Une approche basée sur les modèles pour le test de robustesse

Cyril Alexandre Pachon

UNIVERSITÉ JOSEPH FOURIER - GRENOBLE I
SCIENCES ET GEOGRAPHIE

THÈSE

pour obtenir le grade de

DOCTEUR DE L'UNIVERSITÉ JOSEPH FOURIER

Discipline : Informatique

présentée et soutenue publiquement par

Cyril Alexandre PACHON

Le 4 Octobre 2005

TITRE :

UNE APPROCHE BASÉE SUR LES MODÈLES POUR LE TEST DE ROBUSTESSE

Directeurs de thèse :
Jean-Claude Fernandez et Dorel Marius Bozga

COMPOSITION DU JURY :

Président	**Roland Groz**
Directeurs	**Jean-Claude Fernandez**
	Dorel Marius Bozga
Rapporteurs	**Richard Castanet**
	Thierry Jéron

Remerciements

Je remercie tout d'abord *Messieurs* Jean-Claude Fernandez, *Professeur à l'Université Joseph Fourier, Grenoble I,* et Dorel Marius Bozga, *Ingénieur de Recherche au CNRS, Grenoble,* mes directeurs de thèse qui m'ont fait confiance et m'ont permis de mener à bien cette thèse. Je les remercie pour leur disponibilité, les corrections qu'ils ont apportées à ce travail, leur engagement, leur soutien, leur compétence et leur exigence.

Je remercie *Messieurs* Richard Castanet, *Professeur classe exceptionnelle à l'ENSEIRB de l'Université de Bordeaux I,* et Thierry Jèron *Chargé de recherche et responsable scientifique du projet VerTeCs à l'IRISA/INRIA, Rennes,* pour avoir accepté de juger ce travail. Merci, pour les commentaires, les remarques et les suggestions qui m'ont permis d'améliorer mon travail.

Je remercie *Monsieur* Roland Groz, *Professeur à l'ENSIMAG/INPG de Genoble I,* pour m'avoir fait l'honneur de présider cette thèse.

Je remercie chaleureusement *Monsieur* Laurent Mounier, *Maître de conférences à l'Université Joseph Fourier, Grenoble I,* pour son aide, ses conseils, sa compétence, son exigence et toutes les corrections constructives apportées à mon travail.

Je remercie *Monsieur* Joseph Sifakis, *directeur de recherche au CNRS et directeur du laboratoire Verimag/IMAG, Grenoble,* qui m'a accueilli au sein de son équipe.

Je remercie *Monsieur* Yassine Lakhnech, *Professeur de l'Université Joseph Fourier, et responsable de l'équipe DSC au laboratoire Verimag/IMAG, Grenoble I,* qui m'a apporté son soutien pour réaliser dans de bonnes conditions cette thèse.

Je remercie tous les membres du laboratoire Verimag que j'ai croisés pendant mes années de thèse, vous, Lionel, David, Ana, Liana, Moussa, Mickael .... qui ont partagé de bons moments, un café, un séminaire ou une école jeune chercheur je vous dis à bientôt.

Je remercie également Fabien, Elaine, Ronald, Lionel, Sébastien, Maryline, Rachel, ... Julie, Bénédicte, Cécile, ... Isabelle, Sylvain ... et ce pour divers raisons. Je tiens particulièrement à remercie Julien qui non seulement pense à moi au point de me remercier dans sa thèse, mais parce qu'il a su me remettre à niveau lorsque j'en ai eu le plus besoin, merci à toi Julien et à Eva.

Je remercie Lina avec qui j'ai partagé de nombreuses activités qui m'ont gardé en forme pour accomplir toutes ces années de thèse.

Enfin, je dédie cette thèse à ma Maman, Anny, et mon Papa, Marcel, qui sans leur soutien, je n'aurais jamais fait d'étude. Je les remercie de m'avoir donné de vraies valeurs, l'envie d'un travail accompli, la patience et l'acharnement pour le réussir. Je dédie cette thèse à Franck, nom frère, qui a toujours su rester présent pendant les difficiles et les bons moments.

à tous merci...

et balade !...FP

Table des matières

Table des figures

Liste des Algorithmes

Introduction

L'utilisation des systèmes informatiques prend une place de plus en plus importante dans la vie de chacun. Qui n'a pas utilisé un jour, un téléphone portable ou un organisateur électronique ? Si nous parlons sans risque d'une erreur intervenue lors d'une communication téléphonique, quelles seraient les conséquences d'une erreur survenue dans un programme, devenu point névralgique d'un système de freinage (d'un avion ou d'un train) ou d'un contrôleur de centrale nucléaire ? La réponse est sans hésiter une catastrophe. Or, ces programmes considérés comme critiques sont pourtant utilisés en permanence dans le monde qui nous entoure. Les manières de créer et de développer des programmes informatiques standard ne cessent d'évoluer. Nous parlons désormais constamment de composants, d'agents, de code réutilisable et portable. De plus, la complexité croissante d'un progamme, due à la répartition de code, à la réutilisation de composants, à la limitation des ressources, sont autant de facteurs qui rendent sa validation difficile. Un programme qui fonctionne dans un contexte précis et défini, fonctionne-t-il dans un autre contexte ? Ce sont toutes ces interrogations qui alimentent le besoin d'effectuer des tests avec des méthodes de conception et de validation rigoureuses.

Définition I.0.1 (Le test) *Selon la norme proposée par IEEE 729, le test vise à établir qu'un système doit vérifier des propriétés exigées par sa spécification. Il peut aussi servir à détecter des différences entre les résultats engendrés par le système et ceux attendus par la spécification. Le test permet de mettre en évidence les erreurs d'un programme. Par contre, le test n'a pas pour objectif de diagnostiquer la cause des erreurs, de corriger les fautes ou de prouver la correction d'un programme.*

Cependant, la phase de test reste la dernière étape d'un processus de développement d'un logiciel. Car, cette dernière phase est généralement considérée comme coûteuse en temps par rapport au coût global de conception d'un programme. Pourtant, dans le cas des systèmes critiques, la qualité du programme et du test doivent augmenter pour tendre vers des systèmes avec zéro défaut. C'est avec tous ces paramètres que les chercheurs et praticiens du test s'intéressent depuis de nombreuses années à automatiser la phase de test. Automatiser la phase de test permet de réduire les coûts tout en maintenant un niveau de qualité de l'application. Il existe plusieurs méthodes de test : *les méthodes de test statiques*, en utilisant une analyse textuelle du programme à tester, sans l'exécuter, et *les méthodes de test dynamiques*, en confrontant les résultats des comportements obtenus par

l'exécution du programme à ceux attendus par une spécification de référence.

I.1 Contexte : Le test de Conformité

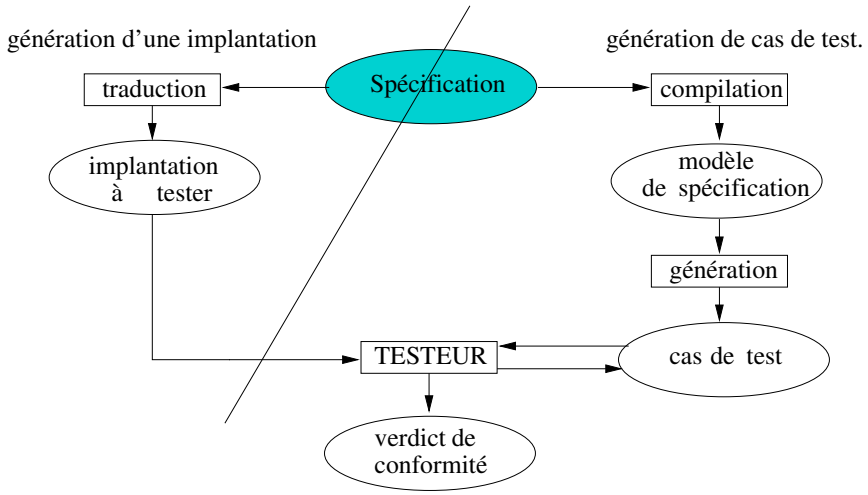

FIG. 1 – Génération et principe d'exécution de cas de test pour le test de conformité.

Pour décrire les protocoles de communications, de manière concise, complète et non ambiguë des *langages de spécification formelle* ont été conçus : ESTELLE [ISO89, BD88], LOTOS [ISO87, BB88] et SDL [IT99a, ST87]. Ces formalismes, désignés par le nom générique de langage FDT (Formal Decription Techniques), ont été définis et normalisés par des organismes internationaux de standardisation dans les domaines des télécommunications comme ISO (International Standards Organisation) et l'ITU (International Telecommunication Union). Assistés par des méthodologies de développement et des outils allant de simples éditeurs et analyseurs syntaxiques jusqu'à des vérificateurs et des générateurs de code, ces langages constituent actuellement la base formelle de ce que nous appelons l'ingénierie des protocoles. Depuis leur définition, les langages FDT connaissent une évolution continue, soigneusement contrôlée par les comités de normalisation.

Dans le domaine des télécommunications, l'utilisation des méthodes formelles a permis de développer une théorie du test de conformité. Le test de conformité est un test fonctionnel, permettant de vérifier les comportements d'une implantation par rapport à sa spécification. Généralement, c'est une méthode de test de type boîte noire, c'est-à-dire que le code de l'implantation est inconnu. La génération automatique de tests nécessite de formaliser les différents concepts liés à la conformité : la spécification, les interactions entre l'implantation à tester et le testeur, et la relation de conformité. A partir d'une spécification formelle, nous extrayions des cas de test décrivant une suite d'interaction entre le testeur et l'implantation. Les algorithmes mis en œuvre sont basés sur ceux utilisés dans les outils de vérification par les modèles (*model checking*).

Le travail proposé

Le travail se positionne sur une approche inspirée des techniques de génération de test. Plus précisément, sur une méthode de test axée sur la conformité des protocoles de communications dans lesquelles des séquences de test sont générées à partir d'un modèle comportementale du logiciel.

Dans l'approche du test de conformité (dont une méthode sera présentée), la construction automatique des séquences de test est réalisée à partir d'une spécification donnée (et accessoirement d'un objectif de test). Or, si cette construction automatique est réaliste pour certaines classes d'applications, elle n'est toutefois pas toujours facilement praticable. Nous constatons également qu'en pratique, nous disposons que très rarement d'une spécification, aussi "complète" et "exhaustive" que souhaitée, pour concevoir, décrire et développer les fonctionnalités d'un logiciel (même déjà implémenté). De plus, l'écriture (ou l'amélioration) d'une spécification formelle peut devenir une tâche coûteuse, qui ne peut être envisagée pour toutes les applications logicielles. Une perspective, dans notre méthode de test, est de construire automatiquement des séquences de test à partir d'une *spécification partielle*. Nous proposons d'utiliser cette perspective, pour tester des implantations en fonction de propriétés à satisfaire, où la construction automatique est réalisée à partir d'une spécification partielle et d'une propriété données. Le principe de construction permettra d'inclure aux séquences de test, les caractéristiques de la propriété (à tester), guidées par les actions de la spécification partielle. Les séquences seront ensuite exécutées, à l'aide d'un testeur, sur l'implantation à tester. Le but de la méthode sera de tester si la propriété donnée est préservée ou non par une exécution de l'implantation. Nous parlerons alors d'une méthode de test axée sur la *satisfaction de propriétés* nommée *test de propriétés*.

Dans cette thèse, nous nous intéresserons ensuite et plus particulièrement à définir une méthode automatique de génération et d'exécution de séquences de test, destinée à évaluer la *robustesse* d'un système. En d'autres termes, nous allons définir une méthode pour tester la capacité d'un système à respecter certaines propriétés comportementales en dépit d'un environnement d'exécution dégradé.

Les systèmes testés, représentent des ensembles de composants interagissant entre eux. Dans la conception même d'un tel système, un composant peut être réutilisable ou mis à la place d'un autre. Le test proposé, devra alors prendre en compte la diversité des utilisations potentielles des composants du système, d'une part pour consolider leur aptitude à réagir correctement aux sollicitations, et d'autre part pour permettre l'identification explicite de leurs modes de fonctionnement ou de défaillances. Ces propositions ont donc pour but de faciliter la mise en œuvre de parades lors de l'intégration des différents composants.

Le travail consiste à développer une méthode de *test de robustesse* d'un protocole de communications. Plus particulièment, nous voulons définir une méthode, pour tester un système devant s'exécuter de façon autonome, en dépit d'aléas, et en particulier en dépit de fautes internes ou externes au système. La méthode permettra de construire, puis d'exécuter sur des implantations, des séquences de test, comportant les fautes ou les aléas (les fautes étant indentifiées préalablement). La méthode de construction des séquences exécutées reprendra alors le principe du test de propriétés. Dont, le principe global reste un parallèle de la méthode du test de conformité, présentée en première partie du document. Nous parlerons alors de *test de robustesse* de programme.

I.2 Notre approche

Nous proposons, dans cette thèse, une nouvelle alternative pour générer et exécuter automatiquement des cas de test, orientée test de robustesse, basée sur les modèles. L'approche choisie est inspirée d'une méthode de génération de cas de test de conformité qui utilise la relation de conformité **ioco** [Tre96] et les méthodes de l'outil TGV [JJ02]. Cette méthode de test est fondée sur les modèles des systèmes de transitions. Elle suppose l'existence d'une spécification formelle, dont la sémantique est un système de transitions, et le fait que l'ensemble des interactions entre l'implantation sous test et son environnement peut être modélisé aussi par un système de transitions. Cette présentation permet d'identifier les hypothèses, les techniques de génération de cas de test et d'éventuelles restrictions pour les adapter à notre méthode.

Selon la relation **ioco**, une implantation est non conforme si une exécution comporte au moins une action de sortie interdite (non spécifiée). Les propriétés testées sont uniquement des propriétés de ”sûreté”.

> Une première question est : ”Pouvons-nous proposer et modéliser d'autres classes de propriétés que celle de sûreté ?”

Il est bien évident que toutes les propriétés ne peuvent pas être testées, mais, nous pouvons en identifier certaines. De par la nature du test, un cas de test modélisé par un automate, représente un ensemble de séquences finies d'exécution. Pouvons-nous envisager de tester des propriétés de vivacité ? À cette question, nous pouvons d'ores et déjà dire non. Une propriété de vivacité représente des séquences infinies d'exécution, il nous est donc impossible de les tester. Mais, si les séquences d'exécution sont infinies, elles peuvent être représentées par des modèles d'automate d'états finis, comme les automates de Muller, de Büchi, ... Pour tester ces propriétés, nous envisageons de borner leur exécution. Pour cela, il est raisonnable d'effectuer un aménagement des modèles et principes de génération de séquences de test (dans le cas où un utilisateur veut ”tester” des propriétés de vivacité). Pour modéliser et vérifier de telles propriétés, nous proposons de borner l'exécution des tests en paramétrant les cas de test avec des valeurs fixées par l'utilisateur. Dans un premier temps, il nous faut donc donner un nouveau modèle (automate déterministe paramétré) pour représenter toutes les propriétés envisagées. Puis, nous composerons avec ce nouveau modèle pour obtenir des séquences de test (dont l'exécution sera bornée). Cette étude permet de tester la classe des propriétés paramétrées que nous nommons ”vivacité bornée” (bounded liveness).

> Nous proposons de tester des propriétés de sûreté et des propriétés de ”vivacité bornée”.

Une hypothèse forte pour générer (principe donné dans par la figure 1) des cas de test de conformité est l'utilisation d'une spécification supposée ”complète”, ”exhaustive” et ”non ambiguë”. Dans la méthode de test de conformité présentée (en première partie), la construction de cas de test repose

uniquement sur les actions données par la spécification (la spécification servant d'oracle au test). Cette spécification est donc complète et exhaustive dans le sens où elle décrit, à elle seule, toutes les actions et comportements prévus pour réaliser une fonctionnalité. Dans ce cas, nous constatons que produire des cas de test, avec une spécification partielle limite la pertinence donnée par les cas de test produits. En effet, si une spécification est supposée partielle, elle ne représente pas toutes les actions et les comportements souhaités. Et donc, par conséquence, les séquences produites avec de telles spécifications ne comportent pas toutes les actions utiles pour tester de façon efficace la conformité en fonction de la réalisation souhaitée. Or, comme la conformité est effectuée vis-à-vis des actions et des comportements spécifiés, certaines implantations seraient jugées non conformes seulement parce que nous disposons que d'une spécification partielle (absence de certaines actions de l'implantation dans le modèle). Avec de telles conditions, il est bien évident que la spécification joue un rôle majeur pour la génération et le test des implantations. Et, même si la spécification est partielle, elle reste la référence pour le test de conformité. Nous proposons une méthode pour être en mesure de construire des cas de test, même en présence d'une spécification partielle, pour vérifier certaines propriétés sur les implantations à tester. Pour cela, nous proposons que la spécification ne soit pas la seule hypothèse de construction des cas de test. Nous proposons alors de construire des cas de test à l'aide d'une spécification partielle et d'une propriété (à vérifier). Dans ce cas, les cas de test comportent les actions de la spécification, complétés éventuellement par les actions de la propriété (si la spécification ne comporte pas toutes les actions décrites par le modèle de la propriété). Dans notre méthode, la propriété sert d'oracle au test. En résumé, nous proposons une méthode de test dynamique en mesure de générer et d'exécuter des cas de test avec des propriétés et une spécification partielle.

Une deuxième question : "Pouvons-nous générer des cas de test en utilisant des spécifications partielles ? Pouvons-nous considérer que les actions d'une spécification deviennent uniquement des guides pour amener une exécution de cas de test dans certaines parties d'une implantation ?"

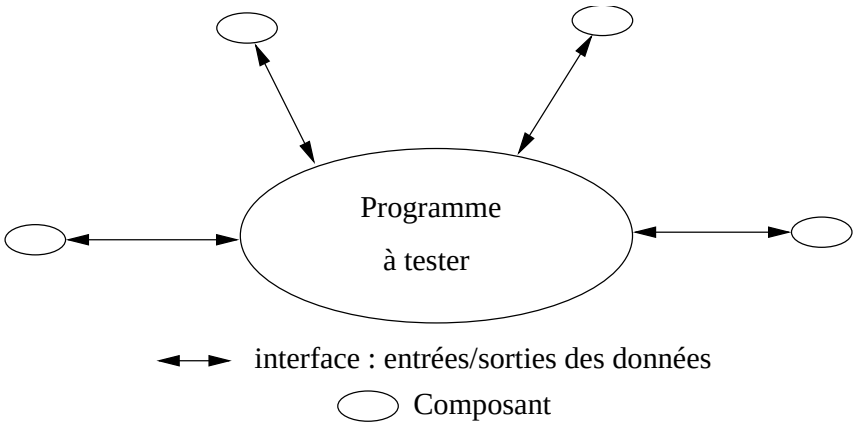

FIG. 2 – Programme à tester avec son environnement initial

La spécification donnée pour le test de conformité comporte une description sous-jacente des conditions d'*environnement nominal*. Nous entendons par environnement nominal (figure 2) toutes les actions sortantes et entrantes de l'implantation à tester ainsi que toutes les conditions nécessaires, provenant des autres éléments composant un système, pour interagir avec l'implantation. En d'autres termes, toutes les données et les valeurs d'interaction définissent un environnement nominal. Dans ces conditions, une spécification qui comporte toutes les actions à vérifier contient également un environnement nominal (soit toutes les interfaces des composants du système testé). Nous proposons de faire évoluer cet environnement en fonction des conditions de fonctionnement "réelles", c'est à dire lorsque celles-ci évoluent. Nous parlons alors d'*environnement dégradé* du composant testé. Entre une description initialement prévue par une spécification nominale et une implantation, les actions de l'interface peuvent être différentes, car l'implantation peut intégrer de nouveaux paramètres selon certaines conditions "réelles" d'utilisation et être sujettes à des aléas internes et externes. Pour générer des cas de test, nous intégrons les modifications de l'environnement en utilisant une spécification. Le choix d'intégrer de nouveaux comportements à une spécification initiale s'explique par : soit nous considérons que la spécification initiale est partielle, et l'intégration permet de prendre en compte des informations supplémentaires, soit l'implantation comporte de nouveaux paramètres, car elle est conçue pour évoluer dans un environnement dégradé et donc dans ce cas la spécification initiale est devenue partielle.

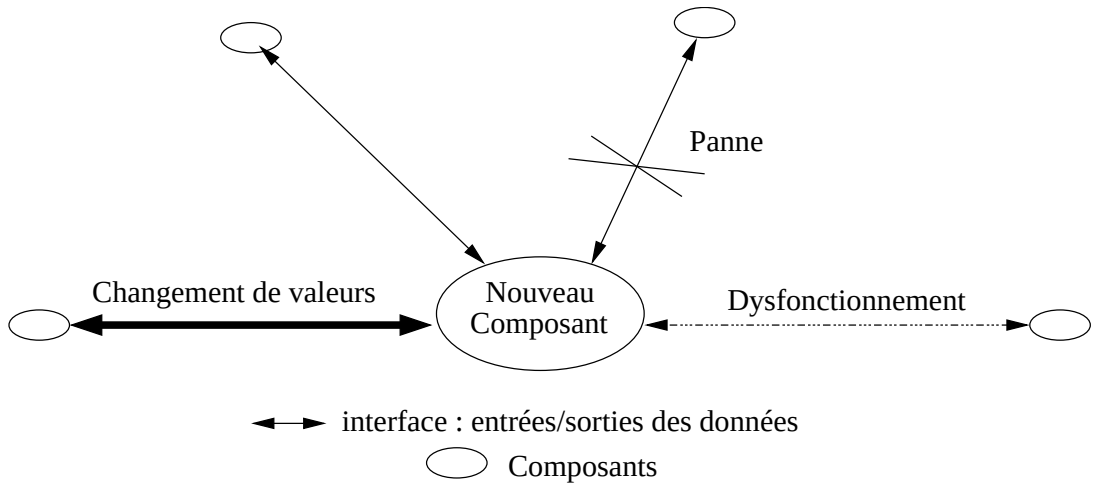

FIG. 3 – Nouveau composant dans un système avec un environnement dégradé (nouvelle interface avec des pannes, des dysfonctionnements et des changements de données éventuels).

Dans le cadre du test orienté robustesse, nous proposons de voir comment cet environnement peut (doit) évoluer (se comporter) lorsque l'implantation (ou un de ses composants) est réutilisée dans un autre environnement. Cette évolution d'environnement aura pour impact de changer (supprimer, ajouter ou modifier) des paramètres d'entrée et de sortie. De plus, les implantations peuvent être conçues pour être robustes à certains aléas et donc comporter des actions "non spécifiées" initialement. Par exemple, dans une description nominale d'un protocole, le fait d'avoir des com-

munications fiables ou non fiables peut être implicite. Pour le test de robustesse ces conditions de fonctionnement sont alors données explicitement par l'environnement dégradé redéfinissant les paramètres de l'interface en incluant éventuellement les pannes, les coupures et d'autres dysfonctionnements appliqués lors du fonctionnement du système (figure 3).

La troisième question : "Pouvons-nous intégrer de nouveaux comportements à la spécification initiale, dans le but de vérifier tous les comportements et aléas de fonctionnement des implantations testées ?"

La relation de conformité **ioco** présentée est une inclusion de traces de l'implantation dans celles de la spécification nominale. Pour la relation **ioco**, les actions de sortie non spécifiées, rendent les implantations testées non conformes. Nous voulons spécifier qu'une implantation peut avoir temporairement un comportement dégradé. Nous n'envisageons donc pas de garder la relation **ioco** pour définir la relation de robustesse. Cela s'explique en partie car les éléments de construction ne représentent plus les mêmes objectifs à vérifier. Nous proposons que la spécification dégradée joue un rôle de guide pour la génération des cas de test, que la propriété de robustesse devienne l'oracle de test pour définir une relation de satisfiabilité entre la propriété et l'implantation testée. Ces hypothèses nous permettront ensuite à définir les verdicts de robustesse.

En résumé et pour présenter de façon progressive notre prototype final (incluant une partie d'exécution des cas de test), nous proposons de réaliser :

Le "test de propriétés" en définissant :
- les classes et la modélisation des propriétés
- la méthode de génération de cas de test avec une spécification (un guide de construction) partielle et une propriété,
- la relation de satisfiabilité et le nouvel oracle.

Le "test de robustesse" en définissant :
- l'environnement dégradé,
- les fautes et l'intégration de ces nouvelles conditions dans les modèles de spécification,
- la définition d'un comportement acceptable,
- la relation de robustesse et les verdicts de robustesse.

Nous présentons maintenant un aperçu des différentes architectures pour générer les cas de test de propriétés et de robustesse.

Le test de propriétés

Le test de propriétés va nous permettre de mettre en place une méthode de génération de cas de test où la spécification initiale sert de guide à cette génération. Puis, nous donnons le modèle d'automate de Rabin paramétré (un observateur) pour décrire les propriétés de sûreté (safety) et de vivacité bornée (bounded liveness). Et enfin, nous présentons la relation de satisfiablité entre la propriété et l'implantation testée.

La figure 4 représente le principe de génération de cas de test de propriétés (où la spécification de référence peut être partielle).

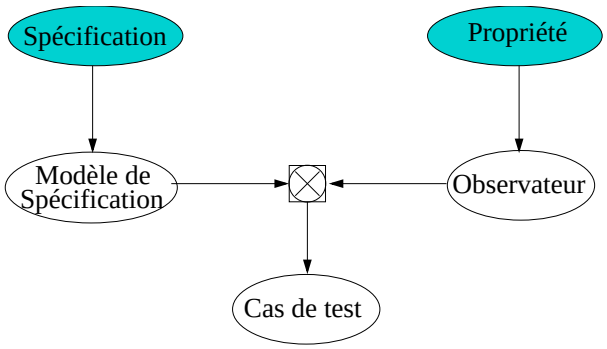

FIG. 4 – Génération de cas de test pour le test de propriétés

Le test de robustesse

Le test de robustesse est un test de propriétés où des comportements environnementaux sont donnés et intégrés au modèle de génération des cas de test. L'environnement tient compte des comportements de la propriété à préserver et d'un modèle de faute représentant les aléas et éventuellement d'autres actions à intégrer dans la spécification (figure 5). Les modèles pour décrire la spécification initiale sont les automates étendus communicants. Les mutations d'environnement sont intégrées au modèle par une fonction de mutation de comportement opérant sur une forme intermédiaire de la spécification. Les propriétés de robustesse sont modélisées par des automates de Rabin paramétrés complets (à chaque état de contrôle, toutes les actions d'entrée et de sortie sont possibles).

La figure 5 présente un principe de génération de cas de test de robustesse.

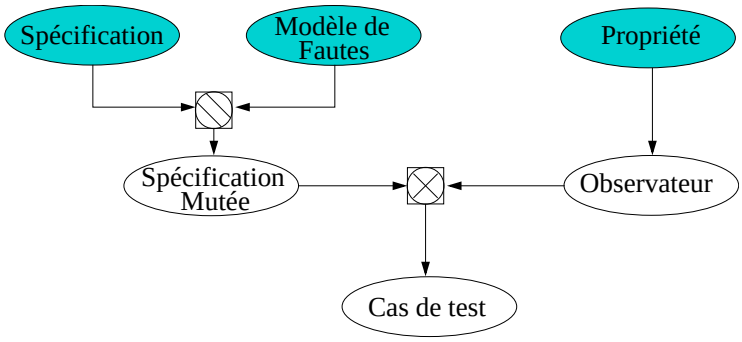

FIG. 5 – Génération de cas de test pour le test de robustesse

La méthode de test de robustesse, proposée dans cette thèse, est différente du principe de l'injection de faute, dans le sens, où la construction de séquences de test permet de guider les tests. Pendant toute la durée du test, le testeur garde un certain contrôle sur le test par l'intermédiaire des séquences qu'il exécute.

