New York - USA, 1962.

[71] G. Gonenc, "A Method for the design of fault detection experiment", *IEEE transactions on Computers*, Vol.C-19, pp.551-558, 1970.

[72] J. Grabowski, D. Hogrefe and R. Nahm, "A Method for the Generation of Test Cases Based on SDL and MSCs", *Institut fur Informatik*, Universitat Bern, April 1993.

[73]] J. Grabowski, "SDL and MSC Based Test Case Generation : An Overall View of the SaMsTaG Method", *Institut fur Informatik*, Universitat Bern, May 1994.

[74] A. Ghosh, M. Schmid and V.Shah, "Testing the robustness of Windows NT Software", *In Pandroceedings of the 9th Int. Symp. on Software Reliability Engineering (ISSRE'98)*, Los Alamitos, CA, IEEE, Computer Society Press, pp.231-235, Novembre 1998.

[75] D. Harel, "Statecharts : A Visual Formalism for Complex Systems", *Journal of Science of Computer Programming*, Vol.8, pp.231-274, 1987.

[76] D. Harel, H. Lachover, A. Naamad, A. Pnueli, M. Politi, R. Sherman, A.-S. Trauring and M. Trakhtenbrot, "STATEMATE : A Working Environnement for the Developpement of Complex Reactive Systems", *IEEE Trans. on Software Engineering*, Vol.SE-16(4), pp.403-414, 1990.

[77] D. Harel and A. Naamad, "The STATEMATE Semantics of Statecharts", *The Journal of ACM Transaction on Software Engineering and Methodologies*, Vol.5(4), pp.293-333, October 1996.

[78] A. Helmy and D. Estrin and S. Gupta, "Fault-oriented Test Generation for Multicast Routing Protocol Design", *in proceedings of Formal Description Techniques and Protocol Specification, Testing and Verification (FORTE'98)*, pp.93-109, 1998.

[79] M. Hennessy, "Algebraic theory of processes", *MIT Press*, Cambridge, MA, 1988.

[80] T.-A. Henzinger, Z. Manna and A. Pnueli, "What good are digital clocks ?", *ICALP'92*, LNCS 623, 1992.

[81] K. Hickman, "The SSL Protocol", *Netscape Communications Corp.*, 501 E. Middlefield Rd., Mountain View, CA 94043, Feb. 1995, http ://home.netscape.com/newsref/std/SSL.html.

[82] W.-M. Ho, F. Pennaneac'h, and N. Plouzeau, "UMLAUT : A framework for weaving uml-based aspect-oriented designs", *In Technology of object-oriented languages and systems (TOOLS Europe)*, Vol.33, pp.324-334. IEEE Computer Society, June 2000.

[83] G.-J. Holzmann, "Design and Validation of Computer Protocols", *Prentice Hall*, 1991.

[84] C.-A.-R. Hoare, "Communicating Sequential Processes", *Commun. ACM*, Vol.21(8), pp.666-677. ACM Press. New York, NY, USA. 1978.

[85] W.-E. Howden, "Weak Mutation Testing and Completeness of Test Sets", *IEEE Trans. on Software Testing*, Vol.SE-8, pp.371-379, 1982.

[86] D. Hogrefe, "Conformance testing based on formal methods", *In Proc of Formal Description Techniques*, ed. J. Quemada, A. Fernandez, 1990.

[87] C.-M. Huang, Y.-C. Lin and M.-Y. Jang, "Executable Data Flow and Control Flow Protocol Test Sequence Generation for EFSM-Specified Protocol", *International Workshop on Protocol Test Systems (IWPTS)*, Evry, 4-6 September, 1995.

[88] IEEE Standard Glossary of Software Engineering Terminology 610.12-1990. *In IEEE Standards Software Engineering, 1999 Edition*, Vol.1 : Customer and Terminology Standards. IEEE Press, 1999.

[89] ISO 9646-1, International Organization for Standardization, "Conformance testing methodology and framework - part 1 : general concepts", 1994.

[90] ISO 9646-2, International Organization for Standardization, "Conformance testing methodology and framework - part 2 : abstract test suite specification", 1994.

[91] ISO 9646-3, International Organization for Standardization, "Conformance Testing Methodology And Framework - Part 3 : The Tree And Tabular Combined Notation (TTCN)", 1992.

[92] ISO 9646-4, International Organization For Standardization, "Conformance testing methodology and framework - part 4 : test realization", 1994.

[93] ISO 9646-5, International Organization For Standardization, "Conformance testing methodology and framework - part 5 : requirements on test laboratories and clients for the conformance assessment process", 1994.

[94] ISO 9646-6, International Organization For Standardization, "Conformance testing methodology and framework - part 6 : protocol profile test specification", 1994.

[95] ISO 7498 :Systèmes de traitement de l'information - interconnexion de systèmes ouverts - modèle de référence de base, 1984.

[96] ISO/TC97/SC21/WG 1 - FDT/C. Lotos, a formal description technique based on the temporal ordering. Technical Report DIS 8807, ISO, 1987.

[97] ISO/IEC. "ESTELLE : A formal description technique based on an extended state transition model". *International Standard no 9074, International Organization for Standardization , Information Processing Systems ,Open Systems Interconnection*, Genève, September 1988.

[98] ISO/CEI 8824-1, Notation de syntaxe abstraite numéro 1 (ASN.1) Spécification de la notation de base Amendement 1 : Prise en charge des règles de codage XML étendues (EXTENDED-XER), 2003

[99] ISO/IEC JTC1 DTR-10000. Information Technology - Framework and Classification of International Standard Protocol. 1994.

[100] ITU-T X.290 Series, "Conformance Testing Methodology and Framework", 1994.

[101] ITU-T SG 10/Q.8 ISO/IEC JTCI/SC21 WG7. Information Retrieval. Transfer and Management for OSI ; Framwork : "Formal Methods in Conformance Testing". Committee Graft CD 13245-1, ITU-T proposed recommandation Z 500. ISO-ITU-T, 1996.

[102] ITU-T SG 10 :Q.8 ISO :IEC JTCI/SC21 WG7. Information Retrieval, Transfer and Management of OSI ; Framework : Formal Methods in Conformance Testing. Committee Draft CD 13245-1, ITU-T proposed recommendation Z 500. ISO-ITU-T, 1996.

[103] "ITEX User Manual", Telelogic AB, 1995. [Jack 75] M. Jackson, "Principles of Program Design" Academic Press, New York, N.Y., 1995.

[104] ITU-T, "SDL combined with ASN.1 modules (SDL/ASN.1)", *ITU-T Recommendation no Z.100, International Telecommunication Union*, November 1994, Genève.

[105] ITU-T, "Specification and Description Language (SDL)", *ITU-T Recommendation no Z.105, International Telecommunication Union*, 1999, Genève.

[106] ITU-T Recommendations Z.140-142 (2002) : "The Testing and Test Control Notation", Version 3 (TTCN-3), Rec. Z.140 : TTCN-3 Core Language, Rec. Z.141 : Tabular Presentation Format for TTCN-3 (TFT), Rec. Z.142 : Graphical Presentation Format for TTCN-3 (GFT). ITU-T, Geneva (Switzerland).

[107] C. Jard and T. Jéron, "TGV : theory, principles and algorithms", *J. of Software Tools for Technology Transfer (STTT)*, Vol.7(4), pp.297-315, 2005.

[108] T. Jéron and P. Morel, "Test generation derived from model-checking", *CAV'99*, Trento, Italy, LNCS 633, pp. 108-122. Springer-Verlag, July 1999.

[109] T. Jéron, "TGV : théorie, principes et algorithmes", *Techniques et Sciences Informatiques*, numéro spécial Test de Logiciels, Vol.(21), 2002.

[110] S. Kang and M. Kim, "Test sequence generation for adaptive interoperability testing", *In Proceedings of Protocol Test Systems VIII*, pp.187-200, 1995.

[111] S. Kang and M. Kim, "Interoperability test suite derivation for symmetric communication protocols", *Proceedings of FORTE/PSTV'97*, pp.57-72, November 18-21, 1997.

[112] S. Kang, J. Shin and M. Kim, "Interoperability test suite derivation for communication protocols", *Computer Network*, Vol.32, pp.347-364, 2000.

[113] W.-L. Kao, R. K. Iyer and D. Tang, "FINE : A Fault Injection and Monitoring Environnement for tracing the UNIX System Behavior under Faults", *IEEE Transaction on Software Engineering*, Vol.19(11), pp.1105-1118, Novembre 1993.

[114] W.-L. Kao, and R.-K. Iyer, "DEFINE : A Distributed Fault Injection and Monitoring Environnement ", *in Fault Tolerant Parallel and Distributed Systems*, pp.252-259, IEEE CS Press, Los Alamitos, CA, USA, 1995.

[115] S. Kim, S. Cho, and S. Hong, "Schedulabilityaware mapping of real-time object-oriented models to multithreaded implementations", *Real-Time Computing Systems and Applications*, pp.7-14, December 2000.

[116] O. Koné and R. Castanet, "Deriving coordinated testers for interoperability", *Protocol Test System VI*, Pau, France, 28-30 September 1993.

[117] O. Koné and R. Castanet, "Test generation for interworking systems", *Computer Communications*, Vol.23(7), pp.642-652, 2000.

[118] P.-J. Koopman, J. Sung, C. Dingman, D.-P. Siewiorek and T. Marz, "Comparing Operating Systems using Robustness Benchmarks", *In Proc. 16th Int. Symp. on Reliable Distributed Systems (SRDS-16)*, pp.72-79, IEEE Computer Society Press, 1997.

[119] P.-J. Koopman and J. Deval, "Comparing the Robustness of POSIX Operating Systems", *In Proc. 29th Int. Symp. on Fault-Tolerant Computing (FTCS-29)*, pp.30-37, IEEE CS Press, 1999.

[120] M. Krichen and S. Tripakis, "Black-box conformance testing for real-time systems", *In SPIN 2004*, Spring-Verlag Heidelberg, pp.109-126, 2004.

[121] R.-J, Laakso, M, Takanen and A. Kaksonen, "PROTOS - systematic approach to eliminate software vulnerabilities", *Invited presentation at Microsoft Research*, Seattle, USA. May Vol.6, 2002.

[122] C. Lafontaine, Y. Ledru and P.-Y. Schobbens, "An Experiment in Formal Software Develeppemnt : Using the B Theorem Prover on a VDM case Study", *Com. of the ACM* , Vol.34(5), pp. 62-71 + 87, 1991.

[123] J.-C. Laprie, J. Arlat, J.-P. Blanquart, A. Costes, Y. Crouzet Y. Deswarte, J.-C. Fabre, H. Guillermain, M. Kaâniche, K. Kanoun, C. Mazet, D. Powell, C. Rabéjac and P. Thévenod, "Guide de la sûreté de fonctionnement", *Cépaduès-Edition*, 1995.

[124] G. Leduc, "On the role of Implementation Relations in the Design of Distributed Systems using LOTOS", *Dessertation d'agrégation*, Université de Liège, Belgique, 1990.

[125] G. Leduc, "A framework based on implementation relations for implementing LOTOS specifications", *In Computer Networks ans ISDN Systems*, Vol.25, pp.23-41. North Holand 1992.

[126] D. Lee and M. Yannakakis, "Testing finite state machines : state identification and verification", *IEEE Trans. Computer*, Vol.43(3), pp.306-320, 1994.

[127] D. Lee and M. Yannakakis, "Principles and methods of testing finite state machines - A survey", *Proceedings of the IEEE*, Vol.84(8), pp.1090-1123, August, 1996.

[128] D. Lee, A.-N. Netravali, K.-K. Sabnani, B. Sugla and A. John, "Passive testing and applications to network management", *ICNP'97 International Conference on Network Protocols*, Atlanta, Georgia, October 28-31, 1997.

[129] P.-H.-J. van Eijk, C.-A. Vissers, and M. Diaz, "The formal description technique LOTOS", Elsevier Science Publishers B.V., 1989.