I.3 Plan du document

Nous décomposons le document en trois parties :

– La première partie présente des formalismes de description pour modéliser et manipuler les éléments de spécification. Cette partie comporte les chapitres suivants :

  Le Chapitre 1 introduit la syntaxe et la sémantique des systèmes de transitions étiquetées à entrées-sorties, et des automates étendus communicants. Ces modèles nous sont utiles pour représenter les comportements des programmes. Nous profitons de ce chapitre pour introduire des notations et certaines définitions nécessaires pour l'ensemble du document.

  Le Chapitre 2 propose un état de l'art sur le test.

  Le Chapitre 3 rappelle un principe de génération et d'exécution de cas de test de conformité. Ce chapitre apporte certaines bases de construction pour introduire les nouvelles approches de test basées sur les modèles.

– La deuxième partie propose les techniques et principes de génération et d'exécution de cas de test de propriétés et de robustesse. Cette partie comporte les chapitres suivants :

  Le Chapitre 4 propose un panorama des différentes techniques, principes et outils décrivant le test de robustesse.

Le Chapitre 5 présente une description complète pour générer des cas de test de propriétés. Il propose et définit des classes de propriétés. Il pose les ingrédients pour modéliser les propriétés, introduit le concept d'observateur et propose une nouvelle relation de test. Il génère des cas de test en tenant compte du fait que la propriété devient l'oracle d'exécution des tests.

Le Chapitre 6 présente les principes pour générer et exécuter des cas de test de robustesse basés sur le test de propriétés. Il introduit les modèles de faute et propose un opérateur pour enrichir de fautes les modèles de spécification. Il décrit la notion d'environnement nominal et propose la notion d'environnement dégradé

– La troisième partie décrit les algorithmes et un prototype (chaîne d'outils) pour le test de robustesse sur des programmes Java.

Le Chapitre 7 présente une plate-forme pour le test de robustesse et décrit les algorithmes mis en œuvre utilisant les théories introduites par les chapitres 5 et 6.

Chapitre 8 propose une chaîne complète d'outils pour générer et exécuter des cas de test de robustesse et explique les choix techniques. La chaîne d'outils est présentée étape par étape à travers un exemple.

Première partie

Le test de conformité

Chapitre 1

Modèles

Le but de ce chapitre est d'introduire les différents modèles de base servant à décrire, modéliser et manipuler les données de références, nécessaires à la génération automatique de cas de test, (comme par exemple pour décrire et représenter les comportements des programmes à tester, ou modéliser les spécifications initiales, etc).

Ces modèles sont d'une part les systèmes de transitions étiquetées (LTS pour Labelled Transition Systems, en anglais), et d'autre part, les systèmes de transitions étiquetées à entrées-sorties (IOLTS pour Input-Output Labelled Transition Systems). En effet, il est parfois commode de pouvoir distinguer les entrées des sorties pour décrire certains comportements, (comme par exemple pour modéliser l'échange de messages entre les éléments des systèmes étudiés). Les modèles LTS et IOLTS servent à décrire les comportements d'un système à l'exécution. Par contre, leurs représentations ne permettent pas de décrire les informations sur les types de données qu'ils manipulent. Pour cela, nous utiliserons les modèles d'automates étendus et les modèles d'automates étendus communicants.

Pour chacun de ces modèles, nous allons définir la syntaxe, la sémantique et ce, en termes du modèle de niveau précédent.

1.1 Systèmes de transitions

1.1.1 LTS : Systèmes de transitions étiquetées

Les modèles élémentaires, utilisés pour représenter les comportements d'un programme, sont, pour nous, les systèmes de transitions étiquetées LTS. L'association d'un LTS à un programme se fait par l'intermédiaire d'une sémantique opérationnelle qui est en général formalisée sous la forme d'un système de déduction. Un système de transitions étiquetées est défini, par un ensemble d'états, un ensemble d'étiquettes et une relation de transition paramétrée par l'ensemble des étiquettes. Les états peuvent caractériser des points de contrôle du programme, et contiennent les valeurs des différentes variables (en ces points). La relation de transition représente l'évolution du programme.

Une transition représente une opération abstraite effectuée par le système lors du changement d'état. De façon syntaxique, nous décrivons un LTS de la manière suivante :

Définition 1.1.1 (LTS : syntaxe) *Un système de transitions étiquetées est un quadruplet* $(Q, A, \longrightarrow, q^{init})$ *où :*

- Q *est l'ensemble fini des états,*
- q^{init} *est un état de* Q *(état initial),*
- A *est l'ensemble des étiquettes,*
- $\longrightarrow$ *est la relation de transition reliant les états, c'est une relation ternaire incluse dans :* $Q \times A \times Q$.

NOTE :
Pour les définitions qui suivent et qui utilisent des systèmes de transitions étiquetées, nous considérons qu'un LTS est toujours représenté par : $(Q, A, \longrightarrow, q^{init})$, sauf mention contraire, et dans ce cas il est explicité.

Quelques notations sur les LTS :
- La notation $(q, a, q') \in \longrightarrow$, indique qu'une transition étiquetée par une action $a \in A$ permet d'atteindre l'état $q^{'} \in Q$ en partant de l'état $q \in Q$: elle est notée par $q \xrightarrow{a} q'$.
- La notation $q \xrightarrow{a}$, indique qu'une transition étiquetée par l'action a est possible depuis l'état q : $\exists\, q^{'} \in Q$, $\mid q \xrightarrow{a} q'$.
- La notation $q \longrightarrow$, indique qu'une transition est possible depuis l'état q : $\exists\ q^{'} \in Q$, $\exists\, a \in A \mid q \xrightarrow{a} q'$.
- La notation $q \not\xrightarrow{a}$, indique qu'aucune transition étiquetée par l'action a n'est possible à partir de l'état q : $\nexists\, q^{'} \in Q$ tel que $q \xrightarrow{a} q'$.
- La notation $q \not\longrightarrow$, indique qu'aucune transition n'est possible en partant de l'état q : $\forall a \in A$, $\nexists\, q^{'} \in Q \mid q \xrightarrow{a} q'$.

NOTE :
Un état $q \in Q$ qui ne possède aucune transition sortante est appelé un état puits.

Quelques définitions sur les LTS :

Nous définissons la fonction Pre_a qui étant donné un ensemble d'états X calcule l'ensemble des prédécesseurs des états de X par une action a :
- $Pre_a : 2^Q \longrightarrow 2^Q$
- $Pre_a(X) = \{q \mid \exists\, q^{'} \in X \; et \; q \xrightarrow{a} q^{'}\}$

Nous définissons la fonction $Succ_a$ qui étant donnée un ensemble d'états X calcule l'ensemble des successeurs des états de X par une action a :
- $Succ_a : 2^Q \longrightarrow 2^Q$
- $Succ_a(X) = \{q \mid \exists\, q^{'} \in X \; et \; q^{'} \xrightarrow{a} q\}$

Définitions des symboles * et $^\omega$ pour les ensembles d'actions :

Pour un ensemble fini X, nous notons par X^* l'ensemble des séquences finies sur X et par $X^\omega = X^{\mathbb{N}}$ l'ensemble des séquences infinies sur X (X^ω est l'ensemble des fonctions de $\mathbb{N}$ dans X). Pour toute séquence x, nous notons par $x(i)$ ou x_i son i^{eme} élément et par $\mid x \mid$ sa longueur.

À partir des LTS, nous pouvons définir différentes notions comme les séquences, les chemins et les séquences d'exécution :

Définition 1.1.2 (LTS : séquence d'exécution) *Soit $q \in Q$.*

1. *Une séquence finie d'exécution σ issue de l'état q est un élément de $\longrightarrow^*$ tel que pour tout $i \in [0, \mid \sigma \mid]$:*

   Si $\sigma(i) = (q_i, a_i, q_i^{'})$, $\sigma(i+1) = (q_{i+1}, a_{i+1}, q_{i+1}^{'})$ alors $q'_i = q_{i+1}$.

   Nous notons alors : $\sigma = q \xrightarrow{a_1} q_1 \xrightarrow{a_2} \ldots \xrightarrow{a_n} q_n$, si $\mid \sigma \mid = n$.

2. *Une séquence infinie d'exécution σ issue de l'état q est un élément de $\longrightarrow^\omega$ tel que pour tout i :*

   Si $\sigma(i) = (p_i, a_i, p_i^{'})$, $\sigma(i+1) = (p_{i+1}, a_{i+1}, p_{i+1}^{'})$ alors $q_i^{'} = q_{i+1}$.

   Nous notons alors : $\sigma = q \xrightarrow{a_1} q_1 \xrightarrow{a_2} \ldots \xrightarrow{a_n} \ldots, q_n \xrightarrow{a_{n+1}} \ldots$

Définition 1.1.3 (LTS : chemin associé à une séquence d'exécution)

1. *Soit $\sigma = q_0 \xrightarrow{a_1} q_1 \xrightarrow{a_2} \ldots \xrightarrow{a_n} q_n$ une séquence finie d'exécution, le chemin associé à σ, noté Chemin(σ), est la projection de σ sur Q :*

   Chemin(σ)= $q_0 q_1 \ldots$ si σ est une séquence infinie (Chemin(σ) $\in Q^\omega$).

   Chemin(σ)= $q_0\ q_1 \ldots\ q_n$ si σ est une séquence finie (Chemin(σ) $\in Q^*$).

Définition 1.1.4 (LTS : trace d'une séquence d'exécution) *Soit σ une séquence d'exécution, la trace de σ, notée Trace(σ), est la projection de la séquence d'exécution σ sur A :*

Trace(σ)= $a_0 a_1 \ldots$ si σ est une séquence infinie (Trace(σ) $\in A^\omega$).

Trace(σ)= $a_0 a_1 \ldots\ a_n$ si σ est une séquence finie (Trace(σ) $\in A^*$).

Si nous notons par $\rho \in Q^\omega$, la séquence infinie d'états = $q\ q_1\ q_2 \ldots\ q_n \ldots$ alors, inf(ρ) dénote l'ensemble des états de Q qui apparaissent infiniment souvent dans ρ : inf(ρ) = $\{q \mid \forall n\ \exists m$, $m > n$ et $\rho(m) = q\ \}$.

Il est parfois utile de raisonner en terme de langage. En projetant les séquences d'exécution sur l'ensemble des actions d'un LTS, nous définissons un langage associé à un état et un langage associé à un système.

Définition 1.1.5 (LTS : langage fini associé à un état, à un LTS) *Soit $q \in Q$ un état, le langage fini associé à l'état q est : $\mathcal{L}(q) = \{ \alpha \in A^* \mid \exists\, q_1 q_2 \ldots q_n \wedge q \xrightarrow{\alpha_1} q_1 \xrightarrow{\alpha_2} \ldots \xrightarrow{\alpha_n} q_n \wedge \alpha = \alpha_1\alpha_2 \ldots \alpha_n\}$ (c'est l'ensemble des traces finies issues de l'état q). Le langage fini associé à un LTS est le langage associé à son état initial.*

Définition 1.1.6 (LTS : langage infini associé à un état, à un LTS) *Soit $q \in Q$ un état, le langage infini associé à l'état q est : $\mathcal{L}^\omega(q) = \{ \alpha \in A^\omega \mid \exists\, q_1 q_2 \ldots \wedge q \xrightarrow{\alpha_1} q_1 \xrightarrow{\alpha_2} \ldots \wedge \alpha = \alpha_1\alpha_2 \ldots\}$ (c'est l'ensemble des traces infinies issues de l'état q). Le langage infini associé à un LTS est le langage associé à son état initial.*

Notations et définitions sur les langages :

Par définition, un langage est dit régulier, s'il existe une grammaire régulière qui le produit. Un ensemble L $\subseteq A^\omega$ est ω-regulier si :

$$L = \bigcup_{i=1}^{k} U_i V_i^\omega$$

où U et V sont des langages réguliers.

Exemples :
- $(a+b)^*a^\omega$: ensemble des séquences infinies sur $\{a,b\}$ ne contenant qu'un nombre fini de b.
- $(a^*b)^\omega$: ensemble des séquences infinies $\{a,b\}$ contenant un nombre infini de b (complémentaire du précédent).

Définition 1.1.7 (LTS : état atteignable) *Un état $q \in Q$ est atteignable dans un LTS, si il existe un chemin allant de q^{init} (état initial du LTS) à q.*

Définition 1.1.8 (LTS : compatible) *$LTS_1 = (Q_{LTS_1}, A_{LTS_1}, \longrightarrow_{LTS_1}, q^{init}_{LTS_1})$ et $LTS_2 = (Q_{LTS_2}, A_{LTS_2}, \longrightarrow_{LTS_2}, q^{init}_{LTS_2})$ sont compatibles si les ensembles A_{LTS_1} et A_{LTS_2} sont comparables. Les deux ensembles A_{LTS_1} et A_{LTS_2} sont comparables si : $A_{LTS_1} \subseteq A_{LTS_2}$ ou $A_{LTS_2} \subseteq A_{LTS_1}$.*

Dans la suite, les LTS que nous considérons ont la propriété que tous les états q du LTS sont atteignables.

1.1.2 IOLTS : Systèmes de transitions étiquetées à entrées-sorties

Dans la suite de notre document, nous sommes amenés à utiliser et donc à décrire des systèmes complexes faisant intervenir des modes de communication. De plus, la nature asymétrique de l'activité de test nous oblige, dans les systèmes étudiés, à distinguer les actions d'entrées de celles de sorties. Or, une limitation des LTS est qu'ils ne permettent pas de distinguer la nature des actions d'échange de messages. Nous introduisons les systèmes de transitions étiquetées à entrées-sorties

ou IOLTS. Un IOLTS est décrit comme un LTS $(Q, A_\tau, \longrightarrow, q^{init})$ dans lesquels les actions d'entrée et les actions de sortie de messages sont distinguées. Ainsi, nous considérons que l'alphabet fini des actions de A_τ est partitionné en sous-ensembles d'actions : *les actions d'entrée* A^I, *les actions de sortie* A^O et *les actions internes* $\{\tau\}$.

Les actions internes sont étiquetées par un symbole particulier τ. Ces actions τ sont non observables, car invisibles pour l'environnement du système (ou pour un testeur) par opposition aux autres actions de A^I et A^O visibles par ce même environnement.

Définition 1.1.9 (IOLTS : Syntaxe) *Un système de transitions étiquetées à entrées-sorties IOLTS est un LTS* $(Q, A, \longrightarrow, q^{init})$ *où :*
- Q *est l'ensemble fini des états,*
- q^{init} *est l'état initial,*
- A *est l'ensemble fini des actions partitionné en sous-ensembles : l'ensemble des actions de sortie* A^O*, l'ensemble des actions d'entrée* A^I *et l'ensemble des actions internes* $\{\tau\}$ *:* $A = A^I \cup A^O \cup \{\tau\}$ *.*
- $\longrightarrow \subseteq Q \times A_\tau \times Q$ *est la relation de transition.*

Pour manipuler les séquences produites en termes de IOLTS, nous complétons les définitions précédentes (sur les LTS) avec un certain nombre de nouvelles notations.

Notations et conventions sur les IOLTS : Soit M = $(Q, A, \longrightarrow, q^{init})$ un IOLTS, $q, q', p, p_0, p_1, \ldots, p_i, p_{i+1}, p_n \in Q$ des états, et a, $\alpha_i \in A$ des actions visibles, $\alpha \in A^*$ une séquence d'actions visibles, et $n, m \in \mathbb{N}$ des entiers :

- La notation $p \xrightarrow{\tau^* a} q$ indique qu'un état q est accessible à partir d'un état p par l'intermédiaire d'une séquence d'exécution dont les transitions sont étiquetées par τ suivie d'une transition étiquetée par l'action a : $p = p_0 \xrightarrow{\tau} p_1 \xrightarrow{\tau} \ldots \xrightarrow{\tau} p_n \xrightarrow{a} q$,

- La notation $p \xRightarrow{\alpha} q$ indique qu'un état q est accessible à partir d'un état p par l'intermédiaire d'une séquence d'exécution de trace α.

- $Act(q) = \{\alpha \in A^* \mid \exists q' \text{ et } q \xrightarrow{\tau^* \alpha} q'\}$ est l'ensemble des actions visibles après un état q parmi lesquelles :

  $\mathcal{I}(q) = Act(q) \cap A^I$ est le sous-ensemble des entrées après l'état q,
  $\mathcal{O}(q) = Act(q) \cap A^O$ est le sous-ensemble des sorties après l'état q.

- Nous étendons cette notion aux ensembles d'états. Pour P $\subseteq Q$ un ensemble d'états, nous avons :

  $Act(P) = \bigcup_{q \in P} Act(q)$,
  $\mathcal{I}(P) = \bigcup_{q \in P} \mathcal{I}(q) = Act(P) \cap A^I$,
  $\mathcal{O}(P) = \bigcup_{q \in P} \mathcal{O}(q) = Act(P) \cap A^O$.

– La notation $p \stackrel{a}{\Longrightarrow}$, indique qu'une action $a \in$ A est possible, à partir de l'état $p \in Q$, après un certain nombre d'actions internes : $\exists p^{'} \in Q$ tel que $p \stackrel{a}{\Longrightarrow} p^{'}$ où $a \in Act(p)$.

– La notation $p \stackrel{a}{\not\Longrightarrow}$, indique que l'action a n'est pas possible à partir de l'état $p \in Q$: $\nexists p^{'} \in Q$ tel que $p \stackrel{a}{\Longrightarrow} p^{'}$ ou $a \notin Act(p)$.

– La notation $p \Longrightarrow$, indique qu'une action est possible à partir de l'état $p \in Q$: $\exists\ a \in A$ tel que $p \stackrel{a}{\Longrightarrow}$ ou $Act(p) \neq \emptyset$.

– La notation $p \not\Longrightarrow$, indique qu'aucune action n'est possible à partir de l'état $p \in Q$: $\forall\ a \in A$ tel que $p \stackrel{a}{\not\Longrightarrow}$ ou $Act(p) = \emptyset$.

– L'ensemble des états accessibles depuis un état après (*after*) une trace :
q after α = { q$^{'}$ $\in$ Q | $q \stackrel{\alpha}{\Longrightarrow} q^{'}$} avec $\alpha \in$ A*.

– L'ensemble des états accessibles depuis un ensemble d'états après une trace :
P after α = $\bigcup_{q \in P}$ q after α, pour un ensemble d'états P.

– L'ensemble des états accessibles pour un système de transitions M est l'ensemble des états accessibles depuis son état initial.

– L'ensemble des séquences d'exécution pour un système de transitions M est l'ensemble associé aux séquences d'exécutions à partir de l'état initial de M.

– Pour parler de comportement interne (ou non visible) nous définissons :
$q \stackrel{\epsilon}{\Longrightarrow} q^{'} \equiv q = q^{'} \vee q \stackrel{\tau_1,\tau_2,\ldots,\tau_n}{\longrightarrow} q^{'}$ où ϵ dénote la séquence vide de Act^*.

Définition 1.1.10 (Opérateur de projection) *Soit V un sous-ensemble de l'alphabet A. Nous définissons l'opérateur de projection $\downarrow V : A^* \longrightarrow V^*$ de la manière suivante :*

$$\epsilon \downarrow V = \epsilon,$$
$$(a.\alpha) \downarrow V = \alpha \downarrow V \text{ si } a \notin V,$$
$$(a.\alpha) \downarrow V = a.(\alpha \downarrow V) \text{ si } a \in V.$$

Cet opérateur peut être étendu au langage L noté $L\downarrow V$ en appliquant l'opérateur sur chaque séquence de L.

Définition 1.1.11 (IOLTS : chemin et trace associés à une séquence d'exécution)

1. *Soit $\sigma = q_0 \stackrel{a_1}{\Longrightarrow} q_1 \stackrel{a_2}{\Longrightarrow} \ldots \stackrel{a_n}{\Longrightarrow} q_n$ une séquence finie d'exécution, le chemin associé à σ est Chemin(σ) = $q_0\ q_1 \ldots\ q_n$ et la trace finie associée à σ est : Trace(σ) = $a_0 a_1 \ldots\ a_n \in A^*$.*

2. *Soit* $\sigma = q_0 \stackrel{a_1}{\Longrightarrow} q_1 \stackrel{a_2}{\Longrightarrow} \ldots \stackrel{a_n}{\Longrightarrow} q_n \stackrel{a_{n+1}}{\Longrightarrow} \ldots$ *une séquence infinie d'exécution, le chemin associé à* σ *est Chemin(*σ*)* $= q_0\ q_1\ \ldots\ q_n\ \ldots$ *et la trace infinie associée à* σ *est : Trace(*σ*)* $= a_0 a_1 \ldots \in A^\omega$.

De façon générale, Trace(σ*)* $\in A^*$ *(ou* $\in A^\omega$*) est la projection sur les étiquettes (ou actions observables différentes des actions* τ*) de la séquence d'exécution* σ.

Définition 1.1.12 (IOLTS : Trace d'un IOLTS) *Soit M un IOLTS et Execution(M) l'ensemble de toutes les séquences d'exécution de M, alors Trace(M) =* $\{$*Trace(*σ*)* $\mid \sigma \in$ *Execution(M)*$\}$.

Plusieurs propriétés peuvent être définies sur la structure d'un IOLTS :

Un système de transitions étiquetées (à entrées-sorties) est considéré comme un système complet si pour n'importe quel état du système, toutes les actions lui sont possibles.

Définition 1.1.13 (IOLTS : complet) *Un* $IOLTS$ *M = (Q, A,* $\longrightarrow$*,* q^{init}*) est complet sur l'alphabet des actions* A*, si et seulement si, dans chaque état* $q \in Q$ *de M, aucune action ne peut être refusée :* $\forall q \in Q, \forall a \in A, q \stackrel{a}{\Longrightarrow}$.

Pour certains IOLTS, nous aurons besoin de supposer qu'il est complet en entrée (pour n'importe quel état du système, une action d'entrée est toujours possible).

Définition 1.1.14 (IOLTS : complet en entrée) *Un* $IOLTS$ *M = (Q, A,* $\longrightarrow$*,* q^{init}*) est complet en entrée sur l'alphabet des actions d'entrée* A^I*, si et seulement si, dans chaque état* $q \in Q$ *de M, il ne peut refuser aucune action d'entrée après d'éventuelles actions internes :* $\forall q \in Q, \forall a \in A^I, q \stackrel{a}{\Longrightarrow}$.

Si le IOLTS M est complet en entrée nous avons : Traces(M).$A^I \subseteq$ Traces(M) où Traces(M) sont les traces finies de M.

> **Remarque :** *La définition d'un IOLTS complet en entrée est celle définie par Tretmans [Tre96]. Elle est moins exigeante que celle de Lynch [Lyn88] qui est définie par :* $\forall\ q \in Q, \forall\ a \in A, q \stackrel{a}{\longrightarrow}$.

Un système de transitions étiquetées à entrées-sorties est considéré (en interprétant les actions internes τ comme des ϵ-transitions) comme un système déterministe si :

Définition 1.1.15 (IOLTS : déterministe) *Un IOLTS est* déterministe *si et seulement si il n'a aucune transition étiquetée par une action* τ *(action interne) et, pour chaque état* $p \in Q$*, il existe au plus une seule transition étiquetée par une même action* $a \in A^O \cup A^I$ *:*

$$\forall a.\ p \stackrel{a}{\Longrightarrow} p' \wedge p \stackrel{a}{\Longrightarrow} p'' \Longrightarrow p' = p'' \text{ avec } p, p', p'' \in Q.$$

Exemple 1 — **[Représentation de IOLTS non déterministes]**
Cet exemple représente deux IOLTS non déterministes. Nous profitons de cet exemple pour compléter notre notation. Les actions d'entrée (resp. de sortie) sont distinguées par le symbole ? (resp. le symbole !). Sur la figure 1.1 l'action d'entrée x est notée ?x, l'action de sortie a est notée !a, et τ est une action interne.

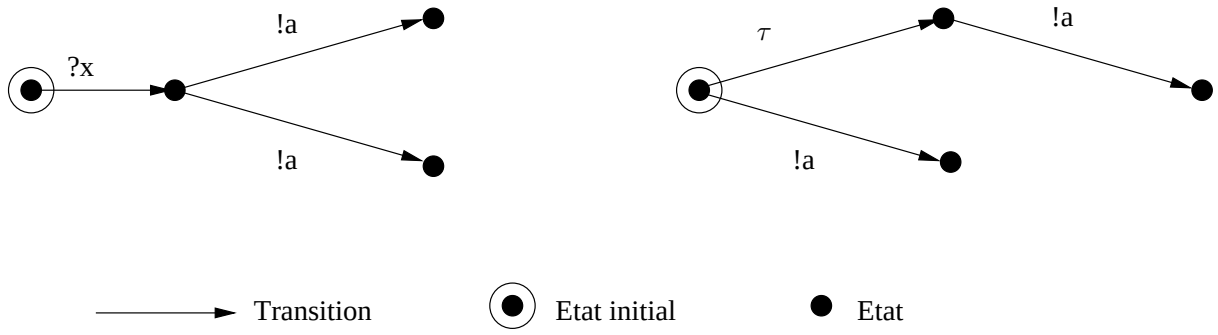

FIG. 1.1 – Exemple de systèmes non déterministes.

Définition 1.1.16 (IOLTS : contrôlabilité) *Un IOLTS satisfait la condition de contrôlabilité si et seulement si, il est déterministe, il ne contient pas d'action τ et pour chaque état où une sortie est possible, alors, il n'y a qu'une transition sortante :*

$$\forall\, p \in Q \;\; . \;\; \mid \mathcal{O}(p) \mid = \; 0 \;\; \vee \;\; (\mid \mathcal{O}(p) \mid = 1 \;\; \wedge \;\; \mathcal{O}(p) = Act(p)).$$

Dans le cas, où il existe un état ayant plusieurs sorties possibles (même après un nombre quelconque d'actions internes), nous parlons de situation de "non déterminisme". Plus précisément, nous parlons de "non déterminisme observable" pour décrire l'existence d'un choix non contrôlé par l'environnement.

Définition 1.1.17 (non déterminisme observable) *Un IOLTS n'a pas de non déterminisme observable si dans tout état $q \in Q$: $\mid \mathcal{O}(q \text{ after } \epsilon) \mid \leq 1$.*

Exemple 2 — **[Représentation du non déterministe observable pour les IOLTS]**
La figure 1.2 représente des cas possibles de non déterminisme observable sur des IOLTS. Dans celui de gauche, les actions !a et !b partant du deuxième état sont deux actions de sortie. Dans celui de droite, τ étant une action interne, les actions partant de l'état initial peuvent être alors les deux actions de sortie !a et !b.

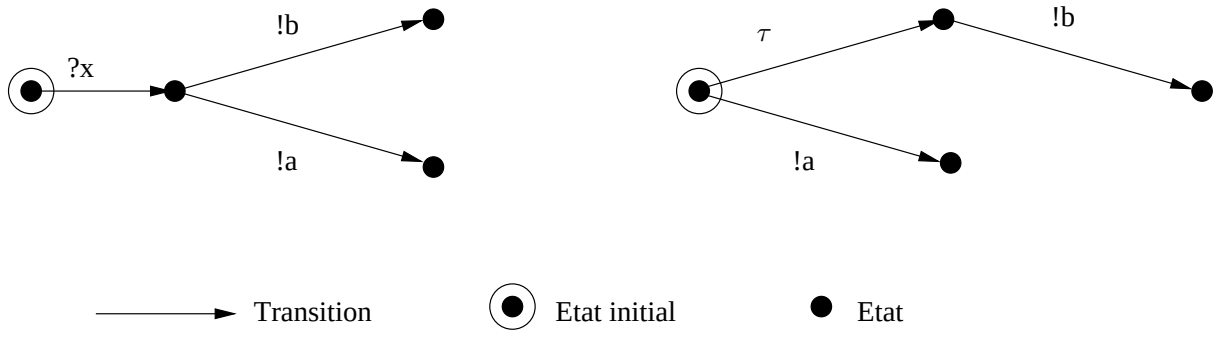

FIG. 1.2 – Exemple de non déterministe observable au sens des IOLTS.

Les modèles comme les LTS (section 1.1.1) et les IOLTS (section 1.1.2) ne permettent pas de décrire des transformations sur les types de données. C'est pourquoi nous utilisons des modèles de plus haut niveau, dans lesquels ces transformations sont explicites.

1.2 Les automates étendus et communicants

Pour garder et manipuler les données des systèmes étudiés, nous introduisons les automates étendus. La plupart des systèmes (programmes à tester) ont souvent plusieurs composants dialoguant ensemble. Pour les représenter, nous utilisons les automates étendus communicants.

1.2.1 Les automates étendus

Les automates étendus permettent de manipuler des variables typées. Dans ce cas, les valeurs de ces différentes variables peuvent être testées, modifiées, ou échangées avec leur environnement à travers des files d'attente externes. Nous donnons la sémantique de ces automates étendus AE en terme de LTS.

1.2.1.1 Syntaxe

Définition 1.2.1 (Syntaxe des automates étendus) *Un automate étendu est un tuple* $(T, X, S, C^{ext}, Q, \longrightarrow, q^0)$ *où :*

– T *est l'ensemble fini des types de données. Un ensemble d'opérateurs est défini sur chacun des types, et domaine(*t*) est l'ensemble des valeurs du type* $t \in T$.

– X *est l'ensemble des variables locales typées. Nous notons par type(*x*)* $\in T$*, le type de la variable* $x \in X$*. Nous représentons par Exp(*X*) l'ensemble des expressions* e *construites à*

partir des variables $x \in X$ et des opérations existantes sur les types de x. Chaque expression $e \in$ Exp(X) a un type unique calculé statiquement, noté : type(e). Nous représentons par Aff[X] = $\{ \{ x := e\} \mid x \in X, e \in$ Exp(X) $\}$ l'ensemble des affectations, et par BExp(X) l'ensemble des expressions booléennes [b] (ou gardes). Une expression [b] est construite à partir d'opérations relationnelles sur des expressions de Exp(X), de conjonction, de disjonction, de négation d'expressions (booléennes) et des variables booléennes : BExp(X) = $\{$ [b], [b_1], ..., [b_n]$\}$.

– *S est un ensemble de signaux typés. Nous notons par type(x) $\in T$ le type (profil) du signal $s \in S$. Ainsi toutes les réceptions et les émissions du signal s seront typées par une valeur du type de s.*

– *C^{ext} est l'ensemble des files d'attente externes. Une entrée de la file est symbolisée par ?. Une entrée $c?s(x)$ est réalisable à l'aide d'un signal $s \in S$ et transite via la file d'attente $c \in C^{ext}$. L'ensemble des entrées est In[X,C^{ext},S] = $\{ \{ c?s(x)\} \mid c \in C^{ext}, s \in S, x \in X \}$. Une sortie de la file est symbolisée par !. Une sortie $c!s(e)$ permet l'envoi de $e \in$ Exp(X), par un signal $s \in S$ via une file d'attente $c \in C^{ext}$. L'ensemble des sorties est Out[X,C^{ext},S] = $\{ \{ c!s(e)\} \mid c \in C^{ext}, s \in S, e \in$ Exp(X)$\}$.*
L'ensemble des actions d'échanges est AcInOut[X,C^{ext},S] = In[X,C^{ext},S] $\cup$ Out[X,C^{ext},S]. Toutes les actions de cet ensemble sont typées selon le type de la variable ou de l'expression échangée.

Les entrées, les sorties et les expressions peuvent être composées avec les gardes de BExp(X) :

$$\text{pour } [b] \times \begin{cases} x := e & \text{nous notons} \quad [b]\ x := e, \\ c!s(e) & \text{nous notons} \quad [b]\ c!s(e), \\ c?s(x) & \text{nous notons} \quad [b]\ c?s(x), \end{cases}$$

– *Q est l'ensemble des états,*
– *q^0 est l'état initial,*

– *$\longrightarrow \subseteq Q \times$ BExp[X] $\times$ (Aff[X] $\cup$ AcInOut[X,C^{ext},S]) $\times Q$ est l'ensemble des transitions.*

NOTE :
Nous précisons que dans la syntaxe des automates communicants présentée, nous autorisons qu'une seule émission ou réception par transition.

1.2.1.2 Sémantique

Nous définissons comme contexte sur les variables de X toute application $\rho : X \longrightarrow \bigcup_{t \in T}$ domaine(t) qui associe à toute variable x la valeur $v = \rho(x)$ dans le domaine de x.
Le contexte sur les variables peut être étendu pour toutes les expressions $e \in \text{Exp}[X]$ utilisées :

- Nous notons par $v = \rho(e)$ la valeur de l'expression $e \in \text{Exp}[X]$ dans le contexte ρ.
- Soit $r \in \text{Aff}[X]$ une affectation partielle des variables de X. Nous notons $\rho[r]$ le nouveau contexte obtenu à partir du contexte ρ en appliquant l'affectation r. Formellement nous avons : si $r = \{x_1 := e_1, \ldots, x_n := e_n\}$ alors $\rho[r] = \rho[\rho(e_1)/x_1, \ldots, \rho(e_n)/x_n]$.

Définition 1.2.2 (Sémantique des automates étendus) *Soit AE = (T, X, S, C^{ext}, Q, $\longrightarrow$, q^0) un automate étendu. La sémantique des automates étendus est donnée en terme de LTS : ($Q^\bullet$, $A^\bullet$, $\longrightarrow^\bullet$, q^0)*

- *$A^\bullet$ est l'ensemble des actions : $C^{ext} \times \{ ?, ! \} \times S^\bullet$.*
- *$S^\bullet = \{s(v) \mid s \in S$, type(s) = (t), $v \in$ domaine(t)$\}$.*
- *$Q^\bullet = \{(\rho, q) \mid q \in Q, \rho : X \longrightarrow \bigcup_{t \in T}$ domaine(t)$\}$. Les états sont des couples (contexte sur les variables, état de contrôle).*
- *$q^0 \in Q^\bullet$ où ρ est fixé initialement.*

Nous disposons de deux modes pour interpréter les transitions $\longrightarrow^\bullet$:
- *Les transitions sont* **internes** *: Elles sont étiquetées par l'action τ, quand elles se dérivent des actions internes.*
- *Les transitions sont* **visibles** *: Elles sont étiquetées avec des actions concrètes, quand elles se dérivent de communications externes.*

$\longrightarrow^\bullet$ est définie par les règles suivantes, avec ρ est un contexte pour les variables, et $q, q' \in Q^\bullet$ sont des états de contrôle :

$$\frac{q \xrightarrow{[b]\, x := e} q' \quad \rho([b]) = \mathbf{t},\ v = \rho(e)}{(\rho,\ q) \xrightarrow{\tau} (\rho[v/x],\ q')} \qquad \text{[T1]}$$

$$\frac{q \xrightarrow{[b]\ c!s(e)} q' \quad \rho([b]) = \mathbf{t},\ v = \rho(e),\ c \in C^{ext},\ s \in S}{(\rho,\ q) \xrightarrow{c!s(v)} (\rho,\ q')} \qquad \text{[T2]}$$

$$\frac{q \xrightarrow{[b]\ c?s(x)} q' \quad \rho([b]) = \mathbf{t},\ c \in C^{ext},\ v \in (domaine(type(x)),\ s \in S}{(\rho,\ q) \xrightarrow{c?s(v)} (\rho[v/x],\ q')} \qquad \text{[T3]}$$

La règle [T1] évalue l'expression e dans le contexte ρ et produit la valeur v. Cette évaluation est considérée comme une transition interne au niveau de l'automate étendu. Dans le contexte ρ, la transition est possible si et seulement si la valeur de la garde [b] est évaluée à vrai (t = true).

La règle [T2] définit l'envoi de la valeur v, résultat de l'évaluation de l'expression e dans le contexte ρ. La valeur v est contenue dans une file d'attente c et l'envoi est fait via un signal s. Dans le contexte ρ, l'envoi est possible si et seulement si la valeur de la garde [b] est évaluée à vrai (t = true).

La règle [T3] définit la réception via la file d'attente externe c d'une nouvelle valeur v de la variable x. La réception est possible si et seulement si la valeur de la garde [b] est évaluée à vrai (t = true).

Exemple 3 — **[Représentation d'un automate étendu]**
La figure 1.3 représente un automate étendu AE. L'automate est défini par les ensembles {de types}, {files d'attente extérieures}, {états}, {signaux}, {actions} et un {état initial}. AE = ({ $\mathbb{Z}, \ldots$ }, {f1, f2}, { 0, 1, 2, 3, 4, 5 }, {$s1$, $s2$}, {f1 !$s1$(x), [x<5] f2 !$s2$(10), x := 5, f2 !$s2$(3+4), x := y, [x<3], f2 ?$s2$(x), [x>3] f1 ?$s1$(x)}, 0).

L'action entre l'état 0 et l'état 4 est de recevoir de l'environnement la valeur de la variable x via la file f1 par le signal $s1$. Une condition évaluant la garde [x<5] permet de faire l'action d'envoi de l'entier 10 sur la boucle état 4 état 4, via la file f2 sur le signal $s2$. Une action d'affectation x := 5 est alors possible en prenant la transition de l'état 4 à l'état 5.
L'action entre l'état 0 et l'état 1 affecte à la variable x la valeur de la variable y. L'évaluation de la garde [x<3] à vrai permet d'aller à l'état 3. Sinon il y a un envoi de la valeur de la variable x via la file f1 par le signal $s1$.
L'action entre l'état 0 et l'état 3 envoie la valeur de l'expression 3 + 4 via la file f2 par le signal s2, puis pour atteindre l'état 1 à partir de l'état 3, il y a une réception de la valeur de la variable x via la file f2 avec le signal $s2$.

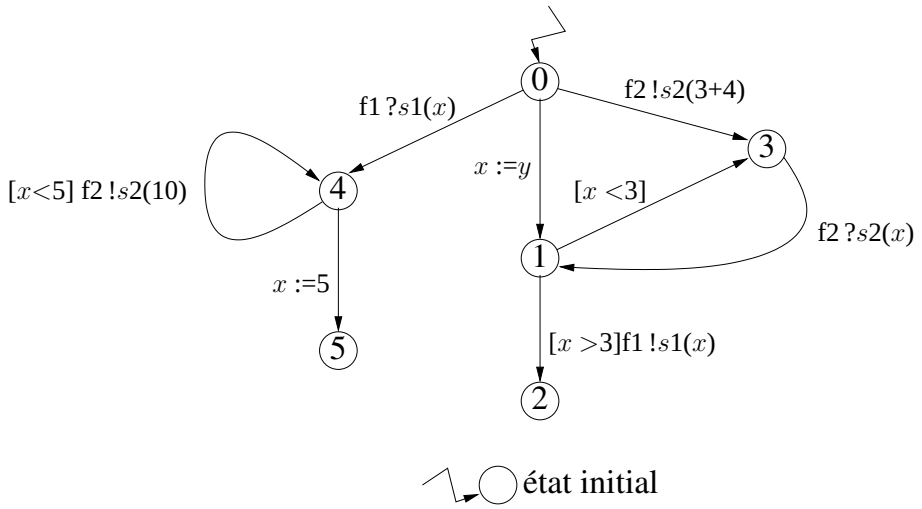

FIG. 1.3 – Représentation d'un automate étendu

Nous ne limitons pas nos systèmes à un seul automate. La plupart des logiciels sont décrits sous la forme de composants communicants (dont chaque composant est représenté par un automate). Pour modéliser le comportement de ces systèmes et leurs composants, nous introduisons les automates étendus communicants.

1.2.2 Les automates étendus communicants

Les automates étendus communicants gardent leur intérêt sur la manipulation des données. Ils rajoutent le fait que les données peuvent être échangées, modifiées et testées par communication d'automates via des files internes et envoyées comme des paramètres.

1.2.2.1 Syntaxe

Définition 1.2.3 (Syntaxe des automates étendus communicants) *La syntaxe d'un automate étendu communicant est sensiblement la même que celle d'un automate étendu avec :*

– *S est un ensemble de signaux typés, type(x) $\in T$ est le profil du signal $s \in S$, les réceptions et les émissions du signal s seront typées par une valeur du type de s.*

– *$C = C^{int} \cup C^{ext}$ est un ensemble de files d'attente composé de files internes et de files externes. Les files permettent l'échange de messages entre automates (files internes) et/ou entre les automates et leur environnement (files externes). Toutes les entrées (resp. les sorties) c ?s(x) (resp. c !s(e)) sont réalisables à l'aide d'un signal $s \in S$ et transite via la file d'attente $c \in C$. L'ensemble des actions d'entrée de l'environnement est In[X,C,S] = $\{ c?s(\overrightarrow{x}) \mid c \in C^{ext}, s \in S, \overrightarrow{x} \in X* \}$, dans la construction $c?s(\overrightarrow{x})$, $\overrightarrow{x}$ désigne*

le paramètre transporté par le signal s. L'ensemble des actions de sortie dans l'environnement est Out[X,C,S] = $\{c\,!\,s(e) \mid c \in C^{ext}, s \in S, e \in Exp[X]\}$. L'ensemble des actions est : In[X,C,S] $\cup$ Out[X,C,S].

1.2.2.2 Sémantique

$\rho : X \longrightarrow \bigcup_{t \in T}$ domaine(t) associe à chaque variable x une valeur v du domaine T. Le contexte ρ est étendu aux expressions avec : $\rho(e)$ est la valeur de l'expression $e \in$ Exp[X]. si $r = \{x_1 := e_1, \ldots, x_n := e_n\}$ alors $\rho[r] = \rho[\rho(e_1)/x_1, \ldots, \rho(e_n)/x_n]$.

$\delta : C \longrightarrow (S \times T)^*$ qui associe à chaque file d'attente interne c une séquence $(s_1, v_1), .., (s_k, v_k)$ de messages définissant les couples (s, v), noté $s(v)$, avec s un signal et v la valeur du paramètre, pour une file c la séquence est notée $\delta(c)$. Les messages indéfinis sont notés $\perp$. La séquence vide est notée ε.

Définition 1.2.4 (Sémantique des automates étendus communicants)
Soit AEC = (T, X, S, C, Q, $\longrightarrow$, q^0) un automate étendu communicant. La sémantique de cet automate est donnée en termes de LTS = ($Q^\bullet$, $A^\bullet$, $\longrightarrow^\bullet$, q^0) :

- *$A^\bullet$ est l'ensemble des actions : $C \times \{\,?, !\} \times S^\bullet$.*
- *$S^\bullet$ = $\{s(v_1, ..., v_n) \mid s \in S$, type(s) = $(t_1, ..., t_n)$, $\forall i = 1, n$, $v_i \in$ domaine(t_i)$\}$.*

Un état du LTS est décrit par un triplet (ρ, δ, θ) où :
- *ρ est un contexte pour les variables et les expressions,*
- *δ est un contexte pour les files d'attente,*
- *$\theta = \langle q_0, .., q_n \rangle \subseteq \times Q$ est un état de contrôle global,*

- *$Q^\bullet$ = $\{(\rho,\delta,\theta) \mid \rho : X \longrightarrow \bigcup_{t \in T}$ domaine(t), $\delta : C \longrightarrow (S \times T)^*\}$.*
- *$q^0 \in Q^\bullet$ où ρ et δ sont fixés initialement.*

Les transitions de $\longrightarrow^\bullet$ étiquetées par τ représentent des expressions ou des communications internes. Les transitions étiquetées par des actions visibles sont des communications externes.

$\longrightarrow^\bullet$ est définie par les règles suivantes :

$$\frac{q \xrightarrow{[b]\ x := e} q' \quad \rho(b) = \mathbf{t},\ v = \rho(e)}{(\rho, \delta, \theta) \xrightarrow{\tau} (\rho[v/x], \delta, \theta')} \qquad \text{[T1]}$$

$$\frac{q \xrightarrow{[b]\ c!s(e)} q^{'} \quad \rho(b) = \mathbf{t},\ v = \rho(e),\ c \in C^{ext}}{(\rho, \delta, \theta) \xrightarrow{c!s(v)} (\rho, \delta, \theta')} \qquad \text{[T2]}$$

$$\frac{q \xrightarrow{[b]\ c!s(e)} q^{'} \quad \rho(b) = \mathbf{t},\ v = \rho(e),\ c \in C^{int},\ \delta(c) = w}{(\rho, \delta, \theta) \xrightarrow{c!s(v)} (\rho, \delta[w.s(v)/c], \theta')} \qquad \text{[T3]}$$

$$\frac{q \xrightarrow{[b]\ c?s(x)} q^{'} \quad \rho(b) = \mathbf{t},\ c \in C^{ext},\ v\ \in (domaine(type(x)),\ s \in S}{(\rho, \delta, \theta) \xrightarrow{c?s(v)} (\rho[v/x], \delta, \theta')} \qquad \text{[T4]}$$

$$\frac{q \xrightarrow{[b]\ c?s(x)} q^{'} \quad \rho(b) = \mathbf{t},\ c \in C^{int},\ \delta(c) = s(v).w}{(\rho, \delta, \theta) \xrightarrow{c?s(v)} (\rho[v/x], \delta[w/c], \theta')} \qquad \text{[T5]}$$

– Avec $\theta^{'} = \theta\ [q^{'}/q]$: $\theta^{'}$ contient l'ensemble des états atteints à partir d'un état de θ en fonction d'une transition t : $q \xrightarrow{t} q^{'}$.

– Le triplet initial ($\rho_0, \delta_0, \theta_0$) est donné avec des valeurs de variables par défaut selon le contexte ρ_0, des files d'attente dans le contexte δ_0 (vides ou non), et les états initiaux des processus (θ_0 = q^0)

Les règles [T1] [T2] et [T4] sont interprétées comme les règles [T1] [T2] et [T3] vues précédemment (définition 1.2.2).

La règle [T3] définit, dans le contexte δ, l'envoi de la valeur v, résultat de l'évaluation de l'expression e et l'envoi est fait via un signal s. Dans le contexte ρ, l'envoi est possible si et seulement si la valeur de la garde [b] est évaluée à vrai (t = true).

La règle [T5] définit, dans le contexte δ, la réception d'une valeur v via la file d'attente interne c. La réception de cette valeur est faite en utilisant un signal s. Dans le contexte ρ, la nouvelle valeur v est affectée à la variable x. La réception est possible si et seulement si la valeur de la garde [b] est évaluée à vrai (t = true).

Exemple 4 — **[Représentation de deux automates étendus communicants]**
La figure 1.4 représente deux automates étendus communicants AEC1 et AEC2. Les automates se composent des ensembles { {types}, {états}, {actions} et de {état initial}}. Les files internes entre automates sont fAEC1 et fAEC2 (nous n'utilisons pas, pour cet exemple, de files externes dans les automates).

– AEC1 = ({ $\mathbb{Z}$, . . . }, {0,1,2,3,4}, {fAEC2 ?$s(x)$, fAEC2 !$s(x)$, [x == 1] , [x == 2], x = 2 * (x/2)}, 0)
– AEC2 = ({ $\mathbb{Z}$, . . . }, {0,1,2,3,4}, {fAEC1 ?$s(x)$, fAEC1 !s(1), fAEC1 !s(2), [x == 2], [x == 1], $x := x + 1$}, 0).

Une valeur est envoyée par AEC2 sur le signal s, à travers le canal interne fAEC1 entre les deux automates. Cette valeur est receptionnée par AEC1. Un traitement conditionnel est effectué à partir de la valeur envoyée :

1. Soit il y a arrêt de communication selon l'évaluation de la condition de x==2 à vrai.
2. Soit il y a traitement du paramètre fourni, x = 2 * (x/2), puis renvoi d'un paramètre x, fAEC2 ?$s(x)$ à AEC2.

– Puis, aprés traitement du paramètre x reçu, $x := x + 1$, nous avons :

1. Soit il y a arrêt de communication selon l'évaluation de la condition de x==2 à vrai.
2. Soit de nouveaux échanges entre AEC1 et AEC2 se déroulent, aprés l'évaluation de la condition de x==1 à vrai.

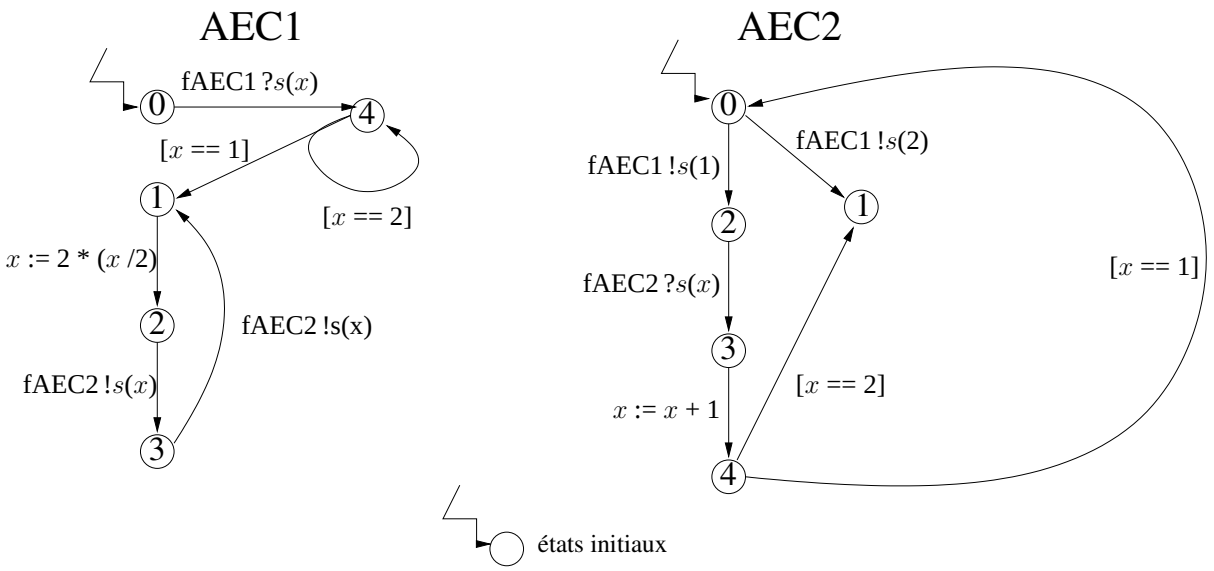

FIG. 1.4 – Représentation de deux automates étendus communicants

1.3 Conclusion

Nous venons de présenter différents formalismes LTS, IOLTS et certains automates qui permettront de décrire et de spécifier des systèmes de communication. Les expressions de composition définies sont des expressions algébriques construites à partir d'opérateurs de composition parallèle.

Le chapitre suivant présente différentes notions de test, comme le test de logiciels, le test de conformité, mais aussi introduit les méthodes de génération de test en proposant certains outils. Enfin, il permet d'introduire la méthode de test de protocole de communication qui sera une base pour les chapitres suivants.

Chapitre 2

Le test

Après une présentation générale du test de logiciels, nous définissons les notions liées au cas particulier du test de conformité et présentons brièvement quelques exemples d'outils existants. Pour introduire le chapitre 3, nous finirons ce chapitre 2 par une présentation du test de protocole de communication en détaillant ce qu'est une architecture de test, puis un testeur. Cette dernière partie permettra également de présenter les notions de contrôlabilité et d'observabilité pour un testeur.

2.1 Le test de logiciels

Parmi les étapes de conception et de validation d'un logiciel, le test est une activité incontournable. Le but du test est de valider le bon fonctionnement des programmes. Pour cela, le test a pour rôle de détecter les erreurs de conception et les erreurs de réalisation des programmes. Le test de logiciels est une activité manuelle ou automatique devenant importante dans le monde industriel et donne lieu à de nombreux efforts de recherche. Le test est largement répandu dans de nombreux domaines extrêmement variés.

De façon générale, le test de logiciels dépend du niveau d'application que nous lui donnons. Les techniques de tests utilisées diffèrent en fonction de plusieurs caractéristiques et l'activité de test comporte généralement trois phases :

1. La première est de générer des jeux de test à partir d'éléments de référence (spécification formelle, code des logiciels . . .).
2. La deuxième est d'exécuter les jeux de test produits. Cette exécution est une interaction entre un testeur et le logiciel testé.
3. La troisième est l'interprétation des résultats. Elle permet la mise en place des verdicts de test et a pour rôle de détecter les éventuelles erreurs du logiciel testé.

La génération et l'exécution des tests dépendent du niveau d'accessibilité et d'observabilité des éléments donnés. Cette observabilité peut être classifiée selon trois types :

1. ”Boîte blanche” : les tests peuvent s’effectuer sur les données internes et/ou externes du programme. En d’autres termes, nous disposons du code source du programme.
2. ”Boîte grise” : le code source du programme est inconnu mais certaines informations internes sont disponibles (ainsi que toutes les actions externes).
3. ”Boîte noire” : le test est réduit à l’interface du système à tester. Dans ce dernier cas, le code et l’architecture interne de l’implantation à tester ne sont pas connus. Seul le comportement du programme lié aux interactions avec son environnement est perçu.

Dans la pratique, il est possible d’utiliser différents types d’observabilité pour générer les tests. Si nous disposons du code source du programme, la génération des tests peut être de type boîte blanche, grise, ou noire. Le type est choisi en fonction de l’observabilité souhaitée. En effet, si seule l’interface nous intéresse (ou est disponible) l’exécution sera uniquement de type boîte noire.

Les méthodes de test de logiciels varient également en fonction de la nature du test. Elles sont différentes, si nous parlons de test fonctionnel, de test de performance, de test de fiabilité, etc. . . . Enfin, le développeur, l’utilisateur, ou la personne extérieure donnant un avis indépendant sur le système (programme) testé, n’emploient pas les mêmes techniques pour construire et exécuter des tests.

2.2 Le test de conformité

Nous nous intéressons dans cette première partie uniquement au test de conformité. Le test de conformité a été particulièrement étudié, et de nombreux travaux ont été menés dans différents contextes. Il a été formalisé dans divers domaines d’application, et a même été standardisé par exemple par [Org93, WG796] dans le domaine des protocoles de communication.

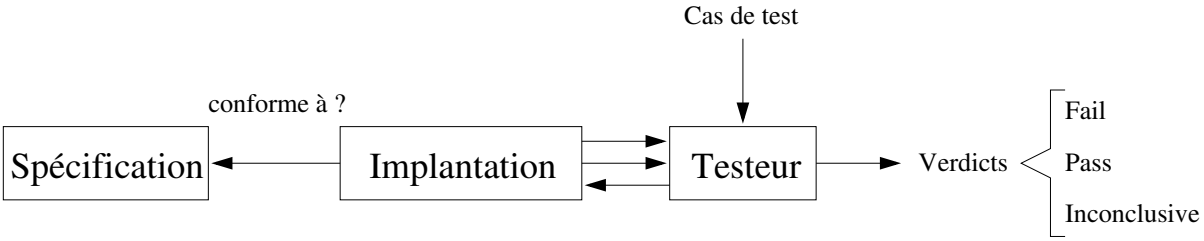

FIG. 2.1 – Principe général pour le test de conformité

Nous définissons le test de conformité (figure 2.1) comme une technique de validation par expérimentation qui a pour objectif de vérifier la correction des comportements d’une implantation

vis-à-vis d'une description des comportements (prévus lors de la spécification). La notion de conformité est formalisée à travers une relation définie entre un modèle d'implantation et un modèle de spécification (relation *conforme à ?* sur la figure 2.1). Le cas échéant, les modèles doivent prendre en compte le non-déterminisme, l'observation partielle, la représentation des temps d'exécution. Les cas de test produits, issus de la spécification et qui représentent des suites d'interactions, sont ensuite exécutés sur l'implantation à l'aide d'un testeur pour aboutir à un verdict. Les interactions avec l'implantation dépendent généralement du pouvoir de contrôle du testeur. Les cas de test exécutés par le testeur permettent de donner accès, par exemple, aux variables, aux méthodes et aux interfaces. Une spécification permet d'inclure les différents verdicts d'exécution dans les cas de test.