[130] G. Luo, R. Dssouli, G.-V. Bochmann, P. Venkataram and A. Ghedamsi, "Generating synchronizable test sequences based on finite state machine with distributed ports", *IFIP TC6/WG6.1 Sixth International Workshop on Protocol Test Systems*, Pau, France 28-30 September 1993.

[131] Ga. Luo, G.-V. Bochmann and A. Petrenko, "Test Selection Based on Communicating Nondeterministic Finite-State Machines Using a Generalized WpMethod", *IEEE Transactions on Software Engineering*, Vol.20(2), February 1994.

[132] N.-A. Lynch, "I/O automata : a model for discrete event systems", *In Proc. of 22nd Conf. on Information Sciences and Systems*, pp.29-38, Princeton, NJ, USA, March 1988.

[133] H. Madeira, D. Costa and M. Vieira, "On the Emulation of Software Faults by Software Fault Injection", *In Proc. Int. Conference on Dependable Systems and Networks (DSN-2000)*, New York-USA, pp.417-426, IEEE CS, Press, Juin 2000.

[134] E. Marsden and J.-C. Fabre, "Failure Mode Analysis of CORBA Service Implementations", *In Proc. IFIP/ACM International Conference on Distributed Systems Platforms (Middleware'2001)*, Heidelberg-Germany, Novembre, 2001.

[135] G. McGraw and J. Viega. "Why COTS Software increase security risks", *In Proc. of the 1st Int. Workshop on Testing Distributed Component-Based Systems*, May 1999.

[136] R.-E Milner, "A calculus of communicating systems", *Lecture Notes Computer Science 92*, 1980.

[137] B.-P. Miller, D. Koski, C.-P. Lee, V. Maganty, R. Murthy, A. Natarjan and J. Steidl, "FUZZ Revisited : A Re-examination of the Reliability of UNIX Utilities and Services", *Research Report, N°CS-TR-95-1268*. University of Wisconsin, USA, Avril 1995.

[138] R.-E. Miller and S. Paul, "Generating Conformance Test Sequences for Combined Control and Data Flow of Communication Protocols", *Proceeding of Protocol Specifications, Testing and Verification (PSTV' 92)*, Florida, USA, June 1992.

[139] R. Milner, "Communication and Concurrency", International Series in Computer Science, Prentice Hall, NY, 1989.

[140] L.-J. Morell. "A Theory of Fault-Based Testing", *IEEE Trans. on Software Engineering*, Vol.16(8), pp.844-857, 1990.

[141] K. Myungchul, K. Gyuhyeong and D.-C. Yoon, "Interoperability testing methodology and guidelines", *Digital Audio-Visual Council, System Integration TC*, DAVIC/TC/SYS/96/06/006, 1996.

[142] S. Naito and M. Tsunoyama, "Fault detection for sequential machines by transition tours", *In Proc. IEE Fault Tolerant Comput. Symp., IEEE Computer Press*, pp.238-243, 1981.

[143] ObjectGEODE Documentation, ObjectGEODE 4.0, SDL Simulator, Chapter 5 : An approach to simulation (CS Verilog).

[144] N. Okazaki, M.-R. Park, K. Takahashi and N. Shiratori, "A new test sequence generation method for interoperability testing", *Protocol Test System VII*, Tokyo, Japan, 8-10 November 1994.

[145] C. Pâchon. "Une Approche basée sur les modèles pour le Test de Robustesse". *Phd thesis*, Université JOSEPH FOURIER, Grenoble, 2005.

[146] J. Pan, P.-J. Koopman, D.-P. Siewiorek, Y. Huang, R. Gruber and M.-L. Jiang, "Robustness Testing and Hardering of CORBA ORB Implementations", *In Proc. 2001 Int. Conference on Dependable Systems and Network (DSN'2001)*, Göteborg-Swiden, pp.141-150, IEEE Computer Society Press, juillet 2001.

[147] C.-A. Petri, "Communication with Automata", *Ph.D. Thesis*, Institute for Instrumental Mathematics, Bonn, Germany, 1962.

[148] C.-A. Petri, "Fundamentals of a theory of asynchronous information Flow". *Proc. of IFIP Congress 62*, pp. 386-390 ,Amsterdam, North Holland Publ. Comp., 1963.

[149] 7 M. Phalippou, "Raltions d'implanatations et hypothèses de test sur les automates à entrées et sorties", *PhD thesis, Université de Bordeaux 1*, 1994.

[150] J. Postel, "Transmission Control Protocol", IETF, RFC793, September 1981.

[151] M.-O. Rabin, "Decidability of second order theories and automata on infinite trees", *Transactions of the AMS*, Vol(141), pp.1-35, 1969.

[152] O. Rafiq and R. Castanet, "From conformance testing to interoperability testing", *Pro. 3rd IWPTS*, October-November, Washington D.C, 1990.

[153] T. Ramalingom, A. Das and K. Thulasiraman, "A Unified Test Case Generation Method for the EFSM Model Using Context Independent Unique Sequences", *International Workshop on Protocol Test Systems (IWPTS)*, Evry, 4-6 September, 1995.

[154] S. Rapps, E.-J. Weyuker, "Selecting Software Test Data Using Data Flow Information", *IEEE Trans. on Software Engigeering*, Vol.SE-11(4), pp.367-375, 1985.

[155] D. Rayner, "OSI Conformance Testing", *Networks and ISDN Systems*, Vol.14, pp.79-98, 1987.

[156] O. Rayner, "Future directions for protocol testing, learning the lessons from the past", *IFIP TC6, 10th International Workshop on Testing of Communicating Systems (IWTCS'97)*, Cheju Island, Korea, September 8-10 , 1997.

[157] A. Rollet, "Test de Robustesse des Systèmes temps-réel", Phd thesis, Université de Reims Chapmagne-Ardenne, 2004.

[158] A. Rollet, "Testing robustness of real-time embedded systems". *In Proceedings of Workshop On Testing Real-Time and Embedded Systems (WTRTES), Satellite Workshop of Formal Methods (FM)* 2003 Symposium, 2003, Pisa, Italy,

[159] M. Roper, "Software Testing : A Selected Annotated Bibliography" *J. of Software Testing, Verification and Reliability*, Vol.2, pp.133-132, 1992.

[160] J.-P. Roth, W.-G. Bouricius and P.-R. Schneider, "Programmed Algorithms to Compute Tests to Detect and Distinguish Between Failures in Logic Circuits", *IEEE. Trans. on Electronic Computer*, Vol(EC-16), pp.567-579, 1967.

[161] J. Rushby and F.-V. Henke, "Formal Verification of Algorithms for Critical Systems", *Proc SigSoft'91 Conf. on Software for Critical Systems*, Software Engineering Notes, Vol.16, pp.1-15, New Orleans, LA, USA, ACM Press, 1991.

[162] R. Ruys and R. Langerak, "Validation of the Bosch'mobile communication network architecture with SPIN". *In participant Proceedings of the third SPIN, Workshop SPIN'97*, Enschede, The Netherlands, 1997.

[163] F. Saad-khorchef, I. Berrada, A. Rollet and R. Castanet, "Automated Robustness Testing for Reactive Systems : Application to Communicating Protocols", *In proceedings The 6th International Workshop on Innovative Internet Community Systems (I2CS 2006)*, LNCS-Springer edit., June 26-28, 2006 Neuchâtel.

[164] F. Saad-khorchef et R. Castanet, "Génération des tests de robustesse", *Nouvelles Technologies de Répartition (NOTERE 2006)*,Hermès edits., 06-09 juin 2006, Toulouse, France.

[165] F. Saad-khorchef et X. Delord, "Une méthode pour le test de robustesse adaptée aux protocoles de communication", *proceedings de 11ème Colloque Francophone sur l'Ingénierie des Protocoles (CFIP'05)* (papier court), Hermès edits., 29 mars- 01 avril ,Bordeaux.

[166] F. Saad-khorchef, I. Berrada, A. Rollet et R. Castanet, "Cadre formel pour le test de robustesse : Application au protocole SSL", *proceedings de 12ème Colloque Francophone sur l'Ingénierie des Protocoles (CFIP 2006)*. Hermès edits., 30 october - 3 novembre 2006. Tozeur, Tunisie.

[167] F. Saad-khorchef, "Robustness Testing for Reactive Systems", *The Sixth European Dependable Computing Conference (EDCC 2006)*, October 18-20 Cuimbra, Portugal.

[168] K. Sabnani and A. Dahbura. "A protocol test generation procedure". *Computer Networks and ISDN Systems*, Vol.15, pp.285-297, 1988.

[169] K.-V. Sam, "Formalisation de l'interfonctionnement dans les réseaux de télécommunication et définition d'une théorie de test pour systèmes concurrents", *Phd thesis*, Université Paris 7, June 2001.

[170] F. Salles, J.-C Fabre , M. Rodriguez and J. Arlat, "Microkernels and Fault Containement Wrappers", *In Proc. 29th IEEE Int. Symposium on Fault Tolerant Computing (FTCS-29)*, Madison-USA, pp.22-29, IEEE Computer Society Press, Juin 1999.

[171] F. Salles, J.-C Fabre , M. Rodriguez and J. Arlat, "MAFALDA : Microkernel Assessment by Fault Injection and Design", *In Proceedings Dependable Computing - EDCC-3 (Third European Dependable Computing Conference)*. Vol.1667, pp.143-160. September 1999. Proceedings Editors : J. Hlavicka, E. Maehle, A. Pataricza (Eds.). Prague, Czech Republic. 1999.

[172] B. Sarikaya, G.-V. Bochmann and E. Cerny, "A Test Methodology for Protocol Testing", *IEEE Transactions on Software Engineering*, Vol.13(5), pp.518-531, May 1987.

[173] B. Sarikaya, "Principles of Protocol Engineering and Conformance Testing", *Ellis Horwood Series in Computer Communications and Networking*, 1993.

[174] Z. Segall, D. Vrsalovic, D. Siewiorek, D. Yaskin, J. Kownacki, J. Barton, D. Rancey, A. Robinson, and T. Lin, "FIAT-Fault Injection Based Automated Testing Environment" *Proc. 18th Int'l Symp. Fault-Tolerant Computing (FTCS-18)*, pp.102-107, 1988

[175] S. Seol, M. Kim and S.-T. Chanson, "Interoperability test generation for communication protocols based on multiple stimuli principle", *Proceedings of TestCom'02)*, Berlin, Germany, March 19-22 , 2002.

[176] S. Seol, M. Kim, S. Kang and J. Ryu, "Fully automated interoperability test suite derivation for communication protocols", *Computer Networks* Vol.43, pp.735-759, December 2003.

[177] C. Shelton, P. Koopman and K. De Vale, "Robustness Testing of the Microsoft Win32 API", *in Proc. Int. Conference on Dependable Systems and Networks (DSN'2000)*, New York, pp.261-270, IEEE Computer Society Press, juin 2000.

[178] J. Shin and S. Kang, "Interoperability test suite derivation for the ATM/B-ISDN signaling protocol", *Proceedings of IWTCS'98*, Tomsk, Russia, August31-September 2, 1998.

[179] M. Schmitt II, J. Grabowski, D. Hogrefe and B. Koch , "Autolink : Putting SDL-based test generation into practice", *in proceedings of 2nd Workshop on Industrial-Strength Formal Specification Techniques (WIFT '98)*, pp.114, October 20-23, Boca Raton, FL, USA, 1998.

[180] I. Schieferdecker, B. Stepien and A. Rennoch, "PerfTTCN, a TTCN language extension for performing testing", *in Testing of Communicating Systems*. Vol.10, Chapman and Hall, pp.21-36, 1997.

[181] P. Schnoebelen, B. Bérard, M. Bidoit, F. Laroussinie and A. Petit, "Systems and software verification - Model-Checking techniques and tools", *Springer-Verlag*, 2001.

[182] "SDT User Manual", Telelogic Malmo" AB, 1995.

[183] Y.-N. Shen, F. Lombardi and A.-T. Dahbura, "Protocol Conformance Testing Using Multiple UIO Sequences", *IEEE Transactions on Communications*, Vol.40(8), August 1992.