Les méthodes de génération de cas de test basées sur les systèmes de transitions ont pour origine les travaux sur les équivalences comportementales, et les pré-ordres. Ces méthodes cherchent à définir des relations entre les systèmes de transitions (suivant les observations).

La méthode de pré-ordre a d'abord permis de formaliser le problème du test. Les travaux sur le pré-ordre de test ($\leq_{tr}$) de [Bri88, Phi87, Abr87, NH84], ont ensuite évolué dans le but de proposer des algorithmes et des outils d'exécution de cas de test. Une implantation à tester I est plus *petite* qu'une spécification S (notée $I \leq_{tr} S$), si toutes les traces d'exécution de l'implantation sont incluses dans les traces possibles de la spécification. Les travaux proposés [Bri88, Phi87, Abr87, NH84] sont restés assez théoriques et privilégient le raisonnement sur les tests à leur construction. C'est une conséquence de la difficulté de construire directement des cas de test à partir des systèmes de transitions avec interactions symétriques (les actions d'entrée et de sortie de l'implantation ne sont pas différenciées). Dans la pratique, la différence entre les actions est fondamentale, puisque le testeur stimule l'implantation avec des actions d'entrée et observe (de façon passive) les sorties produites par l'implantation.

Les études théoriques ont permis de proposer des relations de conformité pour les modèles distinguant les entrées et les sorties. Parmi les nombreuses relations de conformité existantes (voir par exemple celles de [Pet01, LY94, CFP93]) la seule que nous allons utiliser est celle définie par [Pha94, Tre92].

> **Remarque :** *Une relation plus forte que le pré-ordre de test est le pré-ordre ($\leq_{te}$) de refus [Lyn88, NT81]. Il permet au testeur de représenter l'absence d'actions. Une nouvelle action est alors donnée pour représenter l'absence d'action et permettre l'adaptation des compositions parallèles.*

NOTE :
Les méthodes d'équivalences comportementales [SH01] ne sont pas traitées dans ce document. Nous ne présentons pas dans ce document toutes les théories du test comme par exemple celles utilisant les FSM[a] ou encore les Spécifications Algébriques . . . .

[a] Finite State Machine

2.3 Les méthodes de génération de cas de test

La construction de cas de test (et même leur exécution) a longtemps été un travail réalisé à la main. Les principaux défauts de ces méthodes manuelles sont souvent les erreurs de construction induites par les utilisateurs. Il est fastidieux de construire manuellement des tests, en particulier en grand nombre. Il faut déterminer un bon positionnement des verdicts d'exécution et définir correctement les directives de test (i.e. les objectifs à donner aux tests écrits). L'automatisation pour construire des cas de test devient alors un travail indispensable. Des études montrent que les générations automatiques de cas de test [Mor00, FJJV96] apportent un gain en productivité, en temps, en qualité, et en coût. Les automatisations ou les assistances des méthodes de génération peuvent donc être rapidement rentables économiquement. Désormais, les difficultés pour réaliser des cas de test résident dans le choix des modèles, des entrées, mais aussi des méthodes de sélection (parmi l'ensemble des cas de test possibles). Un pré-requis imposé pour automatiser une génération est de disposer d'au moins d'un élément de référence (hypothèse de départ). Soit nous disposons directement du code source de l'implantation à tester, soit nous disposons d'un modèle représentant les fonctionnalités du logiciel à tester (une spécification). À partir de ces références, différentes méthodes s'appliquent pour produire des cas de test. Pour répondre à ces questions, de nombreux travaux théoriques ont été menés pour fournir des algorithmes efficaces de génération de cas de test. Dans la suite, nous balaierons brièvement quelques méthodes de génération avant d'en présenter une de façon détaillée.

État de l'art

Nous commençons notre étude par l'énumération de quelques méthodes de génération de cas de test, généralement syntaxiques, qui utilisent le code source du logiciel à tester. Dans le cadre du test boîte blanche de code séquentiel, la sélection d'un ensemble de test est basée sur des *critères de couverture structurels* du code (voir l'étude de [ZHM97]) qui identifient des séquences d'exécution sur un modèle abstrait du code, nous parlons alors de test structurel. La couverture peut se baser sur la structure de contrôle (couverture d'états, de branches, de chemins, etc) ou sur le flot de données (définition et utilisation des variables, conditions, etc). À partir de ces ensembles, un critère de couverture produit un ensemble de chemins dans le code. Le test structurel se décompose alors en deux étapes : (1) Identification des instructions dont l'exécution est nécessaire pour satisfaire un critère de couverture. (2) Génération d'un cas de test (c'est à dire un ensemble de valeurs pour les variables d'entrée) qui garantit que l'instruction sélectionnée sera effectivement exécutée. Sur ce principe, différentes méthodes de génération de cas de test peuvent être :

– Aléatoire [Nta98] : Cette méthode est construite pour essayer des valeurs de manière aveugle tant qu'une instruction sélectionnée n'est pas atteinte. Ce test présente l'avantage de se passer de l'analyse du programme et s'adapte bien lorsque le temps d'exécution est négligeable. Il suffit de vérifier que le point sélectionné est effectivement atteint. La limite de cette méthode réside dans son incapacité à vérifier des parties très peu probables du programme.

– Symbolique [PC93, IR76] : Cette méthode remplace les paramètres d'entrée d'une procédure par des valeurs symboliques, et sélectionne l'ensemble des instructions à tester. Puis, elle détermine

dans le graphe de contrôle les chemins passant par les instructions sélectionnées. Un problème majeur réside dans la simplification et la résolution des expressions algébriques générées. Ce problème est particulièrement délicat si les chemins sélectionnés sont non exécutables. Si le programme contient des instructions itératives le nombre de chemins non exécutables peut être infini. Déterminer statiquement si un chemin est non exécutable est un problème difficile qui est indécidable dans le cas général [ASU86, Wey79].

– Dynamique [MSG00, MSG99, FK96, Kor90] : Cette méthode part d'un cas de test initial et le "corrige" (en utilisant des techniques d'optimisation) jusqu'à atteindre un point sélectionné. Le premier cas de test est généralement donné de façon aléatoire. Si le chemin associé à ce cas de test ne passe pas par le point sélectionné, ce premier chemin est dérivé pour donner d'autres chemins. Les informations pour sélectionner d'autres chemins sont issues du graphe de flôt de contrôle et du graphe du flôt de données. Cette approche explore ainsi un ensemble de chemins jusqu'à ce que le point sélectionné soit atteint. En général, les méthodes associées à cette approche n'arrivent pas à identifier des parties de code non exécutables. De plus, elles sont fortement dépendantes des techniques d'optimisation et des heuristiques utilisées pour changer de chemin.

Il existe également des méthodes sémantiques de génération qui utilisent principalement comme éléments d'entrée les modélisations des implantations à tester (par exemple : une spécification représentant les comportements d'exécution du logiciel). Pour ces méthodes, il existe différentes façons de générer des cas de test. Nous en citerons trois en exemple, les générations symboliques, par contraintes et par dérivations de spécification. Nous ne détaillons pas les deux premières mais citons les systèmes suivants :

– Le système AUTOFOCUS [PL01, PL00b, PL00a] : Il est basé sur l'exécution symbolique et la programmation par contraintes. Il requiert une modélisation abstraite du programme testé. Un composant du programme à tester est une unité indépendante qui communique avec un environnement via des ports de communication. L'ensemble des ports, qui sont eux-mêmes des composants, représente l'interface du programme. Chaque composant est décrit par l'ensemble des communications qu'il réalise. Le comportement d'un composant est modélisé avec une variante des machines de Mealy (étendue avec les entrées-sorties). Le programme est ensuite traduit en diagramme de transitions d'états (STDS[1]). Un STDS est constitué avec des variables locales, des transitions, et des états de contrôle. Un cas de test est décrit par des contraintes à vérifier, dont son exécution permet de vérifier ces contraintes sur le programme.

– Le système GATEL [MA00] : Il permet la génération des cas de test à partir de la structure de la description LUSTRE[2] [Hal98, HCRP91]. Le système repose sur la programmation logique par contraintes, et les constructions du langage LUSTRE sont traduites en contraintes sur des variables booléennes ou sur des valeurs dans les intervalles d'entiers. Le principe de ce système est de construire pour chaque variable de sortie un ensemble de cas de test (en considérant les différentes façons d'obtenir les valeurs possibles des variables). La génération consiste alors à résoudre les contraintes à l'aide de techniques de dépliage et de programmation logique par

[1] State Transition Diagrams

[2] Langage formel pour les applications à flots de données synchrones.

contraintes (symbolique).

– Le système CASTING [Aer98, ABM97] : Il propose une méthode de test qui consiste à appliquer de manière systématique une stratégie (de test) à une spécification. Ce système utilise une spécification algébrique décrite par un ensemble d'équations ou d'axiomes que doit vérifier toute mise en œuvre de l'application. La génération de cas de test est faite à partir des documents d'entrée structurés et formalisés. La génération est décomposée en plusieurs phases. (1) Extraction d'un ensemble de données de la spécification. (2) Production de scénarii à partir de ces données. (3) Génération d'un graphe d'états symboliques. (4) Construction des cas de test à partir du graphe. Il est à noter qu'un utilisateur peut fournir des spécifications de cas de test supplémentaires qui sont ensuite traitées comme dans la première phase. Le processus d'extraction consiste à appliquer une stratégie donnée à une spécification écrite dans un formalisme d'entrée pour produire un ensemble de spécifications de cas de test. La stratégie de test est donnée par l'utilisateur, c'est à dire les valeurs de paramètres des règles d'extraction. Le système apporte une solution en adoptant une approche fondée sur le traitement syntaxique des documents entrés. Le système utilise également des méthodes semi-automatiques de résolution par contraintes. Seule la phase d'extraction est dépendante du formalisme des entrées et nécessite d'être adaptée pour traiter un nouveau formalisme. Le système proposé repose sur les grammaires attribuées et les graphes. Les hypothèses d'uniformité [Gau95] à l'origine de chaque cas de test sont produites automatiquement (ces hypothèses sont la justification formelle des cas de test).

– Le système LURETTE [RWNH98] : Il apporte une réponse au problème d'automatisation de la production de séquences de test pour les systèmes réactifs. Il fixe deux points : (1) Il produit des données pertinentes, par rapport à la connaissance de l'environnement dans lequel le système évolue. (2) Il vérifie les résultats de l'exécution des tests, selon le comportement prévu du système. L'outil LURETTE [Jah04] utilise des observateurs synchrones pour exprimer la pertinence et l'exactitude des séquences de test. En particulier, l'observateur de pertinence est utilisé au hasard pour choisir des données satisfaisant des suppositions temporelles de l'environnement. Les suppositions peuvent impliquer des contraintes numériques et linéaires.

– Le système LUTESS [dBORZ99] : Il propose un environnement de test fonctionnel pour des logiciels synchrones. Il est composé d'un noyau fonctionnel permettant de construire automatiquement des générateurs de séquences de données de test pour des logiciels réactifs de type synchrone. Il permet également la génération dynamique des données de test. Les données fournies sont uniquement booléennes. Cette génération est menée sous contraintes, de manière aléatoire, et éventuellement guidée par des propriétés. L'environnement de LUTESS a été étendu par une méthode de test de type statistique, qui facilite la génération des données de test considérées comme significatives par l'utilisateur. Le système LUTESS est muni d'une interface réalisée en Tcl/Tk qui permet de paramétrer le test, de formater les résultats obtenus ainsi que de les visualiser.

NOTE :
Nous précisons que les outils LURETTE et LUTESS sont basés sur des spécifications LUSTRE [HCRP91, Hal98].

La génération de cas de test qui nous intéresse et que nous allons développer en détail est celle réalisée par dérivation de la spécification. Les méthodes sémantiques pour générer ces cas de test peuvent être divisées en deux familles. Nous établissons cette séparation car les deux ensembles diffèrent selon les modèles sous-jacents, la théorie, et donc par conséquent, les techniques de génération mises en œuvre. Les deux méthodes (ainsi que les deux types de modélisation des éléments de référence) pour produire des cas de test sont les suivantes :

1. La première, et la plus ancienne, est celle fondée sur les automates. Elle provient des méthodes de test de circuits et des problèmes de reconnaissance de machines [Moo56]. Cette méthode utilise comme modèle les machines de Mealy [Gil62]. Nous précisons que chaque transition d'une machine de Mealy est étiquetée par une entrée et une sortie.

   Nous ne développerons pas les techniques de génération de cas de test de cette méthode en détail dans la suite du document. Une synthèse en est faite dans [LY94] de D. Lee et M. Yanakakis ou dans la bibliographie commentée par A. Petrenko [Pet01]. Il existe également une variété d'algorithmes permettant de générer des cas de test à partir d'automates d'états finis dans [BP94, NT81, Koh79, Cho78].

2. La deuxième est la méthode de génération de cas de test fondée sur les systèmes de transitions. Les modèles restent peu éloignés de ceux des machines d'états finis [PYH03]. Par contre les méthodes sont différentes car elles sont directement pensées avec des modèles non-déterministes [BT01] (pour le modèle de la spécification). Nous étudierons cette méthode, qui est la base de l'outil TGV [JJ02, JM99, FJJV96] et de TORX [TB02].

2.4 État sur les outils existants pour le test de conformité

Nous présentons dans cette section, une description partielle de différents outils académiques et/ou industriels proposant des générations automatiques de cas de test de conformité.

TORX

Présentation : L'outil TORX [TB02] est le résultat d'une collaboration entre l'université de Twente (Tretmans et Brinksma), l'université de Technologie de Eindhoven et les industriels Philips Research Laboratories et KPN Research Groningen, conçu dans le cadre du projet "Cote de Résyste" (COnformance TEsting of REactive SYSTEms). Le projet a pour but de développer des méthodes, des techniques et des outils pour tester la conformité des systèmes réactifs vis-à-vis de leurs spécifications formelles. L'environnement de TORX [TB99] permet actuellement la dérivation (génération) et l'exécution automatique de cas de test pour des spécifications décrites dans un langage de description formelle comme LOTOS [BB88], PMLP (Protocal Meta Langage Promela) [BME98, Hol91], ou SDL [IT99a]. La caractéristique principale de l'outil TORX est la définition d'une architecture modulaire pour chaque composant du système testé (avec des interfaces bien définies). Cette architecture précise permet alors de remplacer un ou plusieurs composants par un

composant amélioré ou mettant en application d'autres relations de communication ou d'exécution de spécifications. L'outil TORX fonctionne en relation avec les choix des connexions existantes pour lier les composants entre eux (cette fonctionnalité est donnée par des normes industrielles). La normalisation des connexions permet à terme d'intégrer dans l'environnement de TORX des composants "tiers" qui mettent en application ces interfaces. L'outil TORX se compose de plusieurs modules : Explorateur, amorce, conducteur, adaptateur, et stockage de TTCN[3]. L'interface utilisée entre les modules de TORX est fournie par la boîte à outils OPEN/CAESAR, qui offre toutes les fonctionnalités requises.

Génération des cas de test : Un cas de test est construit pendant son exécution sur l'implantation. Le cas de test est dérivé directement de la spécification et représente un des comportements attendus pour l'implantation. Le principe de construction est : Lorsque l'implantation émet un message, l'action envoyée est comparée par le testeur avec les actions possibles de la spécification. Si l'action du message envoyé est correcte, il y a une recherche pour sélectionner une action à émettre à l'implantation. Cette recherche est faite parmi les actions successeurs de l'action précédemment donnée par l'implantation. Si il existe un ensemble d'actions successeurs, et si il existe plusieurs actions possibles dans cet ensemble, l'outil sélectionne la première ou de façon aléatoire parmi les actions possibles de cet ensemble [VT00]. L'outil construit itérativement l'ensemble des successeurs par actions de déterminisation et de tau-réduction.

Exécution : L'action émise par l'implantation est comparée avec les actions possibles de la spécification. Deux cas se présentent alors : (1) L'action envoyée par l'implantation est conforme aux attentes de la spécification, et l'outil renvoie une nouvelle action à l'implantation (provenant des successeurs de l'action reçue contenue dans la spécification). (2) L'action envoyée par l'implantation n'est pas conforme aux attentes de la spécification, l'exécution s'arrête.

La notion d'objectif de test étant inexistante, il n'y a que deux verdicts possibles : PASS et FAIL. Un verdict PASS est émis après interaction conforme et totale vis-à-vis du testeur. Un verdict FAIL est émis, si une action envoyée par l'implantation n'est pas conforme aux attentes de la spécification. L'algorithme sous-jacent du testeur est un algorithme de τ-réduction (ϵ-clôture de Aho et Ullman) et de déterminisation. L'exécution d'un cas de test ne nécessite pas beaucoup de place mémoire, seul l'état courant et les successeurs sont stockés temporairement. Il est supposé que les calculs de successeurs et des verdicts n'ajoutent pas de délai aux communications et n'interfèrent pas sur la bonne marche du testeur.

TGV

Présentation : L'outil TGV (Test Generation with Verification technology) dont les premiers développements datent de 1995 a été produit initialement pour l'outil ObjectGéode [RJA99]. L'outil TGV est issu de la contribution commune Vérimag (Grenoble) et l'ancien projet Pampa de l'IRISA (Rennes). Il utilise les bibliothèques des boîtes à outils CADP de Vérimag et de l'Inria Rhône-Alpes (Grenoble) et de la boîte à outils IF (Intermediate Format [Boz99, BGMS98]) : outils de modélisation et de vérification des systèmes distribués par Vérimag.L'outil TGV est aussi intégré à l'outil commercial TestComposer de la boîte à outils ObjectGéode (Telelogic [RJA99]) depuis 1999. Reconnu dans les communautés industrielles et académiques internationales, TGV a été utilisé dans

[3]Tree and Tabular Combined Notation : TTCN

le cadre du projet AGEDIS [4]. L'outil TGV a été conçu de façon à être relativement indépendant des langages de spécification, et incorpore également le calcul des temporisateurs et le calcul des postambules.

Génération des cas de test : La description de cette génération sera proposée en détail dans le chapitre 3 suivant. L'originalité de l'outil TGV tient dans l'efficacité de ses algorithmes qui garantissent la limitation de l'explosion des états pendant les compositions. Les méthodes de cet outil sont constamment améliorées du point de vue algorithmique mais aussi par l'interfaçage. Cet outil génère des cas de test à la volée (sans construire complètement le graphe d'états de la spécification), à la fois sur des spécifications SDL[5][IT99a], grâce à sa connexion à ObjectGéode, sur des spécifications LOTOS [BB88], grâce à sa connexion à l'environnement CADP et sur des spécifications UML [EFLR99], avec l'outil Umlaut du projet Triskell et le langage IF de Vérimag. L'outil TGV peut aussi fonctionner avec des graphes explicites décrits au format Aldébaran [Fer88], et les cas de test produits dans le format Aldébaran, peuvent être traduits en langage TTCN.

Le projet AGEDIS

Présentation : AGEDIS est un projet européen IST[6] réalisé entre 2000 et 2003. Ce projet vise à fournir des méthodes et outils pour l'automatisation du test de logiciels, en particulier pour les logiciels répartis, principalement dans le domaine des télécommunications. Ce projet résulte de la collaboration entre IBM Haifa (Israël) et IBM Hursley (G.-B.), France Télécom R D (Lannion), IntraSoft (Grèce), Imbus (Allemagne), l'université d'Oxford, l'IRISA (Rennes) et Vérimag (Grenoble). Le langage choisi pour décrire les systèmes est un profil UML [EFLR99], contenant les diagrammes de classes, d'objets et d'états. Une description UML avec ses critères de sélection est compilée dans le langage IF de Vérimag, lui-même compilé en une API de simulation qui lui offre les fonctions de parcours et d'observation d'un produit entre modèles et critères de sélection. Ce profil permet également la description de critères de sélection de cas de test, combinant objectifs de test à la TGV et critères de couverture d'expressions, en utilisant les diagrammes d'états UML.

Génération de cas de test : L'outil de génération de cas de test est une extension de l'outil TGV. Il utilise l'API du simulateur IF et incorpore des fonctionnalités de l'outil d'IBM Gotcha, en particulier des critères de couverture.

Exécution : Les cas de test produits sont traduits dans un format XML et utilisés par l'outil d'exécution de test Spider [liH].

STG

Présentation : L'outil STG (Symbolic Test Generation) [CJRZ01, CJRZ02] est un générateur de cas de test symboliques pour des systèmes réactifs. Cet outil prend en entrée une spécification et un objectif de test symboliques. Ces entrées sont décrites par des IOSTS (In/Output Symbolic Transition System extension du langage IF).

Génération de cas de test : La spécification indique les comportements attendus. L'objectif de

[4] Automated Generation and Execution of Test Suites for Distributed Component-based Software : AGEDIS
[5] Specification and Description Language
[6] http ://www.agedis.de/

test permet de sélectionner un sous-graphe intéressant parmi les comportements disponibles dans la spécification. Le produit entre l'objectif de test et la spécification est analysé et simplifié. L'analyse permet de donner un cas de test capable d'émettre un verdict au plus tôt et de guider dynamiquement l'exécution en fonction du comportement de l'implantation, afin d'éviter au maximum les verdicts inconclusifs.

Exécution : Les paramètres de l'implantation et du cas de test doivent être fixés à l'exécution pour les rendre exécutables. Lors de l'exécution, la résolution de contraintes utilisée est liée aux choix des émissions du testeur exprimées en arithmétique de Presburger par le solveur du projet Omega[7].

GOTCHA

Présentation : Dans son principe l'outil GOTCHA (IBM [FHP02]) permet au testeur d'observer des valeurs de variables.

Génération de cas de test : GOTCHA adapte la prise en compte de critères de sélection de cas de test (plus puissant que les objectifs de test actuels). Parmi les possibilités, ces critères de sélection peuvent calculer tous les cas de test couvrant toutes les valeurs possibles d'une expression quelconque sur les variables. Le problème est alors de créer dynamiquement, pendant la création des cas de test, des objectifs de test où les états sont accepteurs, si ils correspondent à l'atteignabilié de nouvelles valeurs d'expressions. Pour cet outil, les algorithmes utilisés permettent de produire éventuellement plusieurs cas de test pour un même objectif en incorporant un choix aléatoire dans le parcours du graphe de test.

Exécution : Le testeur utilise les cas de test pour stimuler l'implantation. Les verdicts (valide ou non valide) sont le résultat des observations des interactions.

MCITT

Présentation : L'outil MCITT (Manufacturer's CORBA Interface Testing Toolkit) est un outil basé sur le développement CORBA (Common Object Request Broker Architecture). CORBA [Bak97, COR] définit une plate-forme de développement permettant aux concepteurs d'abstraire des problèmes liés à la répartition (concurrence, communication entre objets distants de l'application, etc. . .) des données dans les systèmes testés. Il y a deux manières de définir les comportements dans l'outil MCITT. (1) Une manière procédurale, par définition d'un langage pour tester les composants (ITL : Interface Testing Language). (2) Une manière déclarative, en utilisant les spécifications des composants (CISs : Component Interaction Specifications). Actuellement, les spécifications ont quelques limitations, mais elles sont la plupart du temps mises en application. L'architecture de CORBA permet de représenter des systèmes répartis contenant un grand nombre de composants. Les interactions entre composants sont représentées conformément aux modèles des protocoles client-serveur. Les langages utilisés pour représenter les programmes clients-serveurs sont les langages Java ou C^{++}. Les comportements modélisés sont donnés sous forme de diagrammes UML. Enfin, l'utilisation de CORBA permet de définir de nombreux systèmes réels, et les architectures distribuées.

Génération de cas de test : Nous notons tout d'abord que la contruction des cas de test n'est pas

[7] (http ://www.cs.umd.edu/projects/omega)

réalisée par l'outil MCITT lui-même. Les spécifications représentent les interactions entre les composants. Les cas de test construits, à partir des spécifications, représentent chacun un cheminement d'actions (contenues dans la spécification) suivant un scénario d'interaction client-serveur précis. Un cas de test se compose de l'arbre des demandes d'un client à un serveur (sous une description CORBA) en indiquant explicitement les entrées et les sorties.

Exécution : Le testeur est un client qui fournit les actions du cas de test, et les réponses aux stimulis sont gardées sous forme d'arborescence puis comparées aux attentes. Les arborescences sont représentées sous forme de diagrammes de Collaboration UML. Trois verdicts sont possibles pour chaque exécution. (1) Le verdict Pass indique que le programme testé passe tous les cas de test donnés. (2) Le verdict Fail montre que certains résultats aux cas de test ne passent pas (erreur dans le programme). (3) Le verdict Maybe indique que si le cas de test est conçu pour délivrer des exceptions (Java), alors ce cas de test ne peut pas s'appliquer dans un programme, si côté client, une exception est délivrée. Dans cette situation, avec le verdict Maybe, le testeur observe qu'il n'est pas assuré que le serveur soit conçu pour lever les exceptions.

TINA

Présentation : TINA [SL, SLH98] est un outil combinant les méthodologies de RM-ODP [IT95] (Open Distributed Processing - Reference Model) et celles de CORBA, appliquées aux domaines des télécommunications. L'architecture TINA est une architecture ouverte visant à faciliter le développement de services. Les spécifications et les implantations sont définies par des langages ODL (Object Definition Language [IT99b]). Les langages ODL proviennent d'un sur-ensemble des langages définis par les OMG (Object Management Group), et spécifient les définitions des communications entre composants en détaillant : (1) Les types de données, les communications, les opérations, les interfaces et l'architecture de systèmes. (2) La relation de conformité utilisée est décrite dans [ISO97]. (3) La vérification des systèmes se fait à travers des points de contrôle, explicitement décrits. (4) L'architecture de test est basée sur les communications asynchrones des composants des systèmes.

Génération de cas de test : Les cas de test sont représentés sous la forme de TTCN (Tree and Tabular Combined Notation), et sont le résultat d'une extraction d'une spécification en fonction d'un objectif de test. Ils sont une représentation des dépendances existantes entre les opérations des composants. Les algorithmes sont déduits des méthodes utilisant les systèmes de transitions ou/et étendus aux machines d'états finis.

Exécution : Le testeur utilise les cas de test pour stimuler l'implantation. Les verdicts (valide ou non valide) sont le résultat des interactions.

> **Remarque :** *Les environnements de CORBA et de TINA ont bénéficié des recherches autour du test d'application orienté objet. Un protocole peut être considéré comme une application distribuée, et donc utiliser les techniques de test de protocoles comme base de travail. Maintenant la plupart des plate-formes de développement sont orientées objets. En ce qui concerne la phase de test d'application orientée objet, les recherches sont encore jeunes et surtout focalisées sur le test de fonctionnement plutôt que sur le test de conformité.*

SAMSTAG

Présentation : L'outil SAMSTAG {*Sdl And Msc baSed Test cAse Generation*} et la méthode SAMSTAG [Gra94, Nah94] sont le résultat d'un projet de l'Université de Berne entre 1991 et 1993 (soutenu par les PTT Suisse). L'objectif du projet était de développer une méthode et une implémentation pour la génération automatique de cas de test abstraits (au format TTCN[8]) basées sur des spécifications SDL [IT99a] et des objectifs de test en diagramme MSC [IT92]. La méthode de l'outil SAMSTAG est en accord avec la génération de suite de test[9] de conformité donnée par la norme ISO 9646 [ISO94]. Les objectifs de test sont une des bases pour la génération et la sélection des cas de test. Ils sont nécessaires pour analyser les résultats du test pour les fonctions du protocole qui ont été testées. Pendant les années 1993 à 1995, la méthode de SAMSTAG est améliorée pour prendre en compte le problème de l'explosion des états [GHTS96]. Le résultat est le développement et l'implémentation d'une méthode de simulation d'ordre partiel pour des spécifications SDL [GHT95].

Génération de cas de test : Pour une spécification SDL et un objectif de test MSC donnés, l'outil SAMSTAG génére des cas de test sous la forme d'un TTCN. La spécification est un système SDL fermé [GHNS95]. C'est à dire qu'elle ne comprend pas seulement la description du protcole à tester (IUT[10]), mais aussi tous les autres protocoles (et les paramètres) dépendant du protocole testé. L'ensemble du système comprenant, l'IUT à tester et tous les autres paramétres est appelé un SUT[11]. Pour générer les cas de test, l'outil simule d'abord la spécification SDL et cherche dans cette spécification les traces incluant celles de l'objectif de test MSC (ayant une exécution comportant un début et une fin) à partir de l'état initial de la spécification. Cette première recherche permet de trouver des traces (finies) améliorées (avec celles de la spécification) de l'objectif de test. Parmi toutes les traces obtenues, une sélection des traces pertinentes est effectuée. Les traces candidates (PPO = Possible Pass Observable) sont les séquences observables qui conduisent à un état final (*pass*). Il s'agit des traces qui ne contiennent que des actions observables (définies en fonction du système testé), et dont toutes les actions internes des SUT sont enlevées. Une fois qu'une PPO est trouvée (candidate), une nouvelle vérification est effectuée pour garantir que la PPO délivra bien un verdict *pass*. Une séquence vérifiée est alors appelée UPO (Unique pass observable). Il est précisé que les systèmes testés peuvent avoir des comportements non détermistes. En accord avec la norme ISO 9646, il noté qu'un verdict *inconclusive* peut se produire. Pour prendre en compte cette éventualité, l'outil SAMSTAG génére des *observations inconclusives* pour les UPO. Ce qui signifie la prise en compte des actions possibles dans la spécification SDL, mais non présentes dans l'UPO. Finallement, le cas de test TTCN généré, à partir de l'UPO, contiendra une description des comportements dynamiques souhaités combinée avec toutes les actions observables. Puis au TTCN, l'outil génére les définitions des types et les déclarations de contraintes pour tous les messages pouvant être envoyés et reçus par le système testé. Les définitions des types sont basées sur les signaux de la spécification SDL. Les valeurs des contraintes sont fournies pendant l'exécution de l'UPO et des *observations inconclusives* de l'UPO. Dans tous les autres cas que ceux donnés par le TTCN produit, un verdict *fail* sera donné.

[8] Tree and Tabular Combined Notation : TTCN
[9] ensemble de cas de test
[10] Implemention Under Test
[11] System Under Test

Bien que notre présentation soit concentrée sur le test de logiciels, il existe également des outils de génération et d'exécution de cas de test de conformité pour les circuits (hardware). La génération de cas de test de l'outil **DILL** (ci-dessous présenté) est similaire à la méthode TGV.

DILL

Présentation : Pour l'outil DILL [HT99], les modélisations des circuits sont faites avec des IOLTS. La relation de conformité utilisée est **ioco** [Tre96]. Le principe de l'outil est de tester si le circuit réalisé (IUT) est conforme à la modélisation du circuit (spécification).

Génération de cas de test : La génération de cas de test peut être assimilée à celle de l'outil TGV (présentée dans le chapitre 3 suivant). Où, à partir d'une spécification (description du circuit), les cas de test contruits proposent une séquence d'exécution permettant de réaliser (et donc de vérifier) une fonctionnalité précise du circuit. Un objectif de test permet dans la phase de génération de sélectionner les parties du circuit à tester.

Exécution : Un testeur utilise les cas de test générés pour fournir les données attendues au circuit à tester. Des verdicts (valide ou non valide) sont édités en fonction des observations obtenues après l'exécution du cas de test sur le circuit.

2.5 Le test de protocole de communication

Nous savons maintenant que pour produire des cas de test, nous utiliserons un modèle de spécification (représentant les comportements souhaités).

Les comportements à tester (issues de la spécification) sont représentés par des modèles. Pour réduire les modèles à des comportements précis, à tester, certaines directives de test appelées "objectifs de test" sont founies. Un objectif de test, défini par la norme ISO 9646 [ISO92], décrit de manière informelle quel est l'objectif particulier du test à effectuer sur l'implantation. Il a pour rôle, pendant la génération, de limiter la production de cas de test à certains comportements. L'opération de génération, effectuée entre la spécification et l'objectif de test, se déroule en deux phases. (1) La première produit toutes les séquences d'exécution en isolant (par sélection) tous les comportements de la spécification induits par les actions de l'objectif. Elle est réalisée grâce à un produit synchrone entre la spécification et l'objectif. Ce produit synchrone représente l'ensemble de tous les cas de test possibles liés à l'objectif. Maintenant, et comme les modèles de la spécification et de l'objectif sont les systèmes de transitions à entrées-sorties (IOLTS), le modèle des ensembles des cas de test produit est représenté sous la forme d'un graphe de test (IOLTS). (2) Un cas de test correspond à l'extraction d'un des sous-graphes issus du graphe, sa génération est complétée par des verdicts. Ces verdicts permettent de vérifier, lors de l'exécution d'un cas de test, que l'implantation est conforme ou non à la spécification.

Dans le cas particulier du test de conformité des protocoles la norme ISO 9646 [ISO92] propose trois sortes de verdict : $Pass$ (réussi), $Fail$ (échec) et $Inconclusive$ (inconcluant).

Des algorithmes efficaces existent maintenant [JM99] et ont permis de traiter des études de cas de taille industrielle [FJJV97, MB00].

Le test de conformité présenté est un test de type fonctionnel, appliqué sur des protocoles de com-

munications (la validation étant faite entre les fonctions réalisées par une implantation et les fonctions réalisées par la spécification).

La construction et l'exécution des cas de test sont de type boîte noire (par exemple, pour des raisons de confidentialité, le code de l'implantation n'est pas fourni). En effet, dans le domaine des télécommunications, le test de conformité est réalisé par des laboratoires indépendants. Ces derniers fournissent alors un certificat de conformité validant des systèmes déjà développés et utile pour des clients éventuels.

Une architecture de test :

Plusieurs architectures de test, pour générer et exécuter des cas de test, ont été standardisées dans [Org93, WG796]. Les travaux de [WG796] tentent de rapprocher la théorie du test de la norme en proposant un cadre formel sans pour autant définir précisément un algorithme de génération. Nous nous intéressons ici à une architecture de test local (figure 2.2) conçue pour tester les implantations à l'aide d'une interface. Cette interface décrit les *Points de Contrôle et d'observation PCo* de l'implantation. L'architecture de test est prise en compte lors de la génération de cas de test à partir de la spécification.

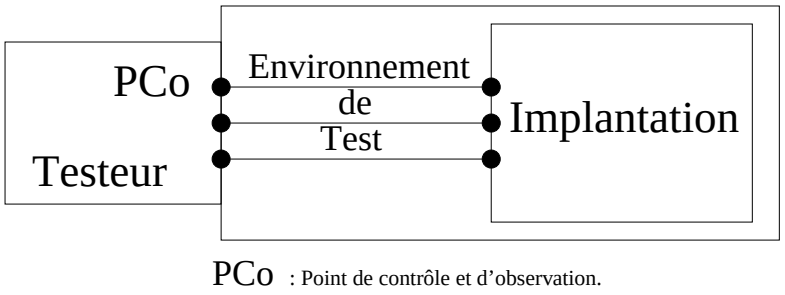

FIG. 2.2 – Présentation d'une architecture de test local

> **Remarque :** *[DKPR95] introduit les notions de points d'observation et de contrôle pour mesurer le comportement d'une implantation (dans le cadre des tests de protocole).*

Testeur :

Un testeur est (figure 2.3) vu comme un composant interagissant (en utilisant les PCo) avec une implantation à tester. Pour stimuler l'implantation, le testeur dispose d'un cas de test donné. Le verdict, délivré par le testeur, est le résultat de son interaction avec l'implantation. Le cas de test fourni est un graphe représentant des traces d'exécution souhaitées, constitué d'actions à émettre et d'actions

de réponse aux émissions. Le rôle du testeur est d'émettre les actions puis d'attendre la réponse de l'implantation. Le testeur arrête l'exécution et délivre un verdict en fonction de l'observation de la trace d'exécution de l'implantation testée. Un testeur est donc, un programme comparant les réponses souhaitées avec les réponses données par l'implantation en fonction des données émises.

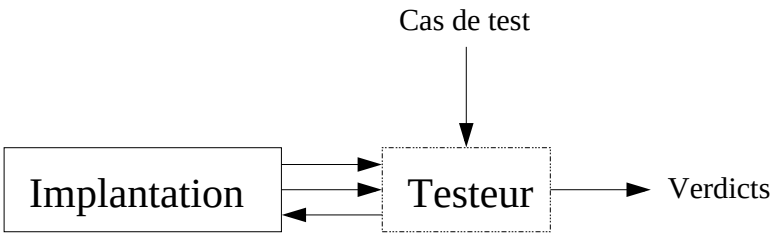

FIG. 2.3 – Présentation du testeur

Pour améliorer la contrôlabilité des tests (via un testeur), il est possible d'agir sur la sélection des actions à placer dans un cas de test. Comme un testeur stimule l'implantation en lui donnant des entrées, le fait qu'un cas de test comporte un grand nombre d'actions d'entrée (quand cela est possible) permet de guider le test sur des actions à tester. Parmi tous les cas de test, les premiers à être exécutés sont ceux comportant plus d'actions d'entrée que d'actions de sortie. Dans ce cas le testeur semble contrôler le test. Dans le cas inverse le rôle du testeur est proche de celui d'un observateur.

Cela nous permet de définir les notions de *contrôlabilité* et d'*observabilité* du testeur :

Définition 2.5.1 (Contrôlabilité et Observabilité) *Les entrées (resp. sorties) de l'implantation correspondent aux sorties (resp. entrées) du testeur. Donc le testeur contrôle les entrées et observe les sorties de l'implantation.*

D'après les modèles choisis, les IOLTS, la notion de contrôlabilité et d'observabilité devient évidente car elle se porte sur les actions d'entrée et de sortie. Nous verrons par la suite comment sélectionner et créer des cas de test en fonction de ces critères d'observabilité et de contrôlabilité.

> **Remarque :** *Le testeur n'interagit qu'en fonction du cas de test donné. Dans notre étude, les cas de test sont des IOLTS déterministes. Lors d'une exécution d'un cas de test, il est impossible pour le testeur de revenir en arrière et donc d'observer un branchement non-déterministe (possédant deux sorties identiques sur un même état décrit dans le modèle d'implantation). Une fois qu'un choix de sortie est établi le testeur continue l'exécution.*

L'exécution de cas de test

Une exécution d'un cas de test se fait à l'aide d'un testeur, et se termine dès l'obtention d'un verdict. Au terme d'une exécution, une implantation est jugée non conforme à la spécification de référence, si le verdict est $Fail$. Une implantation est jugée "conforme" si l'ensemble des cas de test exécuté par le testeur aboutit au verdict $Pass$. Finalement, une exécution peut être rejouée si nous obtenons le verdict $Inconclusive$. Ce dernier verdict indique que l'implantation est passée par un chemin qui n'était pas décrit par le cas de test et donc l'exécution de ce chemin n'est pas réalisable par le testeur. Le fait qu'une implantation exécute un chemin non demandé peut s'expliquer par le non déterminisme de l'implantation. Pour essayer d'avoir un autre verdict (Pass ou Fail, et donc un chemin exécutable par le testeur), il est possible de recommencer une exécution avec le même cas de test.

Conclure à la conformité d'une implantation impose que tous les tests effectués aient obtenu un verdict Pass. En revanche, conclure à la non conformité ne demande qu'un verdict Fail. Du fait de la complexité croissante des systèmes, il n'est pas toujours possible de vérifier une implantation dans sa globalité (problème de réalisation). Il est donc nécessaire de choisir un ensemble de tests parmi tous les tests possibles. Les exécutions ne se limitent qu'aux observations des cas de test donnés. Un système n'est donné conforme que pour l'ensemble des cas de test appliqué. Plus le nombre de cas de test est grand, plus le verdict final est sûr.

Chapitre 3

Génération et exécution de cas de test de conformité

Ce chapitre introduit les différents éléments nécessaires à la génération et à l'exécution de cas de test. Plus précisément, nous présentons en détail une technique de génération de cas de test pour le test de conformité servant de base à notre travail. Cette méthode est présentée car bien adaptée au test des systèmes distribués (asynchrone), en particulier aux systèmes tels que les protocoles de télécommunications. Le test de conformité a pour but de détecter les différences observables entre une spécification de référence et une implantation donnée. La méthode de génération étudiée est décrite dans [JM99, FJJV96]. Elle est totalement automatisable [BFdV$^+$99, Jér02] et a permis de produire des outils académiques [JJ02, TB02] et commerciaux [TestComposer de la boîte à outils ObjectGéode (Telelogic) [RJA99]]. Cette méthode consiste à générer des cas de test à partir d'une spécification formelle et d'un objectif de test comportant les actions précises à tester. La spécification, l'objectif de test et le comportement des systèmes sont modélisés par des systèmes de transitions étiquetées à entrées-sorties (IOLTS).

3.1 Modèles de spécification et d'implantation

Pour décrire en détail la méthode de génération de cas de test de conformité, selon l'outil TGV [JJ02], nous commençons par présenter les différentes modélisations des éléments d'entrée. Puis, nous présentons la relation de conformité utilisée.

3.1.1 Modèle de spécification

Une spécification est une hypothèse de construction pour générer automatiquement les cas de test. Elle représente tous les comportements attendus d'un système à tester. Il faut donc que sa représentation soit suffisamment exhaustive, précise et non ambiguë. Comme elle peut décrire des systèmes communicants, il est nécessaire et utile de la formaliser avec le modèle IOLTS (permettant

ainsi de différencier les actions d'entrée, de sortie et internes).

Définition 3.1.1 (Modèle d'une spécification pour le test de conformité) *Nous considérons que les comportements de la spécification sont modélisés par un IOLTS ($Q, A, \longrightarrow, q^{init}$). Nous supposerons que la spécification ne possède pas de séquences infinies d'actions internes passant par une infinité d'états distincts.*

3.1.2 Modèle d'implantation

Une implantation est généralement un programme réactif ayant pour rôle d'interagir avec un environnement. Dans le cadre de l'exécution de nos tests, l'implantation réelle (logicielle ou matérielle) sous tests (IUT[1]) est une boîte noire. Seul le comportement des interactions est donc visible. Nous ne considérons pas une implantation comme un objet formel, mathématique. Cependant, pour effectuer des raisonnements formels et afin d'établir la conformité de l'implantation vis-à-vis de la spécification, il faut se baser sur des modèles formels. Nous supposons que les comportements possibles de l'implantation sont modélisables par un IOLTS.

Définition 3.1.2 (Formalisation du modèle d'une implantation : IOLTS) *Une implantation est modélisée par un IOLTS = (Q, A, $\longrightarrow$, q^{init}) où les entrées (resp. les sorties) sont les sorties de l'environnement (resp. les entrées) : $A = A^O \cup A^I \cup \{\tau\}$.*

> **Remarque : (1)** *Pendant la phase de génération de cas de test nous supposons que l'alphabet des actions de l'implantation (IUT) est compatible avec celui de la spécification (S) c'est à dire $A^I_S \subseteq A^I_{IUT}$ et $A^O_S \subseteq A^O_{IUT}$. L'hypothèse de compatibilité des alphabets nous permet d'assurer que l'implantation testée acceptera toutes les actions fournies par le testeur.*

> **Remarque : (2)** *La méthode de test de conformité exige dans le modèle de l'implantation la complétude en entrée pour garantir qu'un blocage pendant l'exécution d'un cas de test conduit forcément à un verdict. L'hypothèse de complétude en entrée est raisonnable dans le cas où l'implantation ne refuse aucune entrée invalide ou inopportune.*

3.2 La relation de conformité ioco

Une relation de conformité est une relation entre implantations et spécifications qui définit exactement quelles sont les implantations conformes à une spécification de référence. Parmi toutes les relations de conformité existantes sur les IOLTS, celle que nous utilisons est la relation **ioco** [Tre96] issue des travaux de J. Tretmans et M. Phalippou [Tre92, Pha94].

[1] Implementation Under Test

NOTE :
La définition 3.2.1 suivante fait appel à différentes définitions 3.4.2 et 3.4.3 qui seront présentées plus tard dans le document. Les définitions 3.4.2 et 3.4.3 dépendent des transformations des IOLTS pendant un test de conformité nécessaire pour appliquer la relation **ioco**.

Définition 3.2.1 (ioco : La relation de conformité) *La relation de conformité est établie entre une implantation IUT et sa spécification* S *:* $IUT \xrightarrow{\textbf{ioco}\ ?} S$. *Une implantation est conforme à une spécification pour la relation* **ioco** *si, après toute trace suspendue* α *(définition 3.4.3) de la spécification (c'est à dire une trace de son automate de suspension* $\Delta(S)$ *définition 3.4.2), les sorties de l'implantation (y compris les blocages (section 3.4.2) puisque l'action* $!\delta$ *représentant un blocage dans la définition est une sortie) sont incluses dans les sorties existantes de la spécification.*

$$IUT \textbf{ ioco } S \text{ ssi } \forall \alpha \in STrace(S),\ \big(\mathcal{O}(\Delta(IUT)\ \ after\ \sigma)\ \big) \subseteq \mathcal{O}(\Delta(S)\ after\ \alpha)$$

L'implantation acceptant toutes les entrées, les blocages ne peuvent se produire que si il est possible de ne rien pouvoir émettre.

La relation de conformité **ioco** permet d'établir : "après toutes traces suspendues α de S, les sorties de l'IUT (y compris δ) sont incluses dans celles de S [Jér04] :

$$IUT \textbf{ ioco } S \Longleftrightarrow \big(\ STrace(IUT) \bigcap STrace(S).A^{o}_{\Delta(IUT)}\ \big) \subseteq STrace(S)$$

Dans la définition 3.2.1, nous supposons que seules les implantations sont complètes en entrée (définition 1.1.14), mais, nous ne supposons rien de particulier sur la spécification. Or, si la spécification est complète en entrée sur l'alphabet des implantations alors la formulation de la relation de conformité **ioco** est simplifiée :

Propriété 3.2.1 (ioco : Relation de conformité simplifiée) *Soit IUT et S deux IOLTS compatibles (de même alphabet d'entrée) et complets en entrée sur cet alphabet :*

$$IUT \textbf{ ioco } S \Longleftrightarrow STrace(IUT) \subseteq STrace(S)$$

NOTE :
Les preuves des propriétés précédentes sont disponibles dans [Jér04].

3.3 Exemple d'implantations conformes, ou non conformes.

Quelles sont les implantations conformes, ou non conformes à une spécification donnée selon la relation ioco ?

– La figure 3.1 représente une spécification de référence.

– Les figures 3.2 sont des modèles d'implantation conformes à la spécification de référence.

1. La première implantation comporte la description d'un sous-ensemble des comportements de la spécification. Selon IUT **ioco** S, après les traces de $\Delta(S)$, les actions de sortie de l'IUT sont incluses dans les actions de sortie de S. Par exemple après $(\delta^*.?a!b!c)^*.\delta^*.?a$, et $\{!b\} \in \{!b,!c\}$. La relation **ioco** permet de restreindre les sorties de l'IUT, et donc de faire des choix d'implantation.
2. La deuxième implantation est plus complète. Elle permet une nouvelle fonctionnalité d'entrée après réception d'une action d'entrée $?d$. Notons pour ce deuxième exemple que selon la relation **ioco** les actions d'entrée supplémentaires sont possibles, car la conformité porte uniquement sur les actions de sortie après les traces de $\Delta(S)$.

– Les figures 3.3 représentent des modèles d'implantations non conformes à la spécification de référence.

1. La première implantation a une action de sortie interdite $!d$, ce qui met en défaut l'inclusion des actions de sortie selon **ioco**.
2. La deuxième implantation a une action de blocage non prévue par la spécification par une succession d'actions internes. Or, selon la spécification, l'implantation ne peut pas se bloquer et doit émettre obligatoirement une action de sortie $!c$.

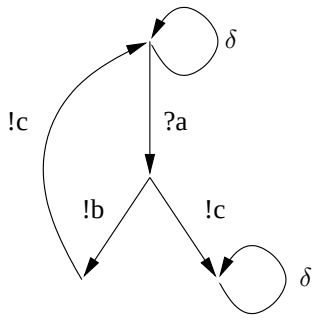

FIG. 3.1 – Spécification de référence

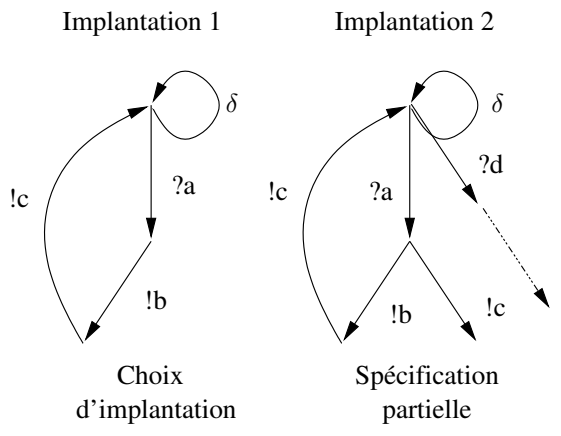

FIG. 3.2 – Implantations conformes (**ioco**)

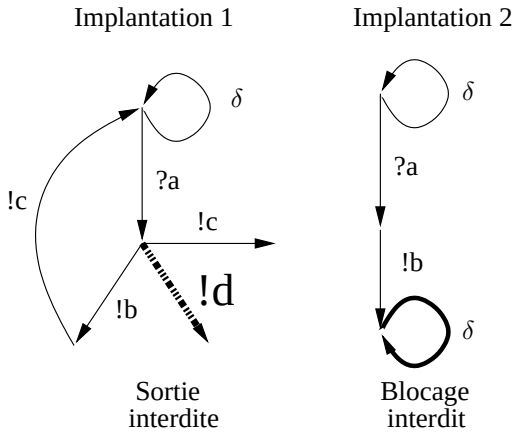

FIG. 3.3 – Implantations non conformes (**ioco**)

NOTE :
Généralement pour un outil, une seule relation de conformité est utilisée pour générer et exécuter des cas de test. C'est la raison pour laquelle nous ne présentons ici que la relation **ioco** comme propriété de conformité, utilisé dans l'outil TGV.

3.4 Génération de cas de test de conformité basée sur les systèmes de transitions à entrées-sorties

Dans le cadre du test de conformité, le modèle d'implantation n'est pas forcément connu, ce qui empêche de vérifier directement qu'une implantation est conforme à sa spécification par des techniques de type model-checking. L'implantation est vue comme un système réactif avec lequel un autre système peut interagir pour exécuter des tests.

Cependant, la génération reste un problème difficile d'un point de vue théorique et même pratique. Bien que la génération automatique de test connaisse depuis plusieurs années un réel succès industriel dans le domaine du matériel, elle a eu plus de mal à percer dans le domaine du logiciel. Une des raisons, est peut être la trop grande distance entre les techniques universitaires et la pratique industrielle [LL95].

Différentes applications pour générer des cas de test de conformité

La méthode présentée est tirée de l'outil TGV [JM99, FJJV96], outil développé conjointement par les laboratoires l'IRISA (Rennes) et Vérimag (Grenoble). Cette méthode est intégrée dans l'outil commercial TestComposer d'*Object*GEODE [RJA99]. Puis, la méthode a évolué grâce au projet européen AGEDIS, (IBM Haifa (Israël) et IBM Hursley (G.-B.), France Télécom R D (Lannion), IntraSoft (Grèce), Imbus (Allemagne), l'université d'Oxford), Vérimag (Grenoble), aboutissant à

une chaîne complète pour générer et exécuter des cas de test de conformité.

L'outil TGV génère automatiquement des cas de test à partir d'une spécification et d'un objectif de test donnés sous forme de IOLTS. Un cas de test produit par TGV est aussi un IOLTS. Les cas de test sont pourvus d'horloges (timers) et de verdicts (Pass, Fail et Inconclusive). Les algorithmes d'exécution détectent les blocages au niveau des communications grâce aux horloges incluses (armées au moment d'exécuter un cas de test) qui garantissent une fin d'exécution.

Une approche pour la génération de cas de test de conformité

La figure 3.4 présente une architecture pour le test de conformité (avec les modèles décrits précédemment).

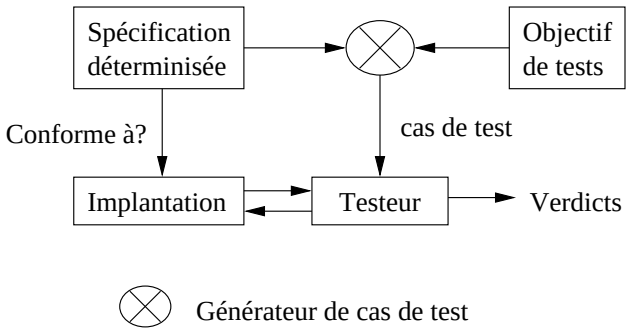

FIG. 3.4 – Modèle de génération et d'exécution de cas de test de conformité

Nous définissons maintenant comment obtenir les cas de test suivant les méthodes intégrées dans l'outil TGV. Un cas de test est un test élémentaire, et pour un système réactif, il décrit une suite d'interactions entre le testeur et l'implantation à tester. Les cas de test doivent permettre d'observer les traces suspendues de l'implantation (selon la relation ioco). Un cas de test est un IOLTS étendu avec un verdict et son produit à partir de la spécification et d'un objectif de test. La spécification est déterminisée et contient différents blocages.

3.4.1 Déterminisation d'une spécification

Il est à noter que généralement l'IOLTS associé à une spécification peut être non déterministe. Dans notre étude, les tests sont des observations de réaction d'un système face à des stimulii. En fait, un testeur interagit avec le système à l'aide de séquences d'exécutions (issues de la spécification), puis, compare les actions de réponse du système aux comportements attendus. Il est donc nécessaire d'identifier l'ensemble des séquences de la spécification, où toutes les actions sont prévisibles par le testeur.

L'ensemble des traces d'un IOLTS M peut être caractérisé par un IOLTS déterministe. La

déterminisation permet de supprimer les situations dans lesquelles, à partir d'un état, il existe plus d'une sortie possible aprés un nombre quelconque d'actions internes. En conséquence, il est indispensable de fournir un ensemble de séquences caractérisées par un IOLTS déterministe.

Définition 3.4.1 (Déterminisation d'un IOLTS) *Soit M = (Q_M, A_M, $\longrightarrow_M$,q_M^{init}), un IOLTS. Nous considérons que le IOLTS déterministe résultant de M est : M_{det}. Nous avons donc Trace(M) = Trace(M_{det}), où M_{det} est défini par : M_{det} = (2^{Q_M}, $A_{M_{det}}$, $\longrightarrow_{M_{det}}$, q_M^{init} after ϵ) où :*

1. $A_{M_{det}} = A^O_{M_{det}} \cup A^I_{M_{det}}$,
2. *Les états de M_{det} sont des parties de Q_M,*
3. $P \xrightarrow{a}_{M_{det}} P' \iff P, P' \in 2^{Q_M}$, $a \in A_{M_{det}}$ *et* $P' = P$ *after a.*
4. *L'état initial* $P^{init} = \{q_M^{init}\}$ *after* ϵ *est l'ensemble des états accessibles depuis* q_M^{init} *par les actions internes,*

> **Remarque :** *Les notions de déterminisme et de déterminisation des IOLTS sont les mêmes que celles des automates d'états finis par [HU79] en interprétant les actions internes par des ϵ-transitions.*

Il est naturel de considérer qu'une trace d'un IOLTS est un comportement du système. Pour garder explicitement tous les comportements, il faut inclure les comportements de blocage. Expliciter un blocage, permet de garder "visible" ce comportement lors des tests.

3.4.2 Les blocages

La conception d'un programme est souvent complexe et l'expérience dit parfois qu'un blocage (ou absence d'actions) d'exécution n'est pas forcement une erreur. Un blocage peut signifier un comportement précis s'il est explicitement décrit initialement dans la spécification (par exemple une temporisation). Par contre, il devient une erreur pendant l'exécution des cas de test si la spécification ne le mentionne pas explicitement.