[184] D. Estrin, S. Gupta and A. Helmy. "STRESS : Systematic Testing of Robustness by Evaluation of Synthesized Scenarios", http ://netweb.usc.edu/stress, 1998.

[185] B. Suh, C. Fineman, and Z. Segall, "FAUST - Fault Injection Based Automated Software Testing," Proc. 1991 Systems Design Synthesis Technology Workshop, Silver Spring, MD, 10-13 Sep 1991, NSWC. 25

[186] Systems engineering, software development and testing automation. Telelogic. http ://www.telelogic.com/corp/products/tau/index.cfm.

[187] P. Thévenod and H. Waeselynk, "An Investigation of Statistical Software Testing", *Journal of Software Testing Verification and Reliability*, Vol.1(2), pp.5-25, 1991.

[188] A. Touag, A. Rouger, "Génération de tests à partir de spécifications basées sur une construction partielle de l'arbre d'accessibilité", *proceedings de CFIP'96*, Rabat, Maroc, pp.515-529, 1996.

[189] J. Tretmans, "A formal approach to conformance testing", *PhD thesis*, University of Twente, Enschede, The Netherlands, 1992.

[190] J. Tretmans, "Test Generation with Inputs, Outputs and Repetitive Quiescence", *Software - Concepts and Tools* Vol.17(3), pp.103-120, 1996.

[191] J. Tretmans, "Test Generation with Inputs, Outputs and Quiescence", *In T. Margaria and B. Steffen, editors, Second International Worshop on Tools and Algorithms for the Construction and Analysis of Systems, TACAS'96, Passau, Germany*. Springer-Arlag, LNCS 1055, March 1996.

[192] J. Tretmans and E. Brinksma, "TorX : Automated model based testing", *In 1st European Conference on Model Driven Software Engineering*, pp. 31-43, Nuremberg, Germany, December 2003.

[193] T. Tsai, R.-K. Iyer and D. Jewittp "An Approach Towards Benchmarking of Fault-Tolerant Commercial Systems"p *In Proc. 26th Int. Symp. on Fault-Tolerant Computing (FTCS-26)*, Sendai, Japan, pp.314-323, IEEE Computer Society Press, juin 1996.

[194] T. Tsai and R.-K. Iyer, "Measuring Fault Tolerance with the FTAPE Fault Injection Tool," Proceedings Eighth International Conference. on Modeling Techniques and Tools for Computer Performance Evaluation, Heidelberg, Germany, Springer-Verlag, pp.26-40, Sept. 20–22 1995

[195] "Unified Modeling Language : Superstructure, Version 2.0". Object Managment Group (OMG), http ://www.omg.org/cgi-bin/doc ?formal/05-07-04.

[196] H. Ural, "Test Sequence Selection Based on Static Data Flow Analysis", *In Computer Communications*. Vol.10(5). pp.234-242. 1987.

[197] H. Ural and B. Yang, "A Test Sequence Selection Method for Protocol Testing", *In IEEE Transactions on Communication*, Vol.39(4), April 1991.

[198] Verilog SA. *TestComposer Reference Manual, 4.1 edition*, 1999.

[199] G.-S. Vermmeer and H. Blik, "Interoperability testing : basis of the accepting of communicating systems", *Protocol Test System VI*, Elsevier Science Publishers 1994.

[200] S.-T. Vuong, W.-W.-L. Chan and M.-R. Ito, "The UIOv Method for Protocol Test Sequence Generation", *In the 2-nd International Workshop Protocol Test System*, Berlin, Germany, Octobre 3-6 1989.

[201] S.-T. Vuong, K.-C. Ko, "A Novel Approach to Protocol Test Sequence Generation", *In Proc. of GlobalCOM'90*, pp.1880-1884, 1990.

[202] S.-T. Vuong and J. A. Curgus, "On test coverage metrics for communication protocols, *In Kroon J. Heijink R.J and Brinksma E. editors. Protocol Test System IV, Vol.C-3 of IFIP Transactions*, North-Holand, 1992.

[203] A. Wasserman, "On the Meaning of Discipline in Software Design and Development", *In Software Engineering Techniques*, Infotech State of the Art Report, 1977.

[204] E.-J. Weyuker, "Evaluating Software Complexity Measures". *The Computer J.*, Vol.25(4), pp. 465-470, 1982.

[205] C.-H. White, "System Reliability and Integrity", *In Infotech State of the Art Report*, Infotech, 1978.

[206] N. Yevtushenko, "Conformance methods and architectures", *Tarot Summer School*, June 28, Paris, France, 2005.

Annexe A

Étude de cas sur le protocole SSL handshake

A.1 Méthode TRACOR

Fichier de spécification nominale

```
PROCESS SSL-Hanshake (1,1);
START NEXTSTATE 1;
STATE 1;
 OUTPUT Client-Hello(no-sid);
 NEXTSTATE 2;
 OUTPUT  Client-Hello(sid);
 NEXTSTATE 3;
STATE 2;
 INPUT No-Certificate-Error;
 NEXTSTATE 4;
 INPUT ?Server-Hello(No-Hit);
 NEXTSTATE 5;
STATE 3;
 INPUT Server-Hello(No-Hit);
 NEXTSTATE 5;
 INPUT Server-Hello(Hit);
 NEXTSTATE 6;
STATE 4;
 INPUT Close-Connection;
 NEXTSTATE 1;
STATE 5;
 OUTPUT Bad-Certificate-Error;
 NEXTSATE 7;
 OUTPUT No-Cipher-Error;
 NEXTSTATE 7;
 OUTPUT Unsupported-Certificate-Type-Error;
 NEXTSTATE 7;
 OUTPUT Client-Master-Key;
 NEXTSTATE 6;
STATE 6;
 OUTPUT Client-Finished;
 NEXTSTATE 8;
 INPUT Server-Verify;
 NEXTSTATE 9;
STATE 7;
```

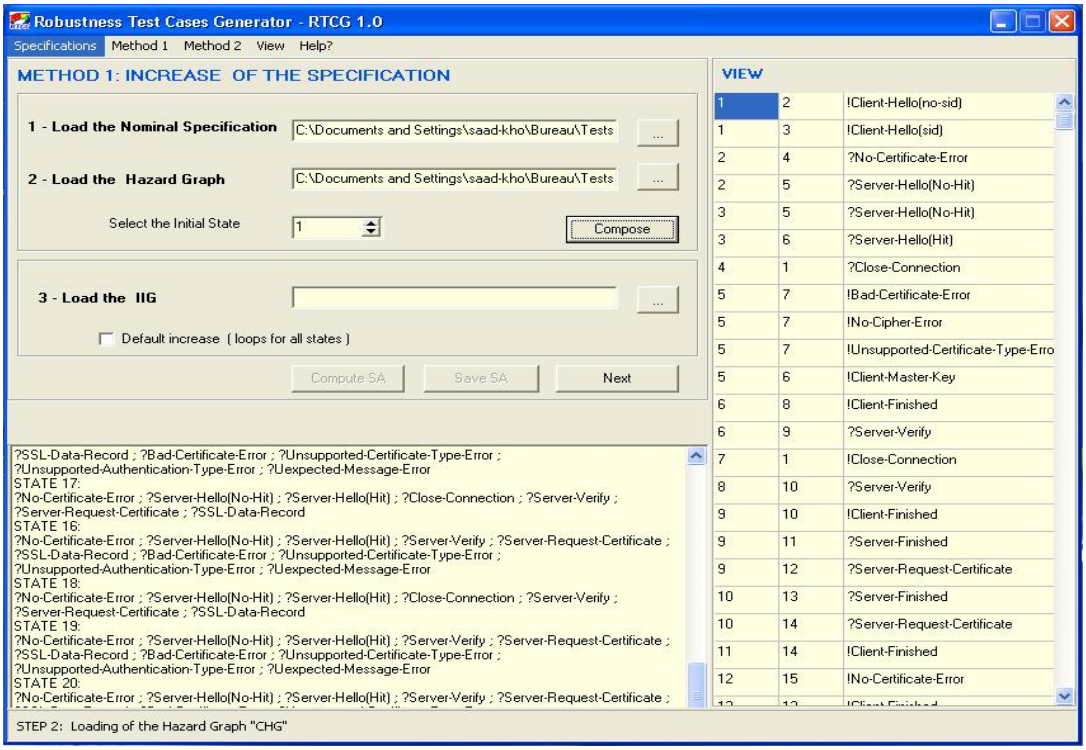
```
 OUTPUT Close-Connection;
 NEXTSTATE 1;
STATE 8;
 INPUT Server-Verify;
 NEXTSTATE 10;
STATE 9;
 OUTPUT Client-Finished;
 NEXTSTATE 10;
 INPUT Server-Finished;
 NEXTSTATE 11;
 INPUT Server-Request-Certificate;
 NEXTSTATE 12;
STATE 10;
 INPUT Server-Finished;
 NEXTSTATE 13;
 INPUT Server-Request-Certificate;
 NEXTSTATE 14;
STATE 11;
 OUTPUT Client-Finished;
 NEXTSTATE 14;
STATE 12;
 OUTPUT No-Certificate-Error;
 NEXTSTATE 15;
 OUTPUT  Client-Finished;
 NEXTSTATE 13;
 OUTPUT Client-Certificate;
 NEXTSTATE 17;
STATE 13;
 OUTPUT No-Certificate-Error;
 NEXTSTATE 16;
 OUTPUT Client-Certificate;
 NEXTSTATE 18;
STATE 14;
 INPUT SSL-Data-Record;
 NEXTSTATE 14;
 OUTPUT SSL-Data-Record;
 NEXTSTATE 14;
 INPUT Close-Connection;
 NEXTSTATE 1;
 OUTPUT Close-Connection;
 NEXTSTATE 1;
STATE 15;
 OUTPUT Client-Finished;
 NEXTSTATE 16;
 INPUT Server-Finished;
 NEXTSTATE 11;
 INPUT Close-Connection;
 NEXTSTATE 1;
STATE 16;
 INPUT Server-Finished;
 NEXTSTATE 14;
 INPUT Close-Connection;
 NEXTSTATE 1;
STATE 17;
 INPUT Server-Finished;
 NEXTSTATE 11;
 INPUT Bad-Certificate-Error;
 NEXTSTATE 19;
 INPUT Unsupported-Certificate-Type-Error;
 NEXTSTATE 19;
 OUTPUT Client-Finished;
 NEXTSTATE 18;
STATE 18;
 INPUT Bad-Certificate-Error;
 NEXTSTATE 20;
```

```
 INPUT Unsupported-Certificate-Type-Error;
 NEXTSTATE 20;
 INPUT Server-Finished;
 NEXTSTATE 14;
STATE 19;
 OUTPUT Client-Finished;
 NEXTSTATE 20;
 INPUT Close-Connection;
 NEXTSTATE 1;
 INPUT Server-Finished;
 NEXTSTATE 11;
STATE 20;
 INPUT Server-Finished;
 NEXTSTATE 14;
 INPUT Close-Connection;
 NEXTSTATE 1;
END PROCESS SSL-Hanshake;
```

Fichier de graphe d'aléas

```
digraph Hazard-Graph {
5 -> 7 [label = "?Unsupported-Authentication-Type-Error"];
5 -> 7 [label = "?Uexpected-Message-Error"];
17 -> 19 [label = "?Unsupported-Authentication-Type-Error"];
17 -> 19 [label = "?Uexpected-Message-Error"];
18 -> 20 [label = "?Unsupported-Authentication-Type-Error"];
18 -> 20 [label = "?Uexpected-Message-Error"];
}
```