Une spécification précise doit comporter tous les comportements possibles des programmes à tester. Elle doit prévoir d'éventuelles situations de blocages des programmes (généralement ce sont les boucles d'actions internes des programmes). Une nouvelle action, notée δ, est donc ajoutée au IOLTS pour désigner l'une des trois formes de blocages : deadlock, livelock et bloquage de sortie :

– **deadlock**, indique qu'à partir d'un état q, il n'y a plus d'actions possibles. Et donc, par conséquent, il n'y a pas de transitions sortantes de q, c'est un état puits. La situation de deadlock est considérée comme un blocage permanent : Un état $q \in Q$ d'un IOLTS est un deadlock si et seulement si $\forall a \in A$, $q \not\xrightarrow{a}$. Nous notons par deadlock(M) l'ensemble des états puits d'un IOLTS M.

– **livelock**, il existe à partir d'un état q, une séquence d'exécution non vide d'actions internes partant de cet état q et aboutissant dans ce même état q. Ce blocage peut se représenter par une boucle sur un état de l'IOLTS : Un état $q \in Q$ d'un IOLTS est un livelock si et seulement si $q \xrightarrow{\tau\tau^*} q$. Nous notons par livelock(M) l'ensemble des états des chemins non vide d'actions internes (chemin menant de l'état source à ce même état source) d'un IOLTS M.

– **blocage de sortie**, précise qu'a partir d'un état q, il n'existe aucune action de sortie. Les seules actions possibles à partir de q sont les actions d'entrée. Cette situation de blocage est temporaire car la réception d'une action attendue permet de sortir de q : Un état q d'un IOLTS est en blocage de sortie si et seulement si $\forall a \in \mathrm{I} \cup \mathrm{A}^O$, $q \not\xrightarrow{a}$. Nous notons par outlock(M) tous les états d'un IOLTS M ne comportant pas de transition tirable interne ou d'émission.

Exemple 5 — **[Représentation d'une spécification sous forme d'IOLTS incluant un blocage δ]**
Nous représentons par la figure 3.5 une spécification (IOLTS) comportant un blocage de sortie δ au niveau de l'état initial.

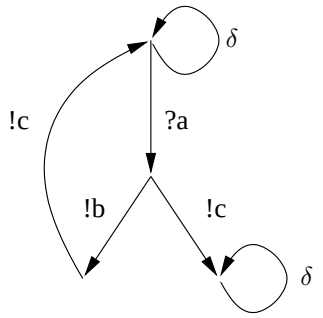

FIG. 3.5 – Une spécification sous forme d'IOLTS incluant un blocage δ

Les blocages conservés par déterminisation

Pour exécuter un cas de test, il faut détecter tous les blocages possibles. Or, la déterminisation ne préserve pas nécessairement les blocages. Pour les conserver, il faut directement les expliciter sur les IOLTS de la spécification (définition 3.4.2). L'action δ est considérée comme une sortie de la spécification, car elle est observable mais non contrôlable par l'environnement. L'IOLTS résultant de l'ajout des actions δ est appelé *IOLTS de suspension* [MV94, Tre96]. La transformation des IOLTS en IOLTS de suspension est décrite par la figure 3.6, pour chaque type de blocage.

Définition 3.4.2 (IOLTS de suspension) *Soit le IOLTS M = (Q_M, A_M, $\longrightarrow_M$,q_M^{init}). Le IOLTS de suspension de M est : Δ(M) = (Q_M, $A_{\Delta(M)}$, $\longrightarrow_{\Delta(M)}$, $q_{\Delta(M)}^{init}$) tel que $A_{\Delta(M)} = A_M \cup \{\delta\}$ avec*

$\delta \in A^o_{\Delta(M)}$ et la relation de transition $\longrightarrow_{\Delta(M)}$ est définie par :

$$\longrightarrow_{\Delta(M)} = \longrightarrow_M \cup \{\ q \xrightarrow{\delta} q \mid q \in \textit{quiescent(M)}\}$$

$\longrightarrow_{\Delta(M)}$ est obtenue à partir de $\longrightarrow_M$ par ajout de boucles $q \xrightarrow{\delta} q$ pour tous les états q de blocage.

La quiescence pour un IOLTS M est l'union de ses blocages :

$$\textit{quiescent(M)} = \textit{deadlock(M)} \cup \textit{livelock(M)} \cup \textit{outlock(M)}$$

NOTE :
Les blocages de sortie sont notés par $!\delta$ (exprimant par exemple une temporisation).

Les exemples de la figure 3.6 représentent quelques automates de suspensions en fonction des blocages.

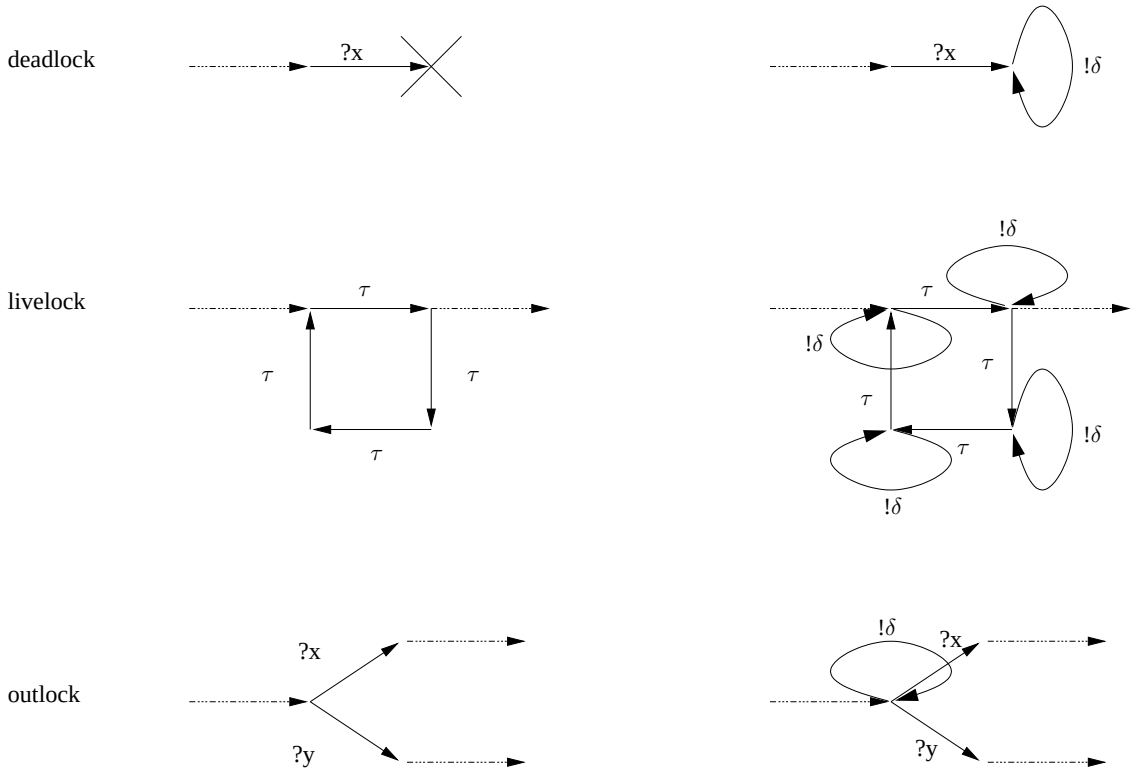

FIG. 3.6 – IOLTS de suspension après la détection et la conservation des blocages

La déterminisation de L'IOLTS après ajout des actions de blocage $!\delta$ caractérise les comportements visibles (avec la conservation de blocages explicites). Ces comportements sont appelés traces suspendues par J. Tretmans [Tre96].

Définition 3.4.3 (Traces suspendues d'un IOLTS) *Les traces suspendues d'un IOLTS M sont notées STrace(M). Elles représentent les traces de son IOLTS de suspension, où, $!\delta$ est considérée comme une action observable.*

$$STrace(M) \equiv Trace(\Delta(M))$$

> **Remarque :** *J .Tretmans [Tre96] définit directement l'IOLTS de suspension comme un IOLTS déterministe résultant de $\Delta(M)_{det}$. Mais il apparaît plus simple de présenter la construction de l'IOLTS de suspension en deux étapes : (1) - ajout des $!\delta$, (2) - déterminisation.*

3.4.3 Objectifs de test

Lors de l'exécution d'un test sur une implantation, une conclusion souhaitée est que l'implantation rende un résultat pour toutes les données fournies. Pour réaliser un tel test, il faut prendre en compte tous les cas possibles. Mais un test exhaustif est difficilement réalisable, car les domaines d'entrée sont souvent trop grands. En pratique, les tests sont construits en fonction de certains critères de test. Un critère doit sélectionner des actions précises à tester pour détecter toutes les implantations ne comportant pas ces actions en n'excluant pas les implantations les comportant. Pour sélectionner (dans une spécification) les actions précises, nous utilisons des objectifs de test. En pratique, les objectifs de test sont modélisés soit par une formule de logique temporelle, soit par un automate étendu, soit par un IOLTS et représentent les actions à tester.

> **Remarque :** *Différents auteurs [DT02, FJJV96] utilisent les objectifs de test comme une description abstraite des tests. Goodenough et Gerhart ont été les premiers a introduire la notion de critère de test [GG75].*

Dans notre cas, un objectif de test est un IOLTS. Pour permettre une sélection plus efficace, nous utilisons deux états distincts marqués (contenus dans l'ensemble des états) servant, soit à accepter, soit à refuser des séquences d'actions de la spécification. Refuser des séquences d'actions permet de limiter l'exploration des comportements de la spécification.

Définition 3.4.4 (Objectif de test : nommé ot) *Un objectif de test est un IOLTS déterministe, complet où sont rajoutés deux états distingués q^{accept} et q^{reject} : (Q, A, $\longrightarrow$, q^{init}, q^{accept}, q^{reject})*

- *Q est un ensemble d'états, avec $q^{accept} \in Q$ et $q^{reject} \in Q$,*
- *A est l'ensemble des actions : A = $A^O \cup A^I \cup \{\delta\}$,*
- *q^{init} est l'état initial,*
- *$\longrightarrow \subseteq Q \times A \times Q$ est un ensemble de transitions,*

- *q^{accept} est l'état d'acceptation dans cet IOLTS, sans successeur par $\longrightarrow$,*
- *q^{reject} est l'état de rejet dans cet IOLTS, sans successeur par $\longrightarrow$.*

Un objectif de test (ot) est un IOLTS sur l'alphabet des actions visibles de la spécification. Il contient deux etats ditingués, "accept" et "reject". Il définit deux langages de traces reconnus :

- Le langage accepté :

$$\text{Trace}_{accept}(\text{ot}) = \{\ \alpha \in (\text{A}^O \cup \text{A}^I \cup \{\delta\})^* \mid \exists\ q^{accept} \in \text{Q},\ q^{init} \xrightarrow{\alpha} q^{accept}\}$$

- Le langage refusé :

$$\text{Trace}_{reject}(\text{ot}) = \{\ \alpha \in (\text{A}^O \cup \text{A}^I \cup \{\delta\})^* \mid \exists\ q^{reject} \in \text{Q},\ q^{init} \xrightarrow{\alpha} q^{reject}\}$$

Nous présentons maintenant comment obtenir un graphe de test (ensemble de cas de test) et les opérateurs de sélection pour produire un cas de test.

3.4.4 Graphe de test : Produit synchrone ($\otimes$)

Un graphe de test est un IOLTS obtenu par produit synchrone entre deux IOLTS : La spécification et l'objectif de test. De manière générale le produit synchrone est défini de la manière suivante :

Définition 3.4.5 (Produit synchrone ($\otimes$) forme générale) *Soit deux IOLTS AC1=(Q_1, A_1, $\xrightarrow{a}_1$, q_1^{init}) et AC2 =(Q_2, A_2, $\xrightarrow{a}_2$, q_2^{init}). Le produit synchrone AC_p = AC1 $\otimes$ AC2 ou AC_p est un tuple (Q_p, A_p, $\xrightarrow{a}_p$, q_p^{init}) s'obtient en composant les deux automates communicants $AC1$ et $AC2$ avec l'opérateur $\otimes$ tel que :*

- $Q_p \subseteq Q_1 \times Q_2$ *est l'ensemble des couples formés des états de AC1 et de AC2,*
- $A_p = A_1 \cap A_2$ *est un ensemble des étiquettes représentant les actions possibles dans* AC_p*,*
- $q_p^{init} = (q_1^{init}, q_2^{init})$ *est le couple initial formé des états initiaux de AC1 et de AC2.*
- $\xrightarrow{a}_p$ *est le plus petit ensemble vérifiant :*

$$\frac{(q_1, q_2) \in Q_p,\ q_2 \xrightarrow{a}_2 q_2^{'},\ q_1 \xrightarrow{a}_1 q_1^{'}}{(q_1^{'}, q_2^{'}) \in Q_p,\ (q_1, q_2) \xrightarrow{a}_p (q_1^{'}, q_2^{'}),\ a\ \in\ A_p}$$

$$\frac{(q_1, q_2) \in Q_p,\ q_1 \xrightarrow{\tau}_1 q_1^{'}}{(q_1^{'}, q_2) \in Q_p,\ (q_1, q_2) \xrightarrow{\tau}_p (q_1^{'}, q_2),}$$

$$\frac{(q_1, q_2) \in Q_p,\ q_2 \xrightarrow{\tau}_1 q_2^{'}}{(q_1, q_2^{'}) \in Q_p,\ (q_1, q_2) \xrightarrow{\tau}_p (q_1, q_2^{'}),}$$

Exemple 6 — **[Représentation d'un produit entre deux systèmes de transitions]**
L'exemple a pour but de représenter de façon générique un produit synchrone entre deux systèmes de transitions nommés AC1 et AC2 en appliquant les règles précédentes.
La figure 3.7 représente le produit entre deux systèmes composés avec :
– AC1 = ({0,1,2,3,4}, { ?a, !b, ?b}, 0)
– AC2 = ({0,1,2,3,4,5}, { ?a, !b, ?b}, 0).

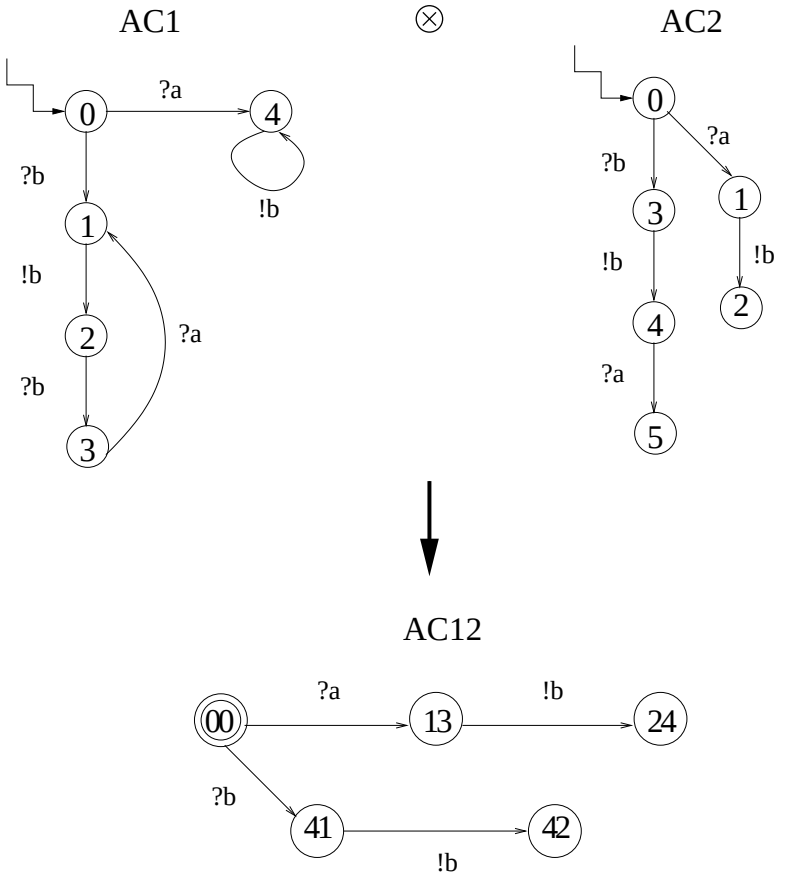

FIG. 3.7 – Représentation du produit synchrone entre deux systèmes de transitions

Le résultat de la composition des deux systèmes en appliquant les règles est aussi un système de transitions : AC12 = ({(0,0), (1,3), (2,4), (4,1), (4,2)}, { ?a, !b, ?b },{(0,0)}).

Produit synchrone pour obtenir un graphe de test à partir d'un objectif de test et d'un automate suspendu associé à un IOLTS

Un graphe de test est un IOLTS obtenu en effectuant le produit synchrone entre une spécification et un objectif de test (figure 3.8 et 3.9).

Soit le IOLTS S = (Q_s, A_s, $\longrightarrow_s$, q_s^{init}), $\Delta(S)_{det}$ = (Q_{det}, $A_s \cup \{\delta\}$ $\longrightarrow_{det}$, q_{det}^{init}) l'automate suspendu associé. Soit ot =(Q_{ot},$A_s \cup \{\delta\}$, $\longrightarrow_{ot}$, q_{ot}^{init}, q_{ot}^{accept}, q_{ot}^{reject}) déterministe et complet sur $A_s \cup \{\delta\}$. Le produit synchrone $\Delta(S)_{det} \otimes$ ot est l'IOLTS P = (Q, A, $\longrightarrow$, q^{init}) muni des deux ensembles "d'états puits" Q^{accept} et Q^{reject} avec :

– Q = $Q_{det} \times Q_{ot}$
– $A \equiv A^I \cup A^O$ avec $A^I \equiv A_s^I$ et $A^O \equiv A_s^O \cup \{\delta\}$
– q^{init} = (q_{det}^{init}, q_{ot}^{init})
– Q^{accept} = $Q_{det} \times q_{ot}^{accept}$
– Q^{reject} = $Q_{det} \times q_{ot}^{reject}$

– $\longrightarrow$ est la relation de transition définie par :

$$\frac{q_{det} \xrightarrow{a} q_{det}^{'} \ , \ q_{ot} \xrightarrow{a} q_{ot}^{'}}{(q_{det},\ q_{ot}) \xrightarrow{a} (q_{det}^{'},\ q_{ot}^{'}) \ , \ q_{det}\ et\ q_{det}^{'} \in Q_{det}, q_{ot}\ et\ q_{ot} \in Q_{ot}}$$

3.4.5 Cas de test : Sélection d'un cas de test

Un cas de test CT est un IOLTS déterministe. Il est représenté par un sous graphe du produit synchrone Spec $\otimes$ Ot et doit satisfaire la condition de contrôlabilité. Les systèmes modélisés par des IOLTS ne sont pas toujours contrôlables par l'environnement, et pour une même entrée, ils peuvent produire plusieurs sorties différentes. Cette notion est appelée non déterminisme observable (définition 1.1.17). Les cas de test qui décrivent les interactions entre le testeur et l'implantation doivent prévoir ce non déterminisme. Ils doivent prévoir l'observation de toutes les sorties de l'implantation, correctes ou non, par complétion sur les entrées. Par contre, ils contrôlent leurs propres sorties, dans un état, si une sortie est possible, c'est la seule action possible. Un cas de test est étendu par des diagnostics exprimés sous forme de verdicts.

Les verdicts :

PASS : Le verdict "$Pass$" indique la conformité de l'implantation (pour le cas de test exécuté).

FAIL : Le verdict "$Fail$" indique la non conformité : Le comportement de l'implantation observé par le testeur n'est pas un comportement conforme à la spécification (par exemple : une sortie de

l'implantation n'est pas spécifiée).

INCONCLUSIVE : Le verdict "$Inconclusive$" n'est pas une non conformité. Le comportement du testeur qui mène à cet état se termine par une réception et correspond à un comportement de la spécification qui ne satisfait pas l'objectif de test, mais qui n'est pas forcément contradictoire avec la spécification.

Définition 3.4.6 (Cas de test) *Un cas de test CT = (Q_{ct}, A_{ct}, $\longrightarrow_{ct}$, q_{ct}^{init}) est un IOLTS déterministe muni de trois sous-ensembles distincts d'états sans successeur Pass, Fail et Inconclusive $\in Q_{ct}$ caractérisant les verdicts, avec : $A_{ct} = A_{ct}^{O} \cup A_{ct}^{I}$ avec $A_{ct}^{I} \subseteq A_{s}^{I}$ CT n'émettant que des entrées de la spécification. Et $A_{ct}^{O} = A_{iut}^{O} \cup \{\delta\}$, CT prévoyant la réception de toutes les sorties de l'implantation :*

1. *Nous supposons que CT (cas de test) est un IOLTS contrôlable par le testeur, c'est à dire qu'il n'y aura jamais le choix entre des sorties ou entre des entrées et des sorties :*

$$\forall q \in Q_{ct}, (\exists a \in A_{ct}^{I}, q \xrightarrow{a}_{ct} \Longrightarrow \forall b \in A_{ct}, \ (b \neq a \Longrightarrow q \not\xrightarrow{b}_{ct})).$$

2. *Tout états permettant une entrée est complet en entrées :*

$$\forall q \in Q_{ct}, (\exists a \in A_{ct}^{I}, q \xrightarrow{a}_{ct} \Longrightarrow \forall b \in A_{ct}, q \xrightarrow{b}_{ct})).$$

3. *Les états Fail et Inconclusive sont accessibles en un pas si il n'y a des entrées permettant de les atteindre :*

$$\forall (q,a,q') \in \longrightarrow_{ct}, (q' \in Fail \cup Inconclusive \Longrightarrow a \in A_{ct}^{O}).$$

4. *De tout état un verdict est accessible :*

$$\forall q \in Q_{ct}, \exists \alpha \in A_{ct}^{*}, \exists q' \in Pass \cup Fail \cup Inconclusive, q \stackrel{\alpha}{\Longrightarrow} q'.$$

Définition 3.4.7 (Suite de test) *Nous appelons suite de test (ST) est un ensemble (fini ou infini) de cas de test (CT) : ST = $\{CT\}$.*

Dans la pratique, nous allons voir comment obtenir un ensemble de cas de test. Nous présentons avec l'exemple 7 une illustration de construction des cas de test.

Exemple 7 **— [Sélection d'un cas de test à partir d'un graphe de test construit entre une spécification et un objectif de test]**
Dans cet exemple nous proposons de décrire une chaîne complète afin d'obtenir un ensemble de cas de test. Pour réaliser le produit synchrone (donné par la figure 3.8) nous disposons du modèle de la spécification et d'un objectif de test qui comporte les deux états "$accept$" et "$reject$". Le cas de test résultant est donné par la figure 3.9.

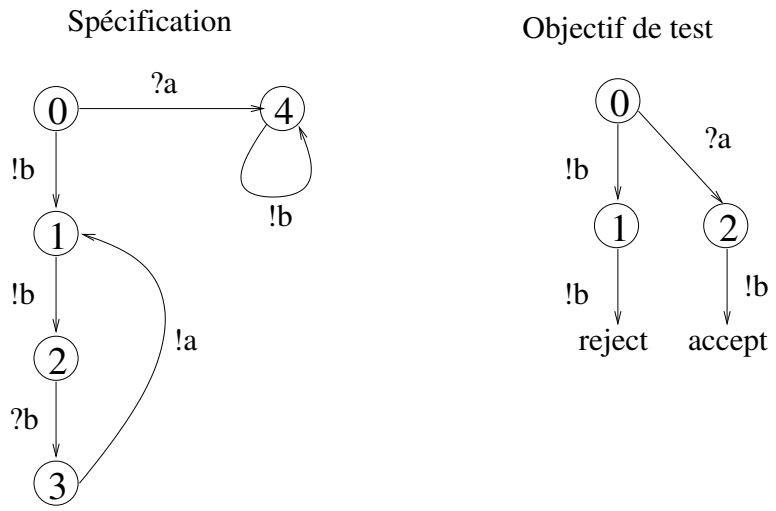

FIG. 3.8 – Spécification et objectif de test

Après application des règles de composition (produit synchrone) entre la spécification et l'objectif de test, nous obtenons un graphe complet (contenant les états distingués de l'objectif de test) représentant les séquences relatives à l'objectif de test. Ce graphe de test est représenté par : GT = $\{\{(00), (11), (22), (1reject), (3accept)\}, \{ ?a,\ !b,\ ?b \}, (00)\}$.

À partir de ce graphe de test GT, nous extrayons (par un algorithme de recherche des états $accept$ [Jér04]) des séquences d'exécution auxquelles nous leur ajoutons les verdicts d'exécution. La séquence comporte tous les états nécessaires pour atteindre un état $accept$ de GT en partant de l'état initial de GT. Les séquences extraites sont les traces minimales s'arrêtant au premier état $accept$ rencontré. La définition 3.4.6 impose que les verdicts soient placés sur des états puits. Ainsi, un cas de test comporte aucun état puits sans verdict. En d'autres termes, le cas de test produit est un graphe acyclique dont toutes les feuilles portent un verdict. Chacune des séquences ainsi obtenues devient un cas de test exécutable par le testeur. Tous les chemins ne permettant pas l'accés à un état $accept$ seront réduits en un état puits. C'est à dire que les transitions sortant des états ne memant pas à un état $accept$ seront coupées et ce à partir d'un menant à un état $accept$. Un cas de test est donc extrait du graphe initial GT de tel sorte que chaque état du cas de test comporte au moins un successeur en direction d'un état accept (qui sera notre verdict $Pass$ et donc état sans sucesseur). Si un état traversé comporte d'autres actions ne memant pas un état $accept$ le verdict de l'état sera Inconclusive. Dans les autres cas les transitions méneront à des états dont le verdict est Fail.

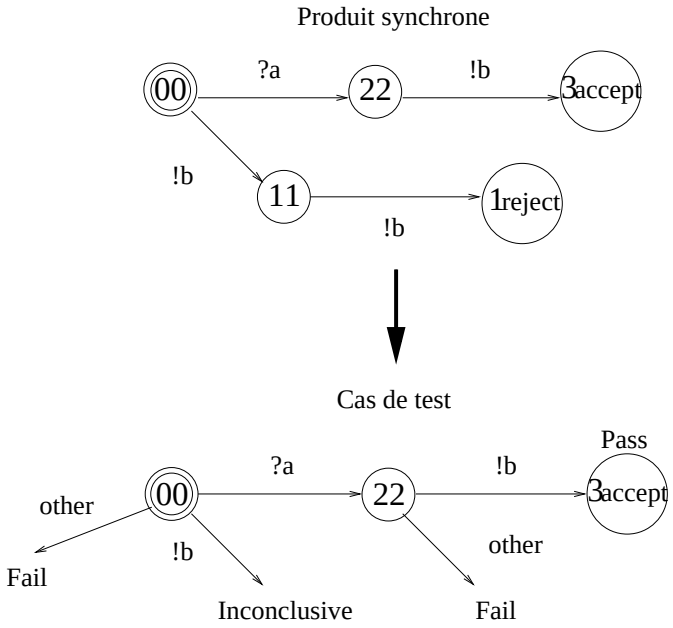

FIG. 3.9 – Sélection d'un cas de test

En résumé, le cas de test que nous obtenons dans la figure 3.9 a été construit à l'aide du produit synchrone $\otimes$ (définition 3.4.5). En appliquant l'opérateur de sélection nous extrayons les cas de test. Pour cet exemple, nous gardons la séquence menant à l'état $(3accept)$ en partant de l'état initial. Une seule séquence est gardée puisqu'il n'existe qu'un chemin possible (dans cet exemple) entre ces états. Si plusieurs chemins étaient possibles, il en découlerait plusieurs cas de test. Le chemin menant à l'état $reject$ n'appartient plus au cas de test. En fait, seule la transition $!b$ issue de l'état (0,0) est conservée. Cette transition est munie d'un verdict Inconclusive qui, lors de l'exécution, indiquera au testeur que cette partie de comportement n'est pas à explorer. Le verdict inconclusive est important, car si pendant l'exécution l'implantation fournit au testeur l'action $!b$, le cas de test doit être recommencé (mais sans émettre de verdict Fail ou Pass indiquant la conformité ou non de l'implantation testée).

Dans la représentation du cas de test à fournir au testeur, un premier verdict est donc possible, c'est le verdict $Pass$ associé à l'état $(3, accept)$. Pour obtenir un état $Pass$ pendant l'exécution du cas de test, le testeur doit prendre la transition contenant l'action $!b$. Pour guider l'exécution nous ajoutons sur le cas de test une action other. En général, l'action other est générique et regroupe toutes les actions autres que l'action $!b$. Cette action, nous permet d'ajouter au cas de test le verdict $Fail$. Ce verdict, ainsi placé, permet pendant l'exécution du cas de test par le testeur d'observer qu'une action non spécifiée (action différente de $!b$) est réalisée par l'implantation. Et donc, de conclure que l'implantation n'est pas conforme à la spécification de référence en fonction du cas de test exécuté par le testeur.

NOTE :

L'IOLTS de l'objectif de test (ot) est complet, P = $\Delta(S)_{det}$ × ot est bisimilaire à $\Delta(S)_{det}$ (intuitivement il préserve tous les comportements). Comme $\Delta(S)_{det}$ et ot sont déterministes P l'est aussi. La bisimulation est donc l'équivalence de traces Trace(P) = STrace(S). De plus, les traces suspendues de S acceptées (respectivement refusées) par ot sont exactement les séquences acceptées (resp refusées) par P sur ses états q^{accept} (resp q^{reject}).

$$\text{Trace}_{accepter}(\text{P}) = \text{STrace(S)} \cap \text{Trace}_{accept}(\text{ot})$$
$$\text{Trace}_{rejeter}(\text{P}) = \text{STrace(S)} \cap \text{Trace}_{reject}(\text{ot})$$

3.5 Modèle d'exécution

L'exécution d'un cas de test CT, sur une implantation (IUT), est modélisée par le produit synchrone entre CT et $\Delta(IUT)$, noté : CT || Δ(IUT). La synchronisation s'effectue avec les actions visibles communes de CT et de l'IUT (y compris les actions !δ).

De manière plus générale la composition paralléle D || O entre deux IOLTS D = $(Q_{\text{D}},\ A_{\text{D}},\ T_{\text{D}},\ q_{\text{D}}^{init})$ et O = $(Q_{\text{O}},\ A_{\text{O}},\ T_{\text{O}},\ q_{\text{O}}^{init})$ s'exprime par les règles (vues en section 3.4.4) :

$$(q_D^{init}, q_O^{init}) = q_{D||0}^{init}$$

$$\frac{(q_O, q_D)\ \in\ Q_{D||0}\ q_D\ \xrightarrow{a}\ q_D^{'}\ ,\ q_O\ \xrightarrow{a}\ q_O^{'}}{(q_D,\ q_O)\ \xrightarrow{a}\ (q_D^{'},\ q_O^{'})(q_O^{'}, q_D^{'})\ \in\ Q_{D||0}}$$

$$\frac{(p,q)\ \in\ Q_{D||0}\ p\xrightarrow{\tau}_D p^{'},\ \tau \in I_D}{(p,q)\xrightarrow[D||0]{\tau}(p^{'},q),(p^{'},q)\ \in\ Q_{D||0}} \qquad \frac{(p,q)\ \in\ Q_{D||0}\ q\xrightarrow{\tau}_O q^{'},\ \tau \in I_O}{(p,q)\xrightarrow[D||O]{\tau}(p,q^{'}),\ (p,q^{'})\ \in\ Q_{D||0}}$$

L'exécution d'un cas de test CT sur une implantation n'est en principe jamais bloquée. Dans le cas d'un blocage, CT possède un état de blocage qui, s'il est atteint pendant l'exécution, produira le verdict associé. Les cas de test comportent les verdicts $pass$, $fail$ ou $Inconclusive$. Lors d'une exécution d'un cas de test, l'implantation est conforme si nous obtenons $pass$, elle est non

conforme si nous avons $fail$ et l'implantation n'est pas dans une partie à tester si nous obtenons $inconclusive$.

Les actions internes de l'implantation étant non visibles du cas de test, les observations faites sont des traces finies maximales de CT || $\Delta(IUT)$ (les boucles infinies étant évitées en utilisant une temporisation globale d'exécution) du cas de test, c'est à dire des séquences d'actions $\alpha \in A^{CT^*}$ telles que $Act\big($ (q_0^{CT} ,$q_0^{\Delta(IUT)}$) after $\alpha\big)$ = $\emptyset$ dont le verdict d'exécution se détermine par :

Définition 3.5.1 (Verdicts) *Soit α une trace finie maximale CT || $\Delta(IUT)$, les verdicts en fonction de α et de CT sont :*

verdict(α) = Fail $\equiv$ CT after $\alpha \subseteq$ Fail
verdict(α) = Pass $\equiv$ CT after $\alpha \subseteq$ Pass
verdict(α) = Inconclusive $\equiv$ CT after $\alpha \subseteq$ Inconclusive

La définition précédente exprime des verdicts possibles pour des exécutions particulières de cas de test sur une implantation. Nous pouvons regarder l'ensemble des exécutions possibles et des verdicts associés pour définir un verdict global. A cause du non déterminisme observable de l'implantation, un même cas de test peut produire plusieurs exécutions possibles pour une même implantation et "probablement" avec des verdicts différents. En fonction de ce constat, nous donnons maintenant trois verdicts (en fonction des verdicts précédents) pour définir qu'une implantation peut soit rejeter un test, soit avoir la possibilité d'accepter un test, soit de produire un verdict inconclusif (selon un cas de test donné) :

Définition 3.5.2 (Verdicts selon rejetée, acceptée, inconclusive) *Considérons l'implantation IUT et le cas de test CT :*

IUT est **rejetée** *par CT si est seulement si $\exists\alpha \in$ Trace(CT || $\Delta(IUT)$) : verdict(α) = Fail*

IUT est **acceptée** *par CT si et seulement si $\forall\alpha \in$ Trace(CT || $\Delta(IUT)$) : verdict(α) = Pass*

IUT est **inconclusive** *par CT si et seuelement $\exists\alpha \in$ Trace(CT || $\Delta(IUT)$) : verdict(α) = Inconclusive*

Un cas de test est supposé nous renseigner sur la conformité de l'implantation vis à vis de la spécification. Dans ce cas il faut qu'un verdict Fail implique la non-confomité et que toute non conformité doit pouvoir être détectée par un cas de test. Ce lien est établi avec la définition suivante :

Définition 3.5.3 (exhaustivité, correction et complétude) *Soit I l'ensemble des implantations compatibles à une spécification S.*

– *Une suite de test ST est exhaustive pour S et ioco si et seulement si* $\forall IUT \in I$, $\neg$*(IUT* **ioco** *S)* $\Longrightarrow \exists CT \in ST$*, IUT est rejetée par CT.*

– *Une suite de test ST est correcte si* $\forall IUT \in I$*, (IUT* **ioco** *S)* $\Longrightarrow \neg$*(IUT est rejetée par CT) d'une suite de test ST vis-à-vis de S et* **ioco** *nous avons que les IUT non conformes sont rejetées :* $\forall IUT \in I$*, IUT est rejeté par CT* $\Longrightarrow \neg$*(IUT* **ioco** *S).*

– *Une suite de test ST est complète si elle est correcte et exhaustive.*

NOTE :
Les termes d'exhaustivité, de correction et de complètude peuvent différer dans la communauté du test. La raison est que ces termes ne sont pas standardisés. Tretmans utilisera ces termes en employant sound pour la correction. Jèron [Jér04] utilisera le terme de non biais pour définir la correction. Tandis que Bernot, Gaudel et Marre [BGM91], dans la thèorie du test algébriques, utilisent le terme de valide pour l'exhaustivité. Dans ce dernier cas, exhaustif désigne la suite de test composé de tous les termes clos des axiomes de l'algèbre. En test structurel, le terme approprié à correction et complet a un sens relatif à un critère de couverture.

> **Remarque :** *Dans la construction des cas de test, nous constatons que les séquences dérivent directement de la spécification. Or, pour faire exécuter ces séquences avec l'implantation il est nécessaire de changer les entrées en sorties et les sorties en entrées. Nous appelons cette réalisation l'image miroir d'une séquence d'exécution. Si habituellement, cette opération est réalisée plustôt dans le processus de génration de test, pour simplifier la pésentation, nous avons choisi de ne pas appliquer l'opération miroir lors de la génération de cas de test.*

> ***Définition 3.5.4 (Représentation de l'image miroir d'un IOLTS)*** *Soit* $S = (Q_S, A_S, T_S, q_S^{init})$ *un IOLTS, l'image miroir de S est* $\overline{S} = (Q_s, \overline{A}_s, T_s, q_s^{init})$ *avec :* $\overline{A} = \overline{A_s^I} \cup \overline{A_s^O} \cup \tau$ *et* $\overline{A_s^I} = A_s^O$ *et* $\overline{A_s^O} = A_s^I$*. Cela signifie que les entrées (resp : les sorties) du IOLTS S deviennent les sorties (resp : les entrées) du IOLTS* $\overline{S}$*.*

3.6 Explosion des états

Un problème peut intervenir lors de la construction : "l'explosion des états". Il signifie que la taille du graphe de test construit est trop importante pour la stoker, voire la traiter. Pour l'éviter lors des constructions des graphes de test, différentes améliorations ont été envisagées :

1. Des travaux d'analyses statiques préalables peuvent limiter le nombre d'états dans les modèles [BFG00, BFG99a] (pour réduire au maximum les comportements).
2. Il est possible de restreindre la construction du graphe de test par analyse du contrôle, c'est à dire de restreindre une spécification tout en conservant ses comportements observables. Cette analyse est fondée sur des techniques de slicing [Ghi02, BFG00, Wei79].
3. Il existe également des algorithmes dits "à la volé" [Mor00] traitant des compositions entre

les modèles. Ces algorithmes ont la particularité de ne conserver en mémoire que les cas de test sélectionnés et non les graphes de constructions.

3.7 Conclusion

Nous avons proposé dans le chapitre 1 différents modèles (les IOLTS, les automates communicants,. . .) pour décrire les comportements d'exécution de programmes. Puis dans le chapitre 3, nous avons présenté une méthode de génération (selon l'outil TGV) de cas de test de conformité pour vérifier la conformité de systèmes distribués (les protocoles de communication) selon une spécification de référence. La conformité est établie suivant la relation de conformité **ioco** [Tre96]. La méthodologie du test de conformité **ioco** a été largement automatisée dans [Jér02, BFdV+99]. L'exécution des cas de test est faite par un testeur en interaction avec l'implantation.

Nous proposons maintenant d'étendre l'approche basée sur les modèles décrit pour le test de conformité au cadre du test de robustesse.

Selon la définition de IEEE : Le test de robustesse vise à garantir qu'un système conserve un fonctionnement acceptable en présence d'aléas (pannes, dysfonctionnements, entrées invalides, . . .) susceptibles d'affecter le système ou l'un de ses composants.

Nous définirons cette proposition et l'adapterons pour tester la robustesse des logiciels.

Avant de proposer notre approche, nous présentons certaines contraintes présentes dans la méthode de génération de cas de test de conformité.

Contraintes de la méthode de génération de cas de test de conformité

Pour la méthode du test de conformité présentée, nous faisons un premier bilan et établissons une liste de ses contraintes :

1. La spécification : c'est un élément important de la génération, elle doit être exhaustive (dans le sens où la spécification précise tous les comportements pour décrire les fonctionnalités des implantations) et reste l'élément essentiel de la génération.

2. La relation à tester : en réalité seule "la relation de conformité" reste et fait souvent partie intégrante des outils. Le test de conformité est réduit à la relation incluse dans les outils.

3. Les propriétés : Seules les propriétés fonctionnelles de sûreté peuvent être éventuellement testées. Il n'est pas possible de faire intervenir un utilisateur pour vérifier une propriété de vivacité (bornée), voire non fonctionnelle.

4. Tous les cas de test sont produits à partir d'un environnement idéal et non corruptible implicitement donné dans les modèles de spécifications. Généralement, ce sont les valeurs des domaines entrant dans le système qui représentent l'environnement des composants. Ces domaines (comprenant les valeurs des différents paramètres) sont supposés fixes et ne peuvent (doivent) pas évoluer lorsqu'un système (voire un composant) est dans son environnement d'application.

Pour garder un principe de génération de cas de test, basé sur les modèles, nous ne supprimons pas la spécification. Nous supposons par contre que la spécification donnée peut être une spécification "partielle". La contrainte est que la spécification partielle ne soit pas trop éloignée des exigences initiales et donc par conséquent des implantations à tester. Car, cette spécification partielle de référence, si elle existe, reste un guide pour construire des cas de test de robustesse.

Nous considérons qu'une implantation (contrairement au test de conformité) n'évolue plus dans un environnement avec des paramètres fixés et invariants. Nous envisageons de tester des implantations, même si l'environnement évolue. Cette évolution ce traduit par l'ajout de certains aléas dans les modèles. Nous allons voir comment l'environnement peut fournir certaines fautes influentes sur le comportement des systèmes implantés. Les fautes sont ensuite intégrées dans le modèle de spécification de référence pour donner une spécification mutée.

Nous ajoutons le fait de pouvoir observer plusieurs propriétés extérieures (en plus de la spécification de référence) . Cela nous donne la possibilité de décrire et modéliser les différents types de propriétés envisagés. Puis, nous allons essayer de voir, si il existe des propriétés vérifiées par la relation de robustesse et non préservées par la relation de conformité **ioco**.

Nous allons voir comment redéfinir les verdicts $Fail$ et $Pass$ lors d'une exécution d'un cas de test pour obtenir la satisfiabilité ou non d'une implantation vis-à-vis d'une propriété.

Perspectives des prochains chapitres

Dans la suite du document, nous présentons comment formaliser la spécification "partielle" de référence, les types et modèles de propriétés envisagées, une relation de test (différente de la relation de conformité **ioco**) et comment prendre en compte des aléas (induits par un environnement).

Tous ces aspects donnent l'occasion de définir une nouvelle architecture de test, en incluant un modèle de faute pour enfin redonner une démarche de génération et exécution de cas de test de robustesse. Pour commencer, nous présentons un état de l'art des techniques existantes pour générer des tests de robustesse, aussi bien dans les domaines matériels que logiciels.

Deuxième partie

De la conformité à la robustesse

Chapitre 4

Etat de l'art

Les travaux présentés dans cette partie II introduisent le test de propriétés et le test de robustesse de programmes. Notre réflexion sur la notion de robustesse a débuté lors d'une action spécifique (AS23, à l'initiative du CNRS) en collaboration avec les laboratoires LAAS (Toulouse), Labri (Bordeaux), IRISA (Rennes), LRI (Orsay-Paris) et Vérimag (Grenoble). L'objectif de cette AS a été d'aborder le problème du test de robustesse pour des systèmes critiques ou embarqués. L'action nous a permis de définir la robustesse pour ce type de systèmes et de préciser ce que voulait dire "tester la robustesse". La robustesse est la capacité d'un système à fonctionner de manière acceptable en présence d'aléas. Un aléa est vu comme une faute ou un changement de paramètres dans un système. Nous avons ensuite proposé une méthode pour générer des cas de test de robustesse.

Nous proposons dans ce chapitre 4, de faire un état de l'art partiel des travaux, des théories, et des outils existants dans le domaine de la génération et de l'exécution de cas de test de robustesse. Notre discussion se limite à une simple présentation des domaines matériels, systèmes d'exploitations, et logiciels.

Parler de test de robustesse par rapport au test de conformité nous impose d'introduire deux nouvelles notions :

1. La première est la notion de condition de robustesse que nous traduisons par les propriétés de robustesse à tester.
2. La deuxième est la notion de faute que nous traduisons par l'ensemble des fautes possibles en fonction des types d'implantation et des environnements de test. Les fautes peuvent être classées dans un modèle de faute. Elles varient selon les composants des implantations à tester. Elles représentent généralement des pannes, des dysfonctionnements, des défaillances, etc.

Les conditions de robustesse peuvent être d'ordre très général, elles décrivent par exemple : "qu'un système ne possède aucune erreur d'exécution", ou "qu'un système n'a pas de situation de deadlock possible". Ces conditions peuvent s'étendre à des propriétés plus spécifiques pour exprimer :

”qu’il est possible pour un système de sortir d’une situation d’état d’erreur afin de revenir dans une situation plus nominale” (situation acceptable d’utilisation), ou ”qu’un système reste toujours dans des conditions de fonctionnement nominales”, ou encore ”certaines ressources des systèmes restent toujours disponibles”.

Le test de ces conditions de robustesse a cependant (et jusqu’à présent) été peu étudié dans les domaines logiciels. Nous le retrouvons plus généralement dans la communauté ”matériel” ou ”système d’exploitation”. Les travaux suivants ont pour but de classer les modes de défaillances dans les systèmes exécutifs. Nous les retrouvons par exemple pour les micronoyaux, les systèmes d’exploitation ou pour les intergiciels. Le test de robustesse dans ces domaines se réduit à une injection de fautes au niveau des systèmes. Un verdict est donné en fonction de l’effet de la faute injectée dans le système :

– L’outil Fuzz [MKL$^+$95, ABS94, MFS90, Spi88] vérifie la robustesse de certains systèmes d’exploitation (Unix, Window NT). Les descriptions des machines (les architectures matérielles) où sont implantés les systèmes sont connues initialement. Par contre ni les spécifications, ni le code du système ne sont 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. Certaines séquences de caractères produites peuvent être faites de façon aléatoire. Initialement le modèle de faute est vide et dès qu’une faute est provoquée la commande relative à cette panne est incluse dans le modèle de faute. Le test effectué est comparable à celui fait pour le test de conformité. Dans son principe, les fautes sont injectées en tant que commandes système. Si le résultat de la commande (donnée) est correct, c’est à dire qu’elle correspond à une exécution (attendue), alors le système est robuste pour ce type de chaînes. Si le résultat n’est pas correct ou inattendu, c’est à dire que le système essaye d’exécuter la commande (mais boucle) ou stoppe le système, alors le système n’est pas robuste pour la chaîne donnée. Une classification des chaînes de caractères testées peut être établie en fonction des résultats obtenus et gardée pour tester les prochaines versions de systèmes.

– Dans l’outil Ballista [KD99, KJS98, JSD$^+$97], les cas de test sont des combinaisons d’entrée valides ou incorrectes portant sur des parties particulières d’un système. Par exemple, un cas de test peut représenter ”les appels les plus fréquemment utilisés dans un système d’exploitation”. Le cas de test est ensuite donné au système sous forme de commandes. Le verdict propose plusieurs critères d’échecs : les accidents, les reprises de traitement, les arrêts, etc, selon les résultats obtenus.

– L’outil Riddle [MV99, GSS98], emploie une grammaire d’entrée pour produire des combinaisons correctes, ou incorrectes, ou des entrées aux limites couvrant au mieux les fonctionnalités du système à tester. Il fournit des critères d’échec appartenant à des parties bien précises des systèmes. Par contre une étude des systèmes utilisés doit être faite au préalable.

– L’outil MAFALDA (Microkernel Assessment by Fault injection Analyse and Design Aid [J.H99]) est un outil d’analyse et d’aide à la conception de systèmes exécutifs sûrs de fonctionnement à bass de micronoyaux (du commerce). Le principe est que la sûreté de fonctionnement est basée sur l’analyse de comportement en présence de fautes. Pour la méthode de test, il convient

alors de compléter les mécanismes internes de détection et de confinement d'erreur. L'outil MAFALDA permet d'évaluer la sûreté de fonctionnement d'un micronoyau COTS[1] par injection de fautes. L'outil manipule les aspects de bas niveau tels que l'utilisation de la mémoire, ou des registres. Cet outil considère les fautes comme un aspect de la réalité non représentées par un modèle. Pour caractériser les modes de défaillances d'un micronoyau, une faute est (par exemple) la corruption bit-flip d'un mot mémoire [AFRS02]. D'un point de vue pratique, la mise en œuvre du test s'appuie sur un outil d'analyse en ligne de l'activité du système, pour déterminer où et quand injecter les fautes de bas niveau [THZ+99]. Enfin l'outil MAFALDA a pour but d'observer les différents comportements (erreurs détectées, blocages, défaillance de l'application) et permet d'analyser les données obtenues. Les résultats de l'analyse sont présentés sous forme statistique.

Après avoir vu quelques outils pour les domaines matériels, systèmes et systèmes d'exploitation, nous présentons maintenant plusieurs approches existantes pour automatiser la génération et l'exécution de cas de test de robustesse dans le domaine du logiciel. Les méthodes pour prendre en compte la robustesse des logiciels sont, pour la plupart, des méthodes basées sur les injections de fautes. C'est à dire qu'elles consistent à fournir au système à tester des cas de tests contenant des entrées incorrectes pour le logiciel. Ces fautes sont généralement choisies parmi un ensemble de fautes possibles (dans un modèle de fautes). Les fautes sont censées exhiber les échecs de robustesse. Cependant, les techniques diffèrent en fonction des entrées choisies. Nous passons en revue certaines d'entre elles que nous considérons représentatives :

– Lorsque le code source du système est disponible, la génération des cas de test appropriés pour examiner la robustesse d'un logiciel est améliorée. Il devient par exemple possible d'employer des techniques d'analyses statiques pour choisir les meilleures entrées capables de couvrir le plus grand nombre de combinaisons et de paramètres possibles (la sélection des cas de test est faite en utilisant des relations d'équivalence). Cette méthode est exploitée par exemple dans l'outil de test JCRASHER [CS00]. Le but est de détecter les exceptions d'exécution non déclarées dans des programmes Java (la détection est faite au niveau des appels de méthodes publics). Cet outil vise seulement des erreurs particulières comme les combinaisons imprévues de paramètres lors des appels de méthodes. Le verdict est la possibilité ou l'impossibilité d'un appel de la méthode en fonction des paramètres.

– La robustesse d'un logiciel en présence de fautes est liée à des propriétés de tolérance aux fautes. Différentes méthodes peuvent être utilisées pour classer les fautes [LAC+96, BDD+91]. La nature des fautes conduit à distinguer les fautes accidentelles et les fautes intentionnelles. Les fautes intentionnelles sont notamment des intrusions et des logiques malignes. En termes de test de robustesse, les tests de pénétration [Wei95] utilisent des fautes intentionnelles. Ces tests ciblent les manipulations non-autorisées de l'information.

Quelques techniques de test peuvent s'effectuer en utilisant des spécifications abstraites. Les spécifications représentent les comportements du logiciel. Les spécifications sont utilisées pour produire les cas de test. Le test peut être conduit dans le but de donner des informations sur le com-

[1] Commercial Off-The-Self

portement des logiciels confrontés à des fautes (erreurs de manipulations, données inopportunes, . . .) :

– Dans la description de [Hel97], la méthode STRESS (Systematic Testing of Robustness by Evaluation of Synthesized Scenario) présente les principes de FOTG [HEG98] (Fault Oriented Test Generation). Cette méthode 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. Les cas de test sont perçus comme des scénarii à réaliser. Ils incluent les paternes : <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 connexions et N les Noeuds de connexions, 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 génération est le suivant : À partir d'une erreur contenue dans une spécification d'un protocole (comme une perte de message, ou un accident de noeud, etc), la méthode FOTG parcourt la spécification (vers l'avant) 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 isolé, la méthode produit (par un parcours en arrière) la séquence de cas de test. Cette approche semble bien adaptée pour les protocoles réseaux qui sont normalement insensibles aux défaillances. Par contre, elle traite seulement des erreurs "simples" (produites une par une).

– Une technique semblable est proposée dans le projet PROTOS [PRO01]. Cette méthode décrit, à niveau élevé et abstrait un comportement de logiciel (à tester), représentant l'ensemble des interactions correctes. À partir de ces interactions, certaines entrées anormales peuvent être mises en évidence. Les cas de test sont alors produits en effectuant des simulations des entrées anormales isolées sur la description abstraite.

Une forme de test de robustesse peut se retrouver dans le test *des données aux limites*. Il apparaît que les valeurs proches des limites des domaines d'entrée et de sortie ont un bon pouvoir de détection de fautes. L'introduction des contraintes permet un traitement des valeurs limites. Les techniques de programmation logique avec contraintes sont utilisées pour la génération de cas de test fonctionnels. Dans ce cadre, les contraintes permettent un traitement des valeurs limites plus réalistes ainsi que la détection précoce de sous-problèmes ne possédant pas de solution dans le domaine des valeurs autorisées [Meu98]. La résolution du système de contraintes restreint progressivement le domaine de variables à des valeurs qui correspondent à un cas de test permettant d'atteindre les points limites sélectionnés [AML99]. L'exécution des cas de test permet de définir le comportement d'un logiciel en présence des valeurs limites. Cependant, une faiblesse de ces techniques est la difficulté à déterminer les entrées provoquant les valeurs aux limites du domaine de sortie (au vu des spécifications). Les limites données ont souvent des valeurs particulières (comme les valeurs nulles ou les valeurs extrêmes des intervalles).

Chapitre 5

Génération et exécution de cas de test pour le test de propriétés

L'objectif de la partie II est de prolonger au cadre du test de robustesse, l'approche du test de conformité basée sur les modèles. Nous formalisons, dans ce chapitre, une méthode pour générer et exécuter automatiquement des cas de test de robustesse en utilisant les modèles d'automates communicants. Le test de robustesse a pour but de vérifier si les propriétés (de robustesse) données sont préservées lors de l'exécution des programmes testés. La particularité de notre approche est de prendre en compte dans la description des modèles de spécification, les éventuels aléas (fautes) induits par l'environnement de fonctionnement des programmes à tester.

Pour réaliser le travail sur la robustesse des programmes, nous présentons d'abord le test de propriétés [FMP04]. Nous faisons ce choix, car la génération et l'exécution de cas de test de propriétés seront les bases du test de robustesse. En fait, nous supposons que l'activité de test de robustesse est une mise en œuvre du test de propriétés en intégrant dans les modèles de référence certains aléas de l'environnement. Mais aussi parce que le test de propriétés de programmes peut être une activité indépendante du test de robustesse et peut être appliquée éventuellement pour d'autres approches.

Nous proposons une méthode pour générer automatiquement des cas de test de propriétés, en définissant : les modèles de spécifications et d'implantations, les "classes" de propriétés envisagées avec leurs modélisations, une architecture de test, et enfin une relation de *satisfiabilité* établie entre une propriété et une implantation. Puis, nous présentons les éléments nécessaires (comme la description des aléas) pour adapter la génération de cas de test de propriétés à celle du test de robustesse.

5.1 Présentation du test de propriétés

Pour nous, le test de propriétés est une méthode de validation qui consiste à faire des expérimentations sur une implantation (IUT). Cette méthode permet de vérifier, si une propriété $\mathcal{P}$ donnée (une condition de fonctionnement) est préservée par le comportement de l'implantation

lors de son exécution. Une expérimentation consiste à faire interagir sur l'implantation (à l'aide d'un testeur) un cas de test produit automatiquement à partir d'une propriété et d'une spécification. Nous supposons que le code de l'implantation à tester n'est pas connu, et considérons que le test de propriétés est un test de type *boîte noire*. Les comportements de l'implantation sont observés à travers des interfaces appelées PCO (Points de Contrôle et d'Observations). La relation définie entre l'implantation et la propriété donnée est nommée "la relation de satisfiabilité" : IUT $\xrightarrow{satisfait?} \mathcal{P}$.

Nous verrons que toutes les propriétés ne peuvent pas être vérifiées (cette restriction est due en partie aux caractéristiques même de la notion de test). Mais nous essayerons d'apporter des solutions (au niveau modélisation des propriétés et exécution des cas de test) pour tester le plus grand nombre de propriétés. Sans faire une classification exhaustive des propriétés $\mathcal{P}$, nous regarderons qu'elles sont les "classes" de propriétés envisagées.