Calcul d'entrées inopportunes



Fichier de spécification augmentée

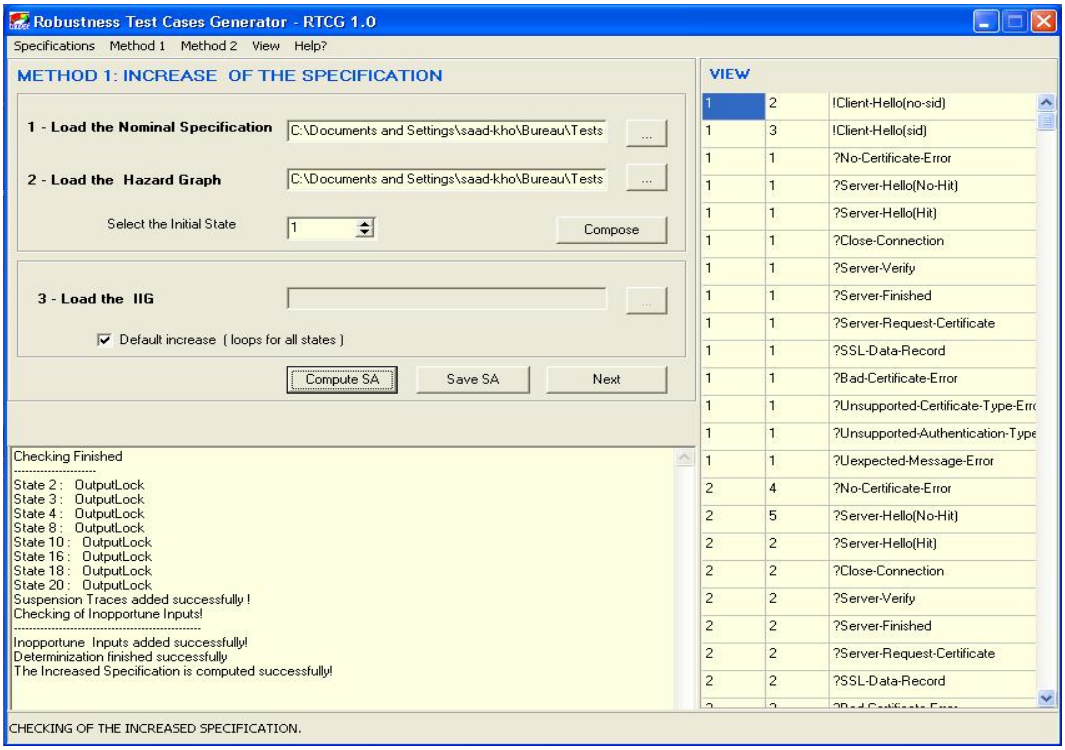
```
digraph SSl-SA {
1 -> 2 [label = "!Client-Hello(no-sid)"];
1 -> 3 [label = "!Client-Hello(sid)"];
2 -> 4 [label = "?No-Certificate-Error"];
2 -> 5 [label = "?Server-Hello(No-Hit)"];
3 -> 5 [label = "?Server-Hello(No-Hit)"];
3 -> 6 [label = "?Server-Hello(Hit)"];
4 -> 1 [label = "?Close-Connection"];
5 -> 7 [label = "!Bad-Certificate-Error"];
5 -> 7 [label = "!No-Cipher-Error"];
5 -> 7 [label = "!Unsupported-Certificate-Type-Error"];
5 -> 6 [label = "!Client-Master-Key"];
6 -> 8 [label = "!Client-Finished"];
6 -> 9 [label = "?Server-Verify"];
7 -> 1 [label = "!Close-Connection"];
8 -> 10 [label = "?Server-Verify"];
9 -> 10 [label = "!Client-Finished"];
9 -> 11 [label = "?Server-Finished"];
9 -> 12 [label = "?Server-Request-Certificate"];
10 -> 13 [label = "?Server-Finished"];
10 -> 14 [label = "?Server-Request-Certificate"];
11 -> 14 [label = "!Client-Finished"];
12 -> 15 [label = "!No-Certificate-Error"];
12 -> 13 [label = "!Client-Finished"];
12 -> 17 [label = "!Client-Certificate"];
13 -> 16 [label = "!No-Certificate-Error"];
13 -> 18 [label = "!Client-Certificate"]; 1
4 -> 14 [label = "?SSL-Data-Record"];
14 -> 14 [label = "!SSL-Data-Record"];
14 -> 1 [label = "?Close-Connection"];
14 -> 1 [label = "!Close-Connection"];
15 -> 16 [label = "!Client-Finished"];
15 -> 11 [label = "?Server-Finished"];
15 -> 1 [label = "?Close-Connection"];
16 -> 14 [label = "?Server-Finished"];
16 -> 1 [label = "?Close-Connection"];
17 -> 11 [label = "?Server-Finished"];
17 -> 19 [label = "?Bad-Certificate-Error"];
17 -> 19 [label = "?Unsupported-Certificate-Type-Error"];
17 -> 18 [label = "!Client-Finished"];
18 -> 20 [label = "?Bad-Certificate-Error"];
18 -> 20 [label = "?Unsupported-Certificate-Type-Error"];
18 -> 14 [label = "?Server-Finished"];
19 -> 20 [label = "!Client-Finished"];
19 -> 1 [label = "?Close-Connection"];
19 -> 11 [label = "?Server-Finished"];
20 -> 14 [label = "?Server-Finished"];
20 -> 1 [label = "?Close-Connection"];
5 -> 7 [label = "?Unsupported-Authentication-Type-Error"];
5 -> 7 [label = "?Uexpected-Message-Error"];
17 -> 19 [label = "?Unsupported-Authentication-Type-Error"];
17 -> 19 [label = "?Uexpected-Message-Error"];
18 -> 20 [label = "?Unsupported-Authentication-Type-Error"];
18 -> 20 [label = "?Uexpected-Message-Error"];
1 -> 1 [label = "?No-Certificate-Error"];
1 -> 1 [label = "?Server-Hello(No-Hit)"];
1 -> 1 [label = "?Server-Hello(Hit)"];
1 -> 1 [label = "?Close-Connection"];
1 -> 1 [label = "?Server-Verify"];
1 -> 1 [label = "?Server-Finished"];
1 -> 1 [label = "?Server-Request-Certificate"];
1 -> 1 [label = "?SSL-Data-Record"];
1 -> 1 [label = "?Bad-Certificate-Error"];
```

```
1 -> 1 [label = "?Unsupported-Certificate-Type-Error"];
1 -> 1 [label = "?Unsupported-Authentication-Type-Error"];
1 -> 1 [label = "?Uexpected-Message-Error"];
2 -> 2 [label = "?Server-Hello(Hit)"];
2 -> 2 [label = "?Close-Connection"];
2 -> 2 [label = "?Server-Verify"];
2 -> 2 [label = "?Server-Finished"];
2 -> 2 [label = "?Server-Request-Certificate"];
2 -> 2 [label = "?SSL-Data-Record"];
2 -> 2 [label = "?Bad-Certificate-Error"];
2 -> 2 [label = "?Unsupported-Certificate-Type-Error"];
2 -> 2 [label = "?Unsupported-Authentication-Type-Error"];
2 -> 2 [label = "?Uexpected-Message-Error"];
3 -> 3 [label = "?No-Certificate-Error"];
3 -> 3 [label = "?Close-Connection"];
3 -> 3 [label = "?Server-Verify"];
3 -> 3 [label = "?Server-Finished"];
3 -> 3 [label = "?Server-Request-Certificate"];
3 -> 3 [label = "?SSL-Data-Record"];
3 -> 3 [label = "?Bad-Certificate-Error"];
3 -> 3 [label = "?Unsupported-Certificate-Type-Error"];
3 -> 3 [label = "?Unsupported-Authentication-Type-Error"];
3 -> 3 [label = "?Uexpected-Message-Error"];
4 -> 4 [label = "?No-Certificate-Error"];
4 -> 4 [label = "?Server-Hello(No-Hit)"];
4 -> 4 [label = "?Server-Hello(Hit)"];
4 -> 4 [label = "?Server-Verify"];
4 -> 4 [label = "?Server-Finished"];
4 -> 4 [label = "?Server-Request-Certificate"];
4 -> 4 [label = "?SSL-Data-Record"];
4 -> 4 [label = "?Bad-Certificate-Error"];
4 -> 4 [label = "?Unsupported-Certificate-Type-Error"];
4 -> 4 [label = "?Unsupported-Authentication-Type-Error"];
4 -> 4 [label = "?Uexpected-Message-Error"];
5 -> 5 [label = "?No-Certificate-Error"];
5 -> 5 [label = "?Server-Hello(No-Hit)"];
5 -> 5 [label = "?Server-Hello(Hit)"];
5 -> 5 [label = "?Close-Connection"];
5 -> 5 [label = "?Server-Verify"];
5 -> 5 [label = "?Server-Finished"];
5 -> 5 [label = "?Server-Request-Certificate"];
5 -> 5 [label = "?SSL-Data-Record"];
5 -> 5 [label = "?Bad-Certificate-Error"];
5 -> 5 [label = "?Unsupported-Certificate-Type-Error"];
6 -> 6 [label = "?No-Certificate-Error"];
6 -> 6 [label = "?Server-Hello(No-Hit)"];
6 -> 6 [label = "?Server-Hello(Hit)"];
6 -> 6 [label = "?Close-Connection"];
6 -> 6 [label = "?Server-Finished"];
6 -> 6 [label = "?Server-Request-Certificate"];
6 -> 6 [label = "?SSL-Data-Record"];
6 -> 6 [label = "?Bad-Certificate-Error"];
6 -> 6 [label = "?Unsupported-Certificate-Type-Error"];
6 -> 6 [label = "?Unsupported-Authentication-Type-Error"];
6 -> 6 [label = "?Uexpected-Message-Error"];
7 -> 7 [label = "?No-Certificate-Error"];
7 -> 7 [label = "?Server-Hello(No-Hit)"];
7 -> 7 [label = "?Server-Hello(Hit)"];
7 -> 7 [label = "?Close-Connection"];
7 -> 7 [label = "?Server-Verify"];
7 -> 7 [label = "?Server-Finished"];
7 -> 7 [label = "?Server-Request-Certificate"];
7 -> 7 [label = "?SSL-Data-Record"];
7 -> 7 [label = "?Bad-Certificate-Error"];
7 -> 7 [label = "?Unsupported-Certificate-Type-Error"];
```

```
7 -> 7 [label = "?Unsupported-Authentication-Type-Error"];
7 -> 7 [label = "?Uexpected-Message-Error"];
8 -> 8 [label = "?No-Certificate-Error"];
8 -> 8 [label = "?Server-Hello(No-Hit)"];
8 -> 8 [label = "?Server-Hello(Hit)"];
8 -> 8 [label = "?Close-Connection"];
8 -> 8 [label = "?Server-Finished"];
8 -> 8 [label = "?Server-Request-Certificate"];
8 -> 8 [label = "?SSL-Data-Record"];
8 -> 8 [label = "?Bad-Certificate-Error"];
8 -> 8 [label = "?Unsupported-Certificate-Type-Error"];
8 -> 8 [label = "?Unsupported-Authentication-Type-Error"];
8 -> 8 [label = "?Uexpected-Message-Error"];
9 -> 9 [label = "?No-Certificate-Error"];
9 -> 9 [label = "?Server-Hello(No-Hit)"];
9 -> 9 [label = "?Server-Hello(Hit)"];
9 -> 9 [label = "?Close-Connection"];
9 -> 9 [label = "?Server-Verify"];
9 -> 9 [label = "?SSL-Data-Record"];
9 -> 9 [label = "?Bad-Certificate-Error"];
9 -> 9 [label = "?Unsupported-Certificate-Type-Error"];
9 -> 9 [label = "?Unsupported-Authentication-Type-Error"];
9 -> 9 [label = "?Uexpected-Message-Error"];
10 -> 10 [label = "?No-Certificate-Error"];
10 -> 10 [label = "?Server-Hello(No-Hit)"];
10 -> 10 [label = "?Server-Hello(Hit)"];
10 -> 10 [label = "?Close-Connection"];
10 -> 10 [label = "?Server-Verify"];
10 -> 10 [label = "?SSL-Data-Record"];
10 -> 10 [label = "?Bad-Certificate-Error"];
10 -> 10 [label = "?Unsupported-Certificate-Type-Error"];
10 -> 10 [label = "?Unsupported-Authentication-Type-Error"];
10 -> 10 [label = "?Uexpected-Message-Error"];
11 -> 11 [label = "?No-Certificate-Error"];
11 -> 11 [label = "?Server-Hello(No-Hit)"];
11 -> 11 [label = "?Server-Hello(Hit)"];
11 -> 11 [label = "?Close-Connection"];
11 -> 11 [label = "?Server-Verify"];
11 -> 11 [label = "?Server-Finished"];
11 -> 11 [label = "?Server-Request-Certificate"];
11 -> 11 [label = "?SSL-Data-Record"];
11 -> 11 [label = "?Bad-Certificate-Error"];
11 -> 11 [label = "?Unsupported-Certificate-Type-Error"];
11 -> 11 [label = "?Unsupported-Authentication-Type-Error"];
11 -> 11 [label = "?Uexpected-Message-Error"];
12 -> 12 [label = "?No-Certificate-Error"];
12 -> 12 [label = "?Server-Hello(No-Hit)"];
12 -> 12 [label = "?Server-Hello(Hit)"];
12 -> 12 [label = "?Close-Connection"];
12 -> 12 [label = "?Server-Verify"];
12 -> 12 [label = "?Server-Finished"];
12 -> 12 [label = "?Server-Request-Certificate"];
12 -> 12 [label = "?SSL-Data-Record"];
12 -> 12 [label = "?Bad-Certificate-Error"];
12 -> 12 [label = "?Unsupported-Certificate-Type-Error"];
12 -> 12 [label = "?Unsupported-Authentication-Type-Error"];
12 -> 12 [label = "?Uexpected-Message-Error"];
13 -> 13 [label = "?No-Certificate-Error"];
13 -> 13 [label = "?Server-Hello(No-Hit)"];
13 -> 13 [label = "?Server-Hello(Hit)"];
13 -> 13 [label = "?Close-Connection"];
13 -> 13 [label = "?Server-Verify"];
13 -> 13 [label = "?Server-Finished"];
13 -> 13 [label = "?Server-Request-Certificate"];
13 -> 13 [label = "?SSL-Data-Record"];
```

```
13 -> 13 [label = "?Bad-Certificate-Error"];
13 -> 13 [label = "?Unsupported-Certificate-Type-Error"];
13 -> 13 [label = "?Unsupported-Authentication-Type-Error"];
13 -> 13 [label = "?Uexpected-Message-Error"];
14 -> 14 [label = "?No-Certificate-Error"];
14 -> 14 [label = "?Server-Hello(No-Hit)"];
14 -> 14 [label = "?Server-Hello(Hit)"];
14 -> 14 [label = "?Server-Verify"];
14 -> 14 [label = "?Server-Finished"];
14 -> 14 [label = "?Server-Request-Certificate"];
14 -> 14 [label = "?Bad-Certificate-Error"];
14 -> 14 [label = "?Unsupported-Certificate-Type-Error"];
14 -> 14 [label = "?Unsupported-Authentication-Type-Error"];
14 -> 14 [label = "?Uexpected-Message-Error"];
15 -> 15 [label = "?No-Certificate-Error"];
15 -> 15 [label = "?Server-Hello(No-Hit)"];
15 -> 15 [label = "?Server-Hello(Hit)"];
15 -> 15 [label = "?Server-Verify"];
15 -> 15 [label = "?Server-Request-Certificate"];
15 -> 15 [label = "?SSL-Data-Record"];
15 -> 15 [label = "?Bad-Certificate-Error"]; 1
5 -> 15 [label = "?Unsupported-Certificate-Type-Error"];
15 -> 15 [label = "?Unsupported-Authentication-Type-Error"];
15 -> 15 [label = "?Uexpected-Message-Error"];
17 -> 17 [label = "?No-Certificate-Error"];
17 -> 17 [label = "?Server-Hello(No-Hit)"];
17 -> 17 [label = "?Server-Hello(Hit)"];
17 -> 17 [label = "?Close-Connection"];
17 -> 17 [label = "?Server-Verify"];
17 -> 17 [label = "?Server-Request-Certificate"];
17 -> 17 [label = "?SSL-Data-Record"];
16 -> 16 [label = "?No-Certificate-Error"];
16 -> 16 [label = "?Server-Hello(No-Hit)"];
16 -> 16 [label = "?Server-Hello(Hit)"];
16 -> 16 [label = "?Server-Verify"];
16 -> 16 [label = "?Server-Request-Certificate"];
16 -> 16 [label = "?SSL-Data-Record"];
16 -> 16 [label = "?Bad-Certificate-Error"];
16 -> 16 [label = "?Unsupported-Certificate-Type-Error"];
16 -> 16 [label = "?Unsupported-Authentication-Type-Error"];
16 -> 16 [label = "?Uexpected-Message-Error"];
18 -> 18 [label = "?No-Certificate-Error"];
18 -> 18 [label = "?Server-Hello(No-Hit)"];
18 -> 18 [label = "?Server-Hello(Hit)"];
18 -> 18 [label = "?Close-Connection"];
18 -> 18 [label = "?Server-Verify"];
18 -> 18 [label = "?Server-Request-Certificate"];
18 -> 18 [label = "?SSL-Data-Record"];
19 -> 19 [label = "?No-Certificate-Error"];
19 -> 19 [label = "?Server-Hello(No-Hit)"];
19 -> 19 [label = "?Server-Hello(Hit)"];
19 -> 19 [label = "?Server-Verify"];
19 -> 19 [label = "?Server-Request-Certificate"];
19 -> 19 [label = "?SSL-Data-Record"];
19 -> 19 [label = "?Bad-Certificate-Error"];
19 -> 19 [label = "?Unsupported-Certificate-Type-Error"];
19 -> 19 [label = "?Unsupported-Authentication-Type-Error"];
19 -> 19 [label = "?Uexpected-Message-Error"];
20 -> 20 [label = "?No-Certificate-Error"];
20 -> 20 [label = "?Server-Hello(No-Hit)"];
20 -> 20 [label = "?Server-Hello(Hit)"];
20 -> 20 [label = "?Server-Verify"];
20 -> 20 [label = "?Server-Request-Certificate"];
20 -> 20 [label = "?SSL-Data-Record"];
20 -> 20 [label = "?Bad-Certificate-Error"];
```

```
20 -> 20 [label = "?Unsupported-Certificate-Type-Error"];
20 -> 20 [label = "?Unsupported-Authentication-Type-Error"];
20 -> 20 [label = "?Uexpected-Message-Error"];
2 -> 2 [label = "!delta"];
3 -> 3 [label = "!delta"];
4 -> 4 [label = "!delta"];
8 -> 8 [label = "!delta"];
10 -> 10 [label = "!delta"];
16 -> 16 [label = "!delta"];
18 -> 18 [label = "!delta"];
20 -> 20 [label = "!delta"]; }
```



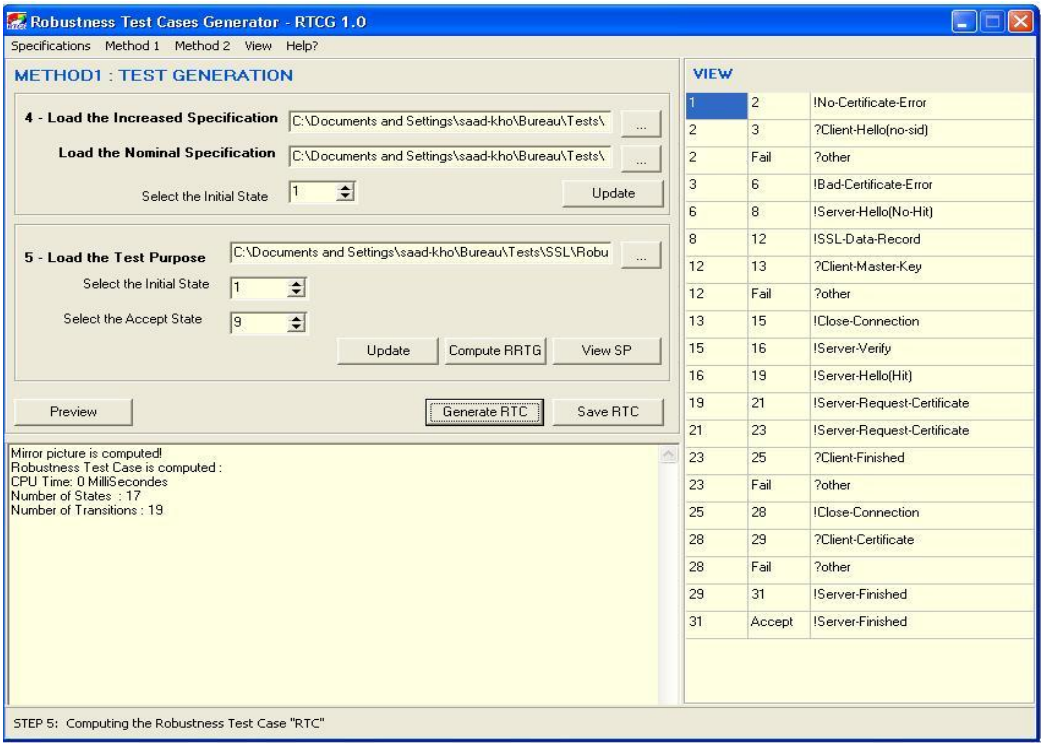
Objectifs de test de robustesse

```
digraph RTP1 {
1 -> 7 [label = "?other"];
7 -> 2 [label = "!Client-Hello(sid)"];
2 -> 8 [label = "?other"];
8 -> 3 [label = "?Server-Hello(Hit)"];
3 -> 9 [label = "?other"];
9 -> 4 [label = "!Client-Finished"];
4 -> 10 [label = "?other"];
4 -> 10 [label = "?Uexpected-Message-Error"];
10 -> 5 [label = "?other"];
11 -> 6 [label = "?Server-Finished"];
}
```

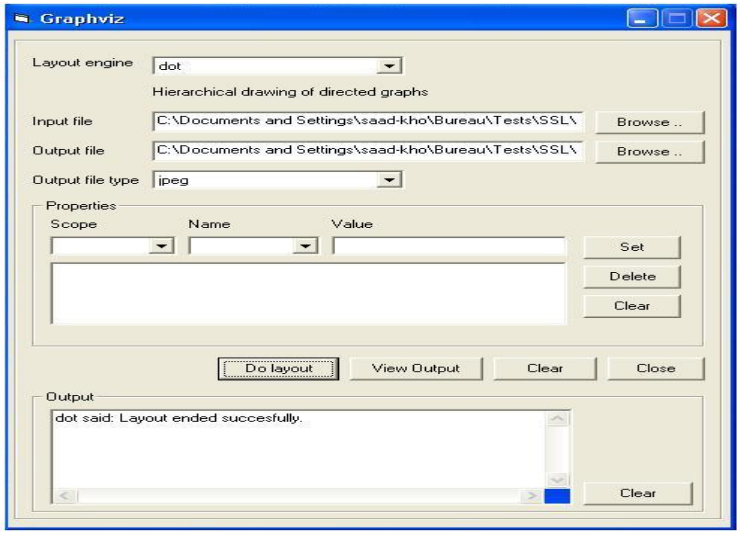
```
 digraph RTP2 {
1 -> 8 [label = "?other"];
8 -> 2 [label = "!Client-Hello(no-sid)"];
```

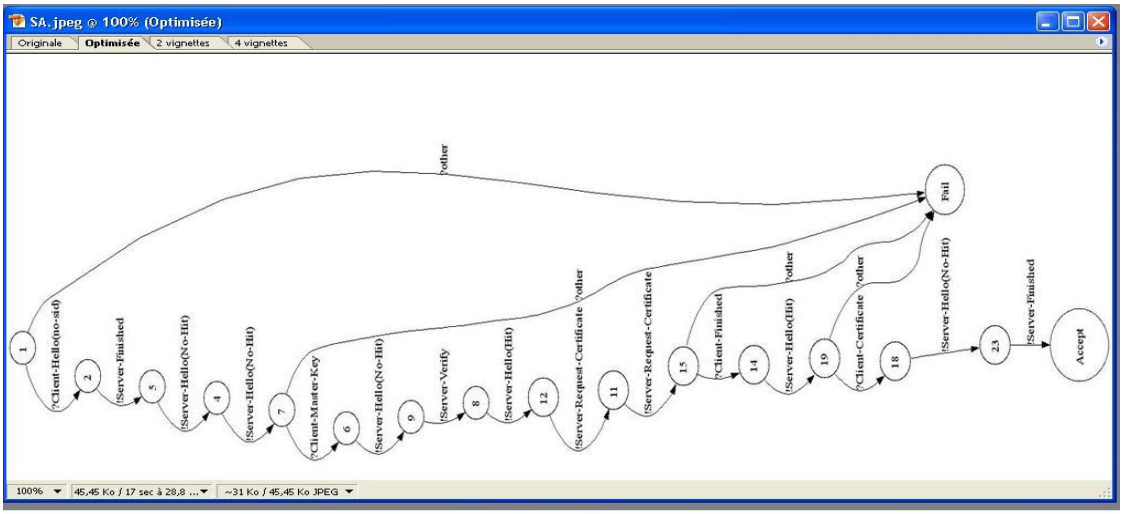
```
2 -> 9 [label = "?other"];
9 -> 3 [label = "?Server-Hello(No-Hit)"];
3 -> 10 [label = "?other"];
10 -> 4 [label = "!Client-Master-Key"];
4 -> 11 [label = "?other"];
11 -> 5 [label = "!Client-Finished"];
5 -> 12 [label = "?other"];
12 -> 6 [label = "?Server-Verify"];
6 -> 13 [label = "?other"];
13 -> 7 [label = "?Server-Finished"];
}

digraph RTP3 {
1 -> 10 [label = "?other"];
10 -> 2 [label = "!Client-Hello(no-sid)"];
2 -> 11 [label = "?other"];
11 -> 3 [label = "?Server-Hello(No-Hit)"];
3 -> 12 [label = "?other"];
12 -> 4 [label = "!Client-Master-Key"];
4 -> 13 [label = "?other"];
13 -> 5 [label = "?Server-Verify"];
5 -> 14 [label = "?other"];
14 -> 6 [label = "?Server-Request-Certificate"];
6 -> 15 [label = "?other"];
15 -> 7 [label = "!Client-Finished"];
7 -> 16 [label = "?other"];
16 -> 8 [label = "!Client-Certificate"];
8 -> 17 [label = "?other"];
17 -> 9 [label = "?Server-Finished"];
}
```

Cas de test de robustesse *RTC*



Visualisation des résultats avec l'outil GraphViz





Exemple de cas de test de robustesse codé en TTCN-3

```
tescase Tester() runs on IUT { timer ReponseTimer := 999 ;
  Tester.send(No-Certificate-Error);
  ReponseTimer.start
  alt
      [] ReponseTimer.timeout
         {  setverdict(fail);
            stop
         }
      [] Tester.receive(Client-Hello(no-sid));
         {  setverdict(pass);
            ReponseTimer.stop
        Tester.send(Bad-Certificate-Error);
        Tester.send(Server-Hello(No-Hit));
        Tester.send(No-Certificate-Error);
```

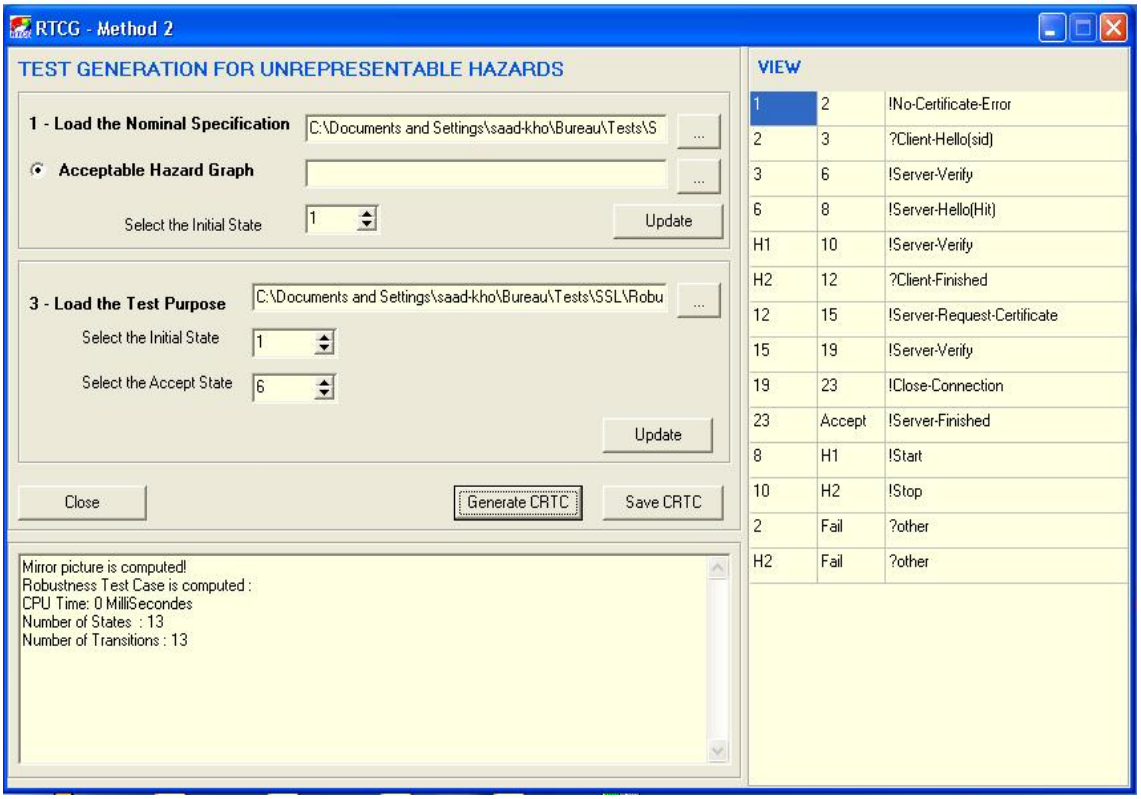
```
        ReponseTimer.start
        alt
            [] ReponseTimer.timeout
               {  setverdict(fail);
                  stop
               }
            [] Tester.receive(Client-Master-Key);
               {  setverdict(pass);
                  ReponseTimer.stop
              Tester.send(Close-Connection);
              ReponseTimer.start
              alt
                  [] ReponseTimer.timeout
                     {  setverdict(fail);
                        stop
                     }
                  [] Tester.receive(Client-Finished);
                     {  setverdict(pass);
                        ReponseTimer.stop
                  [else] { setverdict(fail);
                           stop
                         }
                  }
            [else] { setverdict(fail);
                     stop
                   }
            }
      [else] { setverdict(fail);
               stop
             }
      }
control
  {
     execute (Tester());
  }
}
```

A.2 Méthode TRACON

Exemple d'un cas de test de robustesse contrôlable codé en XML

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<IOLTS name = "Robustness Test Case">
   <States>
      <State  index = "1" >
         <Transition  Output = "!No-Certificate-Error Next = 2" >
      </State>
      <State  index = "2" >
         <Transition  Output = "!Start Next = H1" >
      </State>
      <State  index = "H1" >
         <Transition  Input = "?Client-Hello(sid) Next = 3" >
         <Transition  Input = "?other Next = Fail" >
      </State>
      <State  index = "3" >
         <Transition  Output = "!Server-Finished Next = 6" >
      </State>
      <State  index = "6" >
         <Transition  Output = "!Server-Hello(Hit) Next = 8" >
      </State>
      <State  index = "8" >
         <Transition  Output = "!Server-Verify Next = 10" >
```

```
      </State>
      <State  index = "10" >
         <Transition  Output = "!Stop Next = H2" >
      </State>
      <State  index = "H2" >
         <Transition  Input = "?Client-Finished Next = Accept" >
         <Transition  Input = "?other Next = Fail" >
      </State>
      <State  index = "Accept" >
      </State>
      <State  index = "Fail" >
      </State>
   </States>
</IOLTS>
```



Annexe B

Étude de cas sur le protocole TCP

B.1 Méthode TRACOR

Fichier de la spécification nominale

```
PROCESS TCP-Specification (1,1);
START NEXTSTATE Closed;
STATE CLOSED;
  INPUT Passive-OPEN;
  OUTPUT create-TCB;
  NEXTSTATE LISTEN;
  INPUT Active-OPEN;
  OUTPUT create-TCB;
  NEXTSTATE SYN-SENT;
STATE LISTEN;
  INPUT SYN;
  OUTPUT SYN,ACK;
  NEXTSTATE SYN-RCVD;
  INPUT SEND;
  OUTPUT SYN;
  NEXTSTATE SYN-SENT;
  INPUT Close;
  OUTPUT delete-TCB;
  NEXTSTATE CLOSED;
STATE SYN-SENT;
  INPUT SYN,ACK;
  OUTPUT ACK;
  NEXTSTATE ESTABLISHED;
  INPUT SYN;
  OUTPUT ACK;
  NEXTSTATE SYN-RCVD;
STATE SYN-RCVD;
  INPUT ACK-of-SYN;
  NEXTSTATE ESTABLISHED;
  INPUT Close;
  OUTPUT Fin;
  NEXTSTATE FIN-WAIT-1;
STATE ESTABLISHED;
  INPUT Fin;
  OUTPUT ACK;
  NEXTSTATE CLOSE-WAIT;
  INPUT Close;
  OUTPUT Fin;
```

```
  NEXTSTATE FIN-WAIT-1;
STATE FIN-WAIT-1.
  INPUT ACK-of-Fin;
  NEXTSTATE FIN-WAIT-2;
  INPUT Fin;
  OUTPUT ACK;
  NEXTSTATE CLOSING;
STATE CLOSE-WAIT;
  INPUT Close;
  OUTPUT Fin;
  NEXTSTATE LAST-ACK;
STATE FIN-WAIT-2;
  INPUT Fin;
  OUTPUT ACK;
  NEXTSTATE TIME-WAIT;
STATE CLOSING;
  INPUT ACK-of-Fin;
  NEXTSTATE TIME-WAIT;
STATE LAST-ACK;
  INPUT ACK-of-Fin;
  NEXTSTATE CLOSED;
STATE TIME-WAIT;
  INPUT Timeout=2MSL;
  OUTPUT delete TCB;
  NEXTSTATE CLOSED;
```

Fichier de graphe d'aléas

```
digraph TCP-Hazard-Graph {
2 -> 2 [label="?close-failed"];
2 -> 2 [label="?SYN-failed"];
2 -> 2 [label="?SEND-failed"];
3 -> 3 [label="?SYN,ACK-failed"];
5 -> 5 [label="?SYN,ACK-failed"];
5 -> 5 [label="?SYN-failed"];
7 -> 7 [label="?close-failed"];
7 -> 7 [label="?ACK-of-SYN-failed"];
10 ->10 [label="?close-failed"];
10 ->10 [label="?FIN-failed"];
12 -> 12 [label="?FIN-failed"];
12 -> 12 [label="?ACK-of-FIN-failed"];
14 -> 14 [label="?close-failed"];
15 -> 15 [label="?FIN-failed"];
16 -> 16 [label="?ACK-of-FIN-failed"];
17 -> 17 [label="?FIN-failed"];
19 -> 19 [label="?ACK-of-FIN-failed"];
}
```

Fichier de la spécification augmentée

```
digraph IOLTS {
1 -> 2 [label = "?passive-open"];
1 -> 5 [label = "?active-open"];
2 -> 1 [label = "?close"];
2 -> 3 [label = "?SYN"];
2 -> 4 [label = "?SEND"];
3 -> 7 [label = "!SYN,ACK"];
4 -> 5 [label = "!SYN"];
5 -> 1 [label = "?close"];
5 -> 9 [label = "?SYN,ACK"];
5 -> 6 [label = "?SYN"];
```

```
6 -> 7 [label = "!ACK-of-SYN"];
7 -> 8 [label = "?close"];
7 -> 10 [label = "?ACK-of-SYN"];
 8 -> 12 [label ="!FIN"];
9 -> 10 [label = "!ACK-of-SYN"];
10 -> 11 [label = "?close"];
10 -> 13 [label = "?FIN"];
11 -> 12 [label = "!FIN"];
12 -> 18 [label = "?FIN"];
12 -> 17 [label = "?ACK-of-FIN"];
13 -> 14 [label = "!ACK-of-FIN"];
14 -> 15 [label = "?close"];
15 -> 16 [label = "?FIN"];
16 -> 1 [label = "?ACK-of-FIN"];
17 -> 20 [label = "?FIN"];
18 -> 19 [label = "!ACK-of-FIN"];
19 -> 20 [label = "?ACK-of-FIN"];
20 -> 21 [label = "!ACK"];
21 -> 1 [label = "!Timeout=2MSL"];
2 -> 2 [label = "?close-failed"];
2 -> 2 [label = "?SYN-failed"];
2 -> 2 [label = "?SEND-failed"];
3 -> 3 [label = "?SYN,ACK-failed"];
5 -> 5 [label = "?SYN,ACK-failed"];
5 -> 5 [label = "?SYN-failed"];
7 -> 7 [label = "?close-failed"];
7 -> 7 [label = "?ACK-of-SYN-failed"];
10 -> 10 [label = "?close-failed"];
10 -> 10 [label = "?FIN-failed"];
12 -> 12 [label = "?FIN-failed"];
12 -> 12 [label = "?ACK-of-FIN-failed"];
14 -> 14 [label = "?close-failed"];
15 -> 15 [label = "?FIN-failed"]; 1
6 -> 16 [label = "?ACK-of-FIN-failed"];
17 -> 17 [label = "?FIN-failed"];
19 -> 19 [label = "?ACK-of-FIN-failed"];
1 -> 1 [label = "?close"];
1 -> 1 [label = "?SYN"];
1 -> 1 [label = "?SEND"];
1 -> 1 [label = "?SYN,ACK"];
1 -> 1 [label = "?ACK-of-SYN"];
1 -> 1 [label = "?FIN"];
1 -> 1 [label = "?ACK-of-FIN"];
1 -> 1 [label = "?close-failed"];
1 -> 1 [label = "?SYN-failed"];
1 -> 1 [label = "?SEND-failed"];
1 -> 1 [label = "?SYN,ACK-failed"];
1 -> 1 [label = "?ACK-of-SYN-failed"];
1 -> 1 [label = "?FIN-failed"];
1 -> 1 [label = "?ACK-of-FIN-failed"];
2 -> 2 [label = "?passive-open"];
2 -> 2 [label = "?active-open"];
2 -> 2 [label = "?SYN,ACK"];
2 -> 2 [label = "?ACK-of-SYN"];
2 -> 2 [label = "?FIN"];
2 -> 2 [label = "?ACK-of-FIN"];
2 -> 2 [label = "?SYN,ACK-failed"];
2 -> 2 [label = "?ACK-of-SYN-failed"];
2 -> 2 [label = "?FIN-failed"];
2 -> 2 [label = "?ACK-of-FIN-failed"];
5 -> 5 [label = "?passive-open"];
5 -> 5 [label = "?active-open"];
5 -> 5 [label = "?SEND"];
5 -> 5 [label = "?ACK-of-SYN"];
5 -> 5 [label = "?FIN"];
```

```
5 -> 5 [label = "?ACK-of-FIN"];
5 -> 5 [label = "?close-failed"];
5 -> 5 [label = "?SEND-failed"];
5 -> 5 [label = "?ACK-of-SYN-failed"];
5 -> 5 [label = "?FIN-failed"];
5 -> 5 [label = "?ACK-of-FIN-failed"];
3 -> 3 [label = "?passive-open"];
3 -> 3 [label = "?active-open"];
3 -> 3 [label = "?close"];
3 -> 3 [label = "?SYN"];
3 -> 3 [label = "?SEND"];
3 -> 3 [label = "?SYN,ACK"];
3 -> 3 [label = "?ACK-of-SYN"];
3 -> 3 [label = "?FIN"];
3 -> 3 [label = "?ACK-of-FIN"];
3 -> 3 [label = "?close-failed"];
3 -> 3 [label = "?SYN-failed"];
3 -> 3 [label = "?SEND-failed"];
3 -> 3 [label = "?ACK-of-SYN-failed"];
3 -> 3 [label = "?FIN-failed"];
3 -> 3 [label = "?ACK-of-FIN-failed"];
4 -> 4 [label = "?passive-open"];
4 -> 4 [label = "?active-open"];
4 -> 4 [label = "?close"];
4 -> 4 [label = "?SYN"];
4 -> 4 [label = "?SEND"];
4 -> 4 [label = "?SYN,ACK"];
4 -> 4 [label = "?ACK-of-SYN"];
4 -> 4 [label = "?FIN"];
4 -> 4 [label = "?ACK-of-FIN"];
4 -> 4 [label = "?close-failed"];
4 -> 4 [label = "?SYN-failed"];
4 -> 4 [label = "?SEND-failed"];
4 -> 4 [label = "?SYN,ACK-failed"];
4 -> 4 [label = "?ACK-of-SYN-failed"];
4 -> 4 [label = "?FIN-failed"];
4 -> 4 [label = "?ACK-of-FIN-failed"];
7 -> 7 [label = "?passive-open"];
7 -> 7 [label = "?active-open"];
7 -> 7 [label = "?SYN"];
7 -> 7 [label = "?SEND"];
7 -> 7 [label = "?SYN,ACK"];
7 -> 7 [label = "?FIN"];
7 -> 7 [label = "?ACK-of-FIN"];
7 -> 7 [label = "?SYN-failed"];
7 -> 7 [label = "?SEND-failed"];
7 -> 7 [label = "?SYN,ACK-failed"];
7 -> 7 [label = "?FIN-failed"];
7 -> 7 [label = "?ACK-of-FIN-failed"];
9 -> 9 [label = "?passive-open"];
9 -> 9 [label = "?active-open"];
9 -> 9 [label = "?close"];
9 -> 9 [label = "?SYN"];
9 -> 9 [label = "?SEND"];
9 -> 9 [label = "?SYN,ACK"];
9 -> 9 [label = "?ACK-of-SYN"];
9 -> 9 [label = "?FIN"];
9 -> 9 [label = "?ACK-of-FIN"];
9 -> 9 [label = "?close-failed"];
9 -> 9 [label = "?SYN-failed"];
9 -> 9 [label = "?SEND-failed"];
9 -> 9 [label = "?SYN,ACK-failed"];
9 -> 9 [label = "?ACK-of-SYN-failed"];
9 -> 9 [label = "?FIN-failed"];
9 -> 9 [label = "?ACK-of-FIN-failed"];
```

```
6 -> 6 [label = "?passive-open"];
6 -> 6 [label = "?active-open"];
6 -> 6 [label = "?close"];
6 -> 6 [label = "?SYN"];
6 -> 6 [label = "?SEND"];
6 -> 6 [label = "?SYN,ACK"];
6 -> 6 [label = "?ACK-of-SYN"];
6 -> 6 [label = "?FIN"];
6 -> 6 [label = "?ACK-of-FIN"];
6 -> 6 [label = "?close-failed"];
6 -> 6 [label = "?SYN-failed"];
6 -> 6 [label = "?SEND-failed"];
6 -> 6 [label = "?SYN,ACK-failed"];
6 -> 6 [label = "?ACK-of-SYN-failed"];
6 -> 6 [label = "?FIN-failed"];
6 -> 6 [label = "?ACK-of-FIN-failed"];
8 -> 8 [label = "?passive-open"];
8 -> 8 [label = "?active-open"];
8 -> 8 [label = "?close"];
8 -> 8 [label = "?SYN"];
8 -> 8 [label = "?SEND"];
8 -> 8 [label = "?SYN,ACK"];
8 -> 8 [label = "?ACK-of-SYN"];
8 -> 8 [label = "?FIN"];
8 -> 8 [label = "?ACK-of-FIN"];
8 -> 8 [label = "?close-failed"];
8 -> 8 [label = "?SYN-failed"];
8 -> 8 [label = "?SEND-failed"];
8 -> 8 [label = "?SYN,ACK-failed"];
8 -> 8 [label = "?ACK-of-SYN-failed"];
8 -> 8 [label = "?FIN-failed"];
8 -> 8 [label = "?ACK-of-FIN-failed"];
10 -> 10 [label = "?passive-open"];
10 -> 10 [label = "?active-open"];
10 -> 10 [label = "?SYN"];
10 -> 10 [label = "?SEND"];
10 -> 10 [label = "?SYN,ACK"];
10 -> 10 [label = "?ACK-of-SYN"];
10 -> 10 [label = "?ACK-of-FIN"];
10 -> 10 [label = "?SYN-failed"];
10 -> 10 [label = "?SEND-failed"];
10 -> 10 [label = "?SYN,ACK-failed"];
10 -> 10 [label = "?ACK-of-SYN-failed"];
10 -> 10 [label = "?ACK-of-FIN-failed"];
12 -> 12 [label = "?passive-open"];
12 -> 12 [label = "?active-open"];
12 -> 12 [label = "?close"];
12 -> 12 [label = "?SYN"];
12 -> 12 [label = "?SEND"];
12 -> 12 [label = "?SYN,ACK"];
12 -> 12 [label = "?ACK-of-SYN"];
12 -> 12 [label = "?close-failed"];
12 -> 12 [label = "?SYN-failed"];
12 -> 12 [label = "?SEND-failed"];
12 -> 12 [label = "?SYN,ACK-failed"];
12 -> 12 [label = "?ACK-of-SYN-failed"];
11 -> 11 [label = "?passive-open"];
11 -> 11 [label = "?active-open"];
11 -> 11 [label = "?close"];
11 -> 11 [label = "?SYN"];
11 -> 11 [label = "?SEND"];
11 -> 11 [label = "?SYN,ACK"];
11 -> 11 [label = "?ACK-of-SYN"];
11 -> 11 [label = "?FIN"];
11 -> 11 [label = "?ACK-of-FIN"];
```

```
11 -> 11 [label = "?close-failed"];
11 -> 11 [label = "?SYN-failed"];
11 -> 11 [label = "?SEND-failed"];
11 -> 11 [label = "?SYN,ACK-failed"];
11 -> 11 [label = "?ACK-of-SYN-failed"];
11 -> 11 [label = "?FIN-failed"];
11 -> 11 [label = "?ACK-of-FIN-failed"];
13 -> 13 [label = "?passive-open"];
13 -> 13 [label = "?active-open"];
13 -> 13 [label = "?close"];
13 -> 13 [label = "?SYN"];
13 -> 13 [label = "?SEND"];
13 -> 13 [label = "?SYN,ACK"];
13 -> 13 [label = "?ACK-of-SYN"];
13 -> 13 [label = "?FIN"];
13 -> 13 [label = "?ACK-of-FIN"];
13 -> 13 [label = "?close-failed"];
13 -> 13 [label = "?SYN-failed"];
13 -> 13 [label = "?SEND-failed"];
13 -> 13 [label = "?SYN,ACK-failed"];
13 -> 13 [label = "?ACK-of-SYN-failed"];
13 -> 13 [label = "?FIN-failed"];
13 -> 13 [label = "?ACK-of-FIN-failed"];
18 -> 18 [label = "?passive-open"];
18 -> 18 [label = "?active-open"];
18 -> 18 [label = "?close"];
18 -> 18 [label = "?SYN"];
18 -> 18 [label = "?SEND"];
18 -> 18 [label = "?SYN,ACK"];
18 -> 18 [label = "?ACK-of-SYN"];
18 -> 18 [label = "?FIN"];
18 -> 18 [label = "?ACK-of-FIN"];
18 -> 18 [label = "?close-failed"];
18 -> 18 [label = "?SYN-failed"];
18 -> 18 [label = "?SEND-failed"];
18 -> 18 [label = "?SYN,ACK-failed"];
18 -> 18 [label = "?ACK-of-SYN-failed"];
18 -> 18 [label = "?FIN-failed"];
18 -> 18 [label = "?ACK-of-FIN-failed"];
17 -> 17 [label = "?passive-open"];
17 -> 17 [label = "?active-open"];
17 -> 17 [label = "?close"];
17 -> 17 [label = "?SYN"];
17 -> 17 [label = "?SEND"];
17 -> 17 [label = "?SYN,ACK"];
17 -> 17 [label = "?ACK-of-SYN"];
17 -> 17 [label = "?ACK-of-FIN"];
17 -> 17 [label = "?close-failed"];
17 -> 17 [label = "?SYN-failed"];
17 -> 17 [label = "?SEND-failed"];
17 -> 17 [label = "?SYN,ACK-failed"];
17 -> 17 [label = "?ACK-of-SYN-failed"];
17 -> 17 [label = "?ACK-of-FIN-failed"];
14 -> 14 [label = "?passive-open"];
14 -> 14 [label = "?active-open"];
14 -> 14 [label = "?SYN"];
14 -> 14 [label = "?SEND"];
14 -> 14 [label = "?SYN,ACK"];
14 -> 14 [label = "?ACK-of-SYN"];
14 -> 14 [label = "?FIN"];
14 -> 14 [label = "?ACK-of-FIN"];
14 -> 14 [label = "?SYN-failed"];
14 -> 14 [label = "?SEND-failed"];
14 -> 14 [label = "?SYN,ACK-failed"];
14 -> 14 [label = "?ACK-of-SYN-failed"];
```

```
14 -> 14 [label = "?FIN-failed"];
14 -> 14 [label = "?ACK-of-FIN-failed"];
15 -> 15 [label = "?passive-open"];
15 -> 15 [label = "?active-open"];
15 -> 15 [label = "?close"];
15 -> 15 [label = "?SYN"];
15 -> 15 [label = "?SEND"];
15 -> 15 [label = "?SYN,ACK"];
15 -> 15 [label = "?ACK-of-SYN"];
15 -> 15 [label = "?ACK-of-FIN"];
15 -> 15 [label = "?close-failed"];
15 -> 15 [label = "?SYN-failed"];
15 -> 15 [label = "?SEND-failed"];
15 -> 15 [label = "?SYN,ACK-failed"];
15 -> 15 [label = "?ACK-of-SYN-failed"];
15 -> 15 [label = "?ACK-of-FIN-failed"];
16 -> 16 [label = "?passive-open"];
16 -> 16 [label = "?active-open"];
16 -> 16 [label = "?close"];
16 -> 16 [label = "?SYN"];
16 -> 16 [label = "?SEND"];
16 -> 16 [label = "?SYN,ACK"];
16 -> 16 [label = "?ACK-of-SYN"];
16 -> 16 [label = "?FIN"];
16 -> 16 [label = "?close-failed"];
16 -> 16 [label = "?SYN-failed"];
16 -> 16 [label = "?SEND-failed"];
16 -> 16 [label = "?SYN,ACK-failed"];
16 -> 16 [label = "?ACK-of-SYN-failed"];
16 -> 16 [label = "?FIN-failed"];
20 -> 20 [label = "?passive-open"];
20 -> 20 [label = "?active-open"];
20 -> 20 [label = "?close"];
20 -> 20 [label = "?SYN"];
20 -> 20 [label = "?SEND"];
20 -> 20 [label = "?SYN,ACK"];
20 -> 20 [label = "?ACK-of-SYN"];
20 -> 20 [label = "?FIN"];
20 -> 20 [label = "?ACK-of-FIN"];
20 -> 20 [label = "?close-failed"];
20 -> 20 [label = "?SYN-failed"];
20 -> 20 [label = "?SEND-failed"];
20 -> 20 [label = "?SYN,ACK-failed"];
20 -> 20 [label = "?ACK-of-SYN-failed"];
20 -> 20 [label = "?FIN-failed"];
20 -> 20 [label = "?ACK-of-FIN-failed"];
19 -> 19 [label = "?passive-open"];
19 -> 19 [label = "?active-open"];
19 -> 19 [label = "?close"];
19 -> 19 [label = "?SYN"];
19 -> 19 [label = "?SEND"];
19 -> 19 [label = "?SYN,ACK"];
19 -> 19 [label = "?ACK-of-SYN"];
19 -> 19 [label = "?FIN"];
19 -> 19 [label = "?close-failed"];
19 -> 19 [label = "?SYN-failed"];
19 -> 19 [label = "?SEND-failed"];
19 -> 19 [label = "?SYN,ACK-failed"];
19 -> 19 [label = "?ACK-of-SYN-failed"];
19 -> 19 [label = "?FIN-failed"];
21 -> 21 [label = "?passive-open"];
21 -> 21 [label = "?active-open"];
21 -> 21 [label = "?close"];
21 -> 21 [label = "?SYN"];
21 -> 21 [label = "?SEND"];
```

```
21 -> 21 [label = "?SYN,ACK"];
21 -> 21 [label = "?ACK-of-SYN"];
21 -> 21 [label = "?FIN"];
21 -> 21 [label = "?ACK-of-FIN"];
21 -> 21 [label = "?close-failed"];
21 -> 21 [label = "?SYN-failed"];
21 -> 21 [label = "?SEND-failed"];
21 -> 21 [label = "?SYN,ACK-failed"];
21 -> 21 [label = "?ACK-of-SYN-failed"];
21 -> 21 [label = "?FIN-failed"];
21 -> 21 [label = "?ACK-of-FIN-failed"];
1 -> 1 [label = "!delta"];
2 -> 2 [label = "!delta"];
5 -> 5 [label = "!delta"];
7 -> 7 [label = "!delta"];
10 -> 10 [label = "!delta"];
12 -> 12 [label = "!delta"];
17 -> 17 [label = "!delta"];
14 -> 14 [label = "!delta"];
15 -> 15 [label = "!delta"];
16 -> 16 [label = "!delta"];
19 -> 19 [label = "!delta"]; }
```

Objectifs de test de robustesse

```
digraph RTP1 {
1 -> 2 [label = "?other"];
2 -> 3 [label = "?passive-open"];
3 -> 4 [label = "?other"];
4 -> 5 [label = "?SYN"];
5 -> 6 [label = "?other"];
6 -> 7 [label = "?SYN,ACK"];
7 -> 8 [label = "?other"];
8 -> 9 [label = "?ACK-of-SYN"];
}
```

```
digraph RTP2 {
1 -> 5 [label="?active-open"];
5 -> 6 [label="?SYN"];
6 -> 7 [label="!ACK-of-SYN"];
7 -> 10 [label="?ACK-of-SYN"];
1 -> 1 [label="?other"];
5 -> 5 [label="?other"];
6 -> 6 [label="?other"];
7 -> 7 [label="?other"]; }
```

Exemple d'un cas de test de robustesse

```
tescase TCP-Tester() runs on TCP-IUT
{ timer ReponseTimer := 144 ;
  Tester.send(close);
  Tester.send(passive-open);
  Tester.send(ACK-of-FIN-failed);
  Tester.send(SEND);
  Tester.send(SYN,ACK);
  ReponseTimer.start
  alt
      [] ReponseTimer.timeout
         {  setverdict(fail);
            stop
         }
```

```
      [] Tester.receive(SYN);
         {  setverdict(pass);
            ReponseTimer.stop

        Tester.send(SYN);
        Tester.send(SYN,ACK);
        Tester.send(SYN,ACK);
        ReponseTimer.start
        alt
            [] ReponseTimer.timeout
               {  setverdict(fail);
                  stop
               }
            [] Tester.receive(ACK-of-SYN);
               {  setverdict(pass);
                  ReponseTimer.stop

            [else] { setverdict(fail);
                     stop
                   }
            }
      [else] { setverdict(fail);
               stop
             }
      }


control
  {
     execute (TCP-Tester());
  }
}
```

B.2 Méthode TRACON

Graphe de sorties acceptables

```
digraph AOG {
1 -> 1 [label="!RST"];
2 -> 1 [label="!RST"];
3 -> 1 [label="!RST"];
4 -> 1 [label="!RST"];
5 -> 1 [label="!RST"];
6 -> 1 [label="!RST"];
7 -> 1 [label="!RST"];
8 -> 1 [label="!RST"];
9 -> 1 [label="!RST"];
10 -> 1 [label="!RST"];
11 -> 1 [label="!RST"];
12 -> 1 [label="!RST"];
13 -> 1 [label="!RST"];
14 -> 1 [label="!RST"];
15 -> 1 [label="!RST"];
16 -> 1 [label="!RST"];
17 -> 1 [label="!RST"];
18 -> 1 [label="!RST"];
19 -> 1 [label="!RST"];
20 -> 1 [label="!RST"];
21 -> 1 [label="!RST"];
}
```

Spécification semi-augmentée

```
digraph TCP Spec Semi-augmentee{
1 -> 2 [label = "?passive-open"];
1 -> 5 [label = "?active-open"];
2 -> 1 [label = "?close"];
2 -> 3 [label = "?SYN"];
2 -> 4 [label = "?SEND"];
3 -> 7 [label = "!SYN,ACK"];
4 -> 5 [label = "!SYN"];
5 -> 1 [label = "?close"];
5 -> 9 [label = "?SYN,ACK"];
5 -> 6 [label = "?SYN"];
6 -> 7 [label = "!ACK-of-SYN"];
7 -> 8 [label = "?close"];
7 -> 10 [label = "?ACK-of-SYN"];
8 -> 12 [label = "!FIN"];
9 -> 10 [label = "!ACK-of-SYN"];
10 -> 11 [label = "?close"];
10 -> 13 [label = "?FIN"];
11 -> 12 [label = "!FIN"];
12 -> 18 [label = "?FIN"];
12 -> 17 [label = "?ACK-of-FIN"];
13 -> 14 [label = "!ACK-of-FIN"];
14 -> 15 [label = "?close"];
15 -> 16 [label = "?FIN"];
16 -> 1 [label = "?ACK-of-FIN"];
17 -> 20 [label = "?FIN"];
18 -> 19 [label = "!ACK-of-FIN"];
19 -> 20 [label = "?ACK-of-FIN"];
20 -> 21 [label = "!ACK"];
21 -> 1 [label = "!Timeout=2MSL"];
1 -> 1 [label = "!delta"];
2 -> 2 [label = "!delta"];
5 -> 5 [label = "!delta"];
7 -> 7 [label = "!delta"];
10 -> 10 [label = "!delta"];
12 -> 12 [label = "!delta"];
17 -> 17 [label = "!delta"];
14 -> 14 [label = "!delta"];
15 -> 15 [label = "!delta"];
16 -> 16 [label = "!delta"];
19 -> 19 [label = "!delta"];
1 -> 1 [label = "!RST"];
2 -> 1 [label = "!RST"];
3 -> 1 [label = "!RST"];
4 -> 1 [label = "!RST"];
5 -> 1 [label = "!RST"];
6 -> 1 [label = "!RST"];
7 -> 1 [label = "!RST"];
8 -> 1 [label = "!RST"];
9 -> 1 [label = "!RST"];
10 -> 1 [label = "!RST"];
11 -> 1 [label = "!RST"];
12 -> 1 [label = "!RST"];
13 -> 1 [label = "!RST"];
14 -> 1 [label = "!RST"];
15 -> 1 [label = "!RST"];
16 -> 1 [label = "!RST"];
17 -> 1 [label = "!RST"];
18 -> 1 [label = "!RST"];
19 -> 1 [label = "!RST"];
20 -> 1 [label = "!RST"];
21 -> 1 [label = "!RST"];
}
```

Exemple d'un cas de test de robustesse contrôlable

```
digraph CRTC{
1 -> 3 [label = "!active-open"];
3 -> 6 [label = "!passive-open"];
6 -> 7 [label = "!close"];
7 -> 14 [label = "!SYN"];
14 -> 21 [label = "!SEND"];
21 -> 28 [label = "!SYN,ACK"];
H1 -> 32 [label = "!FIN"];
H2 -> Accept [label = "!ACK-of-SYN"];
28 -> H1 [label = "!Start"];
32 -> H2 [label = "!Stop"];
}
```

Index