La figure 5.1 représente une architecture pour effectuer du test de propriétés.

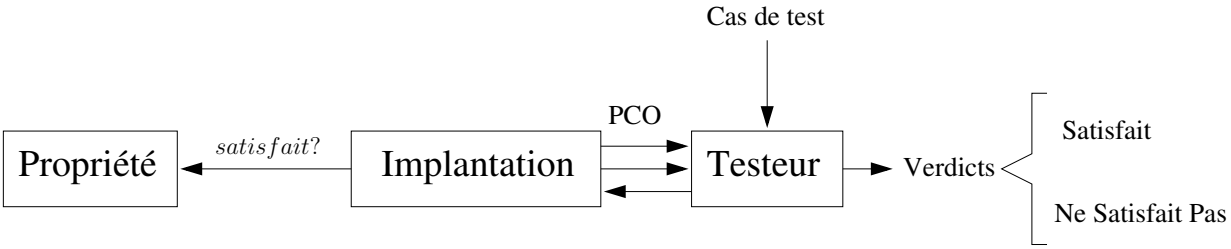

FIG. 5.1 – Test de propriétés

Le domaine particulier d'application visé est le domaine des systèmes de communication. Cependant, les techniques peuvent s'appliquer à des domaines plus larges, comme les systèmes embarqués, ou les systèmes d'information.

Le travail proposé consiste à prolonger le cadre (et en particulier celui décrit par les méthodes de l'outil TGV chapitre 3) de la génération de cas de test de conformité en l'orientant "test de propriétés" [FMP04]. Plus précisément, notre idée générale est de permettre une génération automatique de cas de test à partir d'une spécification "partielle" (la spécification n'exprime pas nécessairement tous les comportements du système testé), et d'une propriété particulière donnée. Une exécution d'un tel cas de test donne lieu à deux verdicts : "Satisfait" ou "Ne Satisfait Pas". Les verdicts émis après l'exécution d'un cas de test indiquent si les implantations testées "satisfont" ou "ne satisfont pas" la propriété donnée.

La production de cas de test à partir de spécifications formelles pour vérifier la satisfiabilité d'une propriété donnée est une idée largement pratiquée. De nombreux travaux, ont déjà traité ce sujet et ont donné lieu à des outils. Les méthodes dans ces outils diffèrent la plupart du temps

sur la modélisation des spécifications et des propriétés considérées. Les méthodes, pour choisir les cas de test, sont souvent basées sur des techniques probabilistes (telles celles proposées dans [RWNH98, PV01, MA00, TF98]). Cependant, un aspect original de notre approche est l'utilisation des "automates paramétrés sur les mots infinis". Nous utilisons ces modèles d'automates pour décrire les propriétés et instancier les cas de test à exécuter. Les cas de test produits seront ensuite exécutés sur des implantations (non déterministes).

L'architecture de la figure 5.2 présente la méthode pour générer nos cas de test de propriétés.

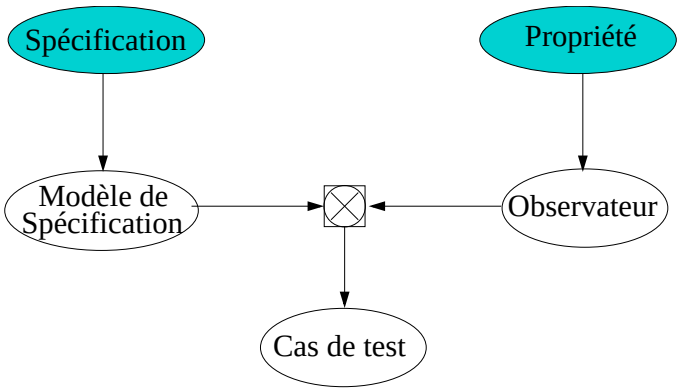

FIG. 5.2 – Principe de la génération automatique de cas de test pour le test de propriétés.

Les propriétés $\mathcal{P}$ données seront modélisées sous la forme d'un observateur $\mathcal{O}$bs. Cet observateur identifiera plus précisément les séquences de $\neg\mathcal{P}$. Le choix de représenter l'observateur $\mathcal{O}$bs comme la négation de la propriété est lié à la relation de satisfiabilité. Ce que nous cherchons à détecter, ce sont des actions ou des comportements interdits au niveau de l'implantation. Dans ce cas le fait d'écrire la négation de la propriété permet d'obtenir toutes les actions indésirables directement dans le cas de test à exécuter par le testeur. Si l'implantation testée satisfait le cas de test alors la propriété n'est pas préservée. Un cas de test de propriétés sera un test élémentaire, produit automatiquement à partir d'une spécification $\mathcal{S}$ et d'un observateur $\mathcal{O}$bs. Un cas de test sera donc généré pour vérifier la "non satisfiabilité" d'une implantation IUT à tester vis-à-vis d'une propriété $\mathcal{P}$ donnée. La production des cas de test de propriétés sera basée sur la méthode de génération proposée dans le cadre du test de conformité. Plus précisément, cette technique de génération de cas de test de propriétés est basée sur :

– Une spécification (partielle) $\mathcal{S}$: elle est utilisée pour la synthèse des cas de test. Si la spécification est partielle, elle doit être supposée "assez proche" des comportements réels de l'implantation à tester. Globalement, les comportements de la spécification ne sont pas trop éloignés de l'implantation en terme d'actions et de représentation. Cette hypothèse est due en partie au fait que la spécification sert de "guide" pour construire les cas de test. Nous notons cependant à ce niveau, l'absence de besoin d'une relation particulière de conformité entre $\mathcal{S}$ et l'IUT, c'est donc pour cela que nous pouvons utiliser une spécification partielle.

– Une propriété $\mathcal{P}$ est représentée sous la forme d'un observateur $\mathcal{O}$bs qui identifiera les séquences de $\neg\mathcal{P}$.

– Les cas de test de propriétés sélectionnés à partir d'un graphe de test ont pour objectif de trouver la plupart des séquences d'exécution "capables" de montrer la relation de "non satisfiabilité" d'une implantation IUT vis-à-vis de $\mathcal{P}$.

Nous allons définir les ingrédients suivants :
- Les modèles de spécification $\mathcal{S}$, d'implantation IUT, et d'observateur $\mathcal{O}$bs.
- Les classes (types) de propriétés $\mathcal{P}$: de sûreté et de vivacité bornée.
- La relation de satisfiabilité définie comme une relation mathématique : $\texttt{imp} \subseteq \text{IUT} \times \mathcal{O}\text{bs}$
- Les cas de test de propriétés CT.
- La génération de cas de test de propriétés à partir de $\mathcal{O}\text{bs} \times \mathcal{S}$.
- La sémantique de l'exécution entre un cas de test CT et une implantation IUT qui délivre un verdict. Ce verdict permet de rejeter ou non une implantation.
- Nous souhaitons qu'un cas de test produit avec une propriété donnée ne rejette que les implantations qui ne vérifient pas cette propriété.

5.2 Les modèles

5.2.1 Modèle de spécification

La spécification considérée ne représente pas exactement tous les comportements attendus par une implantation. Nous utilisons une telle spécification, car elle ne servira que de "guide" pour la construction des cas de test de propriétés. Rappelons qu'une spécification $\mathcal{S}$ est donnée dans un langage de haut niveau comme LOTOS, SDL, ..., sa sémantique est définie par un IOLTS $\mathcal{S} = (Q_{\mathcal{S}},\ A_{\mathcal{S}},\ T_{\mathcal{S}},\ q_{\mathcal{S}}^{init})$ (comme celle proposée pour la spécification du test de conformité).

5.2.2 Modèle d'implantation

Les comportements de l'implantation sont connus à travers un ensemble restreint d'interfaces (cet ensemble est constitué d'actions d'entrées et de sorties de l'implantation). D'un point de vue théorique, les comportements peuvent être considérés comme des IOLTS, IUT = $(Q_{\text{IUT}},\ A_{\text{IUT}},\ T_{\text{IUT}},\ q_{\text{IUT}}^{init})$, où : $A_{IUT}^{I} \cup A_{IUT}^{O}$ est l'interface de l'implantation. Une exécution d'un cas de test sur une implantation IUT sera de type boîte noire. Nous supposons qu'une implantation est complète en entrée A_{IUT}^{I} (l'implantation ne refusera jamais une action d'entrée).

Définition 5.2.1 (Modèle d'implantation : IOLTS) *Une implantation est un IOLTS IUT = $(Q_{IUT},\ A_{IUT},\ T_{IUT},\ q_{IUT}^{init})$ où les actions d'entrée (resp. les sorties) sont les actions de sortie (resp. les entrées) de l'environnement : $A_{IUT} = A_{IUT}^{O} \cup A_{IUT}^{I} \cup \{\tau\}$.*

NOTE :
Nous supposons que $A_S^I \subseteq A_{IUT}^I$ et $A_S^0 \subseteq A_{IUT}^0$. Cette proposition réside dans le fait que l'alphabet des implantations est inconnu.

5.3 Les propriétés

L'objectif de notre travail est donc de générer des cas de test de propriétés. Les propriétés utilisées sont décrites sous la forme d'un observateur $\mathcal{O}$bs (automate) décrivant la propriété $\neg\mathcal{P}$.

Parmi les différentes "classes" de propriétés considérées pour spécifier des comportements, nous distinguons généralement les propriétés de sûreté : "*safety*" et les propriétés de vivacité : "*liveness*".

– Une propriété de sûreté exprime que toutes les actions effectuées par les systèmes sont des actions correctes, ou de façon équivalente, les systèmes ne présentent jamais un comportement non prévu par leurs spécifications. C'est à dire, elle exprime que "quelque chose" de mauvais ne se produit jamais au cours de l'exécution du système. Des exemples classiques sont l'exclusion mutuelle, l'absence de blocage, ...

– Une propriété de vivacité exprime que toutes les actions correctes seront inévitablement effectuées par le système. Par exemple, la terminaison d'un programme, l'absence de famine (tout processus progresse infiniment souvent), ou la garantie de service.

D'un point de vue pratique, nous ne pouvons pas tester les propriétés de vivacité. Le problème est l'arrêt des exécutions des cas de test décrivant ces propriétés. Nous proposons d'utiliser alors, la "classe" de propriétés de "vivacité bornée" (bounded liveness).

Pour modéliser nos deux "classes" de propriétés, les propriétés de sûreté et les propriétés de "vivacité bornée" nous utilisons les automates d'états finis représentant des mots finis ou des mots infinis.

Un observateur modélisant les propriétés de "sûreté" et de "vivacité bornée" : $\mathcal{O}$bs

Une propriété va être directement modélisée par un automate d'états finis (un observateur). Plusieurs automates (Büchi, Muller, Streett, Rabin, etc..) peuvent identifier des séquences infinies (et finies). Nous considérons que notre observateur est un automate déterministe identifiant les langages ω-régulier. Généralement, les automates de Büchi sont utilisés pour ce genre de modélisation [CVWY92]. Mais, nous utiliserons les automates de Rabin pour la description de nos observateurs car il existe des langages ω-régulier reconnaissables par un automate de Büchi non déterministe, mais, non reconnaissables par un automate de Büchi déterministe (cf exemple 5.3).

Nous allons voir à l'aide de l'exemple 5.3, pourquoi nous avons choisi de modèliser nos observateurs avec des automates de Rabin. Pour effectuer cet exemple 5.3, nous introduisons les deux

modèles d'automates suivants (avec les définitions 5.3.1 et 5.3.2) :

Définition 5.3.1 (Automate de Büchi) *Un automate de Büchi A_b est un couple (B,$\mathcal{G}_B$) où :*

- *B = $(Q_B,\ A_B,\ T_B,\ q_B^{init})$ est un IOLTS*
- *$\mathcal{G}_B$ est un sous-ensemble de Q_B représentant l'ensemble des états d'acceptation (ou états répétés).*

L'automate A_b accepte une trace $\alpha \in A_B^\omega$ si et seulement si il existe une séquence d'exécution σ de B telle que α = Trace(σ) et si ρ = Chemin(σ) alors inf(ρ) $\cap\ \mathcal{G}_B \neq \emptyset$.

Le langage $\mathcal{L}(A_b)$ reconnu par A_b est l'ensemble des traces de A_B^ω acceptées par A_b.

Définition 5.3.2 (Automate de Rabin) *Un automate de Rabin A_r est un couple $(R, \mathcal{T})$ où :*

- *R = $(Q_R,\ A_R,\ T_R,\ q_R^{init})$ est un IOLTS,*
- *$\mathcal{T} = \{(L_1^R, U_1^R), \{(L_2^R, U_2^R), \ldots,\ (L_n^R, U_n^R)\}$ est un ensemble de paires d'acceptation avec L_i^R et $U_i^R \subseteq Q_R$ pour i $\in$ {1,2,...,n}.*

L'automate A_r accepte une trace $\alpha \in A_R^\omega$ si et seulement si il existe une séquence d'exécution σ de R et $\exists i \in \{1, 2, \ldots, n\}$ telle que α = Trace(σ) et si ρ = Chemin(σ) alors **inf**$(\rho) \cap L_i \neq \emptyset$ *et* **inf**$(\rho) \cap U_i = \emptyset$.

Le langage $\mathcal{L}(A_r)$ reconnu par A_r est l'ensemble des traces de A_R^ω accepté par A_r.

Nous précisons que les automates déterministes de Rabin [Rab69, Rab72] reconnaissent toute la classe des langages ω-réguliers.

Exemple 8 — **[Représentation d'une propriété sous forme d'un automate déterministe]**
Nous illustrons le fait qu'il existe des langages ω-régulier non reconnaissables par un automate de Büchi déterministe. Pour l'exemple nous considérons la propriété $\mathcal{P}$ suivante : "un système peut toujours revenir dans une situation initiale par une action I après une situation particulière amenée par une action E". Cette propriété peut s'exprimer par le langage (ω-régulier) suivant : L = $(E^*I)^\omega$. La négation de la propriété représentant l'observateur $\mathcal{O}$bs est exprimée par le langage : $\overline{L}$ = $(E + I)^* E^\omega$. Mais, le langage $\overline{L}$ n'est pas reconnu par un automate déterministe de Büchi.
L'automate de Büchi non déterministe reconnaissant $\overline{L}$ est donné par la figure 5.3, avec $\mathcal{G}_B = \{2\}$, en prenant l'état 2 comme l'état répété.
Soit l'automate déterministe de la figure 5.4 considéré comme un automate de Büchi, avec $\mathcal{G}_B = \{2\}$, cet automate accepte toutes les séquences produisant infiniment souvent l'action E ou l'action I. Or, ces séquences ne sont pas prévues dans le langage de $\overline{L}$.
Maintenant, si l'automate de la figure 5.4 est considéré comme un automate de Rabin, avec $\{\{2\},\{1\}\}$ (représentant le couple {U,L} associé aux états d'un automate de Rabin), alors cet automate reconnaît exactement le langage $\overline{L}$.

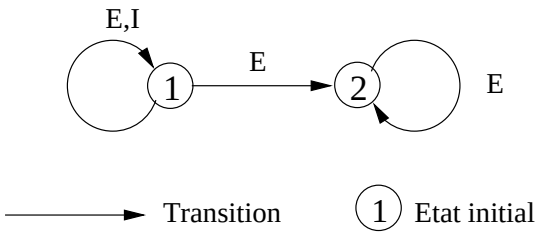

FIG. 5.3 – Automate de Büchi non déterministe reconnaissant (E + I)*E $^{\omega}$

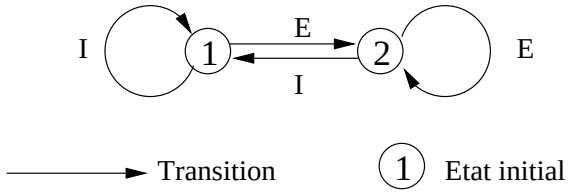

FIG. 5.4 – Automate de Rabin déterministe reconnaissant (E + I)*E $^{\omega}$

Clairement, les automates de Rabin sont un modèle naturel pour exprimer les propriétés de vivacité. Cependant, dans le cadre du test de propriétés, nous ne considérons que des séquences d'exécution finies. Les séquences infinies doivent être approchées par des séquences finies : l'évolution du système sera déterminée par des paramètres (des limites), et n'importe quelle séquence d'exécution dépassant les limites fixées sera considérée comme une séquence infinie. La solution proposée consiste à associer des paramètres (c_{l_i}, c_{u_i}) à chacune des paires $(L_i, U_i) \subseteq \mathcal{T}$. Une trace α est acceptée si et seulement il existe un couple (L_i, U_i) et une séquence d'exécution σ de trace α visitant un nombre suffisant de fois L_i (au moins c_{l_i} fois) et pas trop souvent U_i (au plus c_{u_i} fois).

Avant de donner la définition des automates de Rabin paramétrés, nous redéfinissons la condition **inf**$(\rho) \cap X$, pour un ensemble d'états $X \subseteq Q$.

Définition 5.3.3 *Nous définissons un prédicat $\mathcal{H}(\rho, X)$ pour un chemin ρ et $X \subseteq Q$ de la manière suivante :*

$$\mathcal{H}(\rho, X) = \forall j \exists i \geq j . \rho(i) \in X$$

Intuitivement, le prédicat $\mathcal{H}(\rho, X)$ signifie que le chemin ρ contient une infinité d'états de X.

Lemme II.1

$$\textbf{inf}(\rho) \cap X \neq \emptyset \Longleftrightarrow \mathcal{H}(\rho, X)$$

Preuve :

$\Longrightarrow$: Soit $q \in$ **inf**$(\rho) \cap X$. La condition $\forall j \exists i \geq j . \rho(i) = q$ se réécrit, puisque $q \in X$, $\forall j \exists i \geq j . \rho(i) \in X$.

$\Longleftarrow$: Supposons que **inf**$(\rho) \cap X = \emptyset$. Cela signifie : $\forall q \in X\ \exists j_q\ \forall i\ \geq\ j_q\ \rho(i) \neq q$. Puisque X est fini, on peut définir $j_0 = max\{j_q \mid q \in X\}$. Il vient $\forall i\ \geq\ j_0\ \rho(i)\ \notin\ X$. D'où, le prédicat $\mathcal{H}(\rho, X)$ est évalué à faux.

La notion d'automate de Rabin paramétré est formalisée par la définition suivante :

Définition 5.3.4 (Automate de Rabin paramétré) *Un automate de Rabin paramétré est un tuple A_r^p = $(R, \mathcal{T}, \mathcal{C})$ avec, $(R, \mathcal{T})$ un automate de Rabin (A_r), $\mathcal{C} = \{(c_{l_1}, c_{u_1}), \cdots, (c_{l_n}, c_{u_n})\}$ un ensemble de couples de paramètres, avec $c_{l_i}, c_{u_i} \in \mathbb{N}$ pour i $\in \{1,2,\cdots,n\}$.*

Nous définissons :

- *Une approximation de $\mathcal{H}(\rho, X)$ de la manière suivante : $\mathcal{H}(\rho, X, k) =\mid \{i \mid\ \rho(i) \in X\} \mid \geq k$. $\mathcal{H}(\rho, X, k)$ est vrai si et seulement si ρ contient plus de k états de X.*

- *Une trace σ est acceptée par A_r^p si et seulement si il existe $i \in \{1, 2, \cdots, n\}$ et un chemin ρ de trace σ tel que : $\mathcal{H}(\rho, L_i^R, c_{l_i}) \wedge \neg\mathcal{H}(\rho, U_i^R, c_{U_i})$.*

- *Le langage reconnu par l'automate de Rabin paramétré A_r^p est l'ensemble des traces acceptées par A_r^p.*

5.4 Relation de satisfiabilité

La génération de cas de test est facilitée si nous considérons que l'automate de Rabin paramétré de l'observateur $\mathcal{O}$bs identifie exactement les traces de $\neg\mathcal{P}$. La relation de satisfiabilité se définit formellement entre une implantation IUT et une propriété $\mathcal{P}$ par :

Définition 5.4.1 (Relation de satisfiabilité) *Soit IUT = $(Q_{IUT},\ A_{IUT},\ T_{IUT},\ q_{IUT}^{init})$ un IOLTS représentant une implantation, $\mathcal{P}$ est une propriété à vérifier sur IUT, et $\mathcal{O}$bs = $(\mathcal{O}, \mathcal{T}_\mathcal{O}, \mathcal{C}_\mathcal{O})$ l'observateur de $\neg\mathcal{P}$ représenté par un automate de Rabin paramétré tel que $\mathcal{O}$ = $(Q_\mathcal{O},\ A_\mathcal{O},\ T_\mathcal{O},\ q_\mathcal{O}^{init})$ soit un IOLTS* déterministe, complet *et $\mathcal{L}(\mathcal{O}bs)$ le langage reconnu par $\mathcal{O}$bs, avec $A_\mathcal{O} \subseteq A_{IUT} \cup \{\delta\}$. Nous disons que IUT* satisfait *$\mathcal{P}$ si et seulement si :*

$$\mathcal{L}(\Delta(IUT))\downarrow\ \mathcal{A}_\mathcal{O} \cap \mathcal{L}(\mathcal{O}bs) = \emptyset.$$

En d'autres termes, si IUT satisfait *$\mathcal{P}$, alors aucune trace de l'IUT n'est reconnue par l'observateur $\mathcal{O}$bs.*

5.5 Architecture de test et cas de test de propriétés

Architecture de test

L'architecture de test, pour un cas de test produit entre $\mathcal{S}$ et $\mathcal{O}$bs, doit être constituée de façon à contrôler et observer les comportements de l'implantation (en contrôlant les actions d'entrée et en observant les actions de sortie). Nous considérons une architecture de test simplifiée et réduite à une paire d'ensembles (A_c, A_u), où A_c est l'ensemble des actions contrôlables, et A_u l'ensemble des actions observables. Une architecture de test est *compatible* avec l'observateur $\mathcal{O}$bs si et seulement si les contraintes suivantes sont respectées : $A^I_{\mathcal{O}} = A_c$ et $A^O_{\mathcal{O}} = A_u$ et $\delta \in A_u$. Notons que cette notion diffère légèrement de celle de la compatibilité entre deux LTS.

En d'autres termes, le testeur a besoin de contrôler (resp. d'observer) toutes les actions d'entrée (resp. les actions de sortie) appartenant à l'observateur.

Cas de test de propriétés

Intuitivement, un cas de test pour une propriété $\mathcal{P}$ donnée, est un ensemble de traces reconnues par un observateur de $\neg\mathcal{P}$. Cet ensemble de traces sera décrit par un automate de Rabin paramétré ACT^p_r).

Pour faciliter les exécutions du cas de test, nous avons besoin d'ajouter la condition de contrôlabilité : l'exécution de ACT^p_r est déterministe (au plus une action contrôlable est permise à chaque état), et non bloquante (aucune action observable ne peut être refusée).

Définition 5.5.1 (Cas de test de propriétés) *Soit $\mathcal{O}bs$ = $(\mathcal{O}, \mathcal{T}_{\mathcal{O}}, \mathcal{C}_{\mathcal{O}})$ un observateur, $A_{\mathcal{O}}$ l'ensemble des actions et $\mathcal{AT}$ = (A_c, A_u) l'architecture de test compatible avec l'observateur. ACT^p_r = (CT,$\mathcal{T}_{CT}, \mathcal{C}_{CT}$) est un cas de test de propriétés (automate de Rabin paramétré) pour $\mathcal{O}bs$ avec CT = $(Q_{CT},\ A_{CT},\ T_{CT},\ q^{init}_{CT})$ si et seulement si il satisfait les règles suivantes :*

1. $A_{CT} = A^I_{CT} \cup A^O_{CT}$ *avec* $A^O_{CT} \subseteq A_u$ *et* $A^I_{CT} \subseteq A_c$.
2. *CT est déterministe et contrôlable.*
3. $\mathcal{L}(ACT^p_r) \subseteq \mathcal{L}(\mathcal{O}bs)$

5.6 Graphe de test

La solution proposée pour produire des cas de test de propriétés consiste à générer tout d'abord un graphe de test à partir de la spécification partielle et de l'observateur. Ce graphe de test sera un automate de Rabin paramétré reconnaissant un sous-ensemble du langage de l'observateur. Chaque sous-graphe contrôlable peut alors être transformé en un cas de test exécutable pour la propriété.

Cependant, même pour une propriété simple et une architecture de test restreinte, il s'avère que le nombre de séquences produites est relativement grand. Nous n'allons pas construire un graphe de test "complet". Le fait de calculer un trop grand graphe de test pourrait limiter l'intérêt pratique. En effet, un certain nombre de séquences contenues dans le graphe complet sont susceptibles de ne pas être dans le comportement réel de l'IUT, et l'exécution de ces séquences ne serait pas nécessaire (pour tester la propriété donnée). Réciproquement, la probabilité de choisir un cas de test "pertinent", dans un grand ensemble serait plutôt faible. Par conséquent nous avons besoin de calculer

un graphe de test ”réduit”. L'heuristique proposée ici consiste à exploiter au mieux l'information fournie par la spécification, censée être assez proche du comportement de l'implantation réelle. De façon plus précise, le graphe de test calculé est obtenu comme suit :

1. Nous calculons tout d'abord l'ensemble des séquences d'exécution de $\mathcal{L}(S)$ dont les traces associées sont des *préfixes* de traces de $\mathcal{L}(\mathcal{O}bs)$. Ces séquences sont des “candidates naturelles” pour observer la non-satisfiabilité de $\mathcal{P}$ puisqu'elles appartiennent à la spécification et sont donc supposées exécutables sur l'implantation (permettant ainsi d'amener cette dernière dans un état “proche” d'un état d'erreur).
2. Pour obtenir un ensemble de cas de test, il ne reste alors qu'à prolonger chacune de ces séquences par le suffixe nécessaire pour que la trace associée appartiennent à $\mathcal{L}(\mathcal{O}\text{bs})$. Si les séquences ainsi obtenues s'avèrent exécutables sur l'implantation celle-ci sera clairement incorrecte vis-à-vis de l'observateur (elle ne satisfera pas la propriété attendue). Toutefois, pour limiter le nombre de cas de test produits, seules certaines séquences de la spécification seront prolongées : à une séquence de trace α de $\mathcal{L}(S)$, les suffixes ajoutés seront de la forme $a.\beta$ où a est soit une action de sortie ($a \in A_u$), soit une action d'entrée ($a \in A_c$) à condition que a ne soit plus exécutable ultérieurement dans la spécification ($\nexists \alpha' \,.\, \alpha.\alpha'.a \in \mathcal{L}(S)$). Intuitivement cette condition indique qu'une sortie de l'observateur ne sera jamais refusée par le cas de test, alors qu'une entrée de l'observateur sera exécutée dans un état donné du cas de test soit si la spécification l'autorise dans cet état précis, soit si la spécification ne l'autorisera plus ultérieurement.

D'un point de vue formel, le graphe de test AGT_r^p calculé est un automate de Rabin paramétré $(GT, \mathcal{T}_{GT}, \mathcal{C}_{GT})$: L'IOLTS GT contient les ensembles des séquences, $\mathcal{T}_{GT}$ l'ensemble de couples, et $\mathcal{C}_{GT}$ l'ensemble des compteurs sont hérités de $\mathcal{T}_0$ et de $\mathcal{C}_0$ (définition 5.6.1). AGT_r^p est obtenu de la manière suivante :

1. Transformation de l'automate de la spécification initiale $\mathcal{S}_0$ en un automate déterministe de suspension $\mathcal{S}$ en respectant l'architecture de test $\mathcal{AT}$ = (A_c, A_u) : S = det($\Delta(S_0), A_c \cup A_u$). Cette opération préserve le langage : $\mathcal{L}(\mathcal{S})$ = $\mathcal{L}(\mathcal{S}_0) \downarrow (A_c \cup A_u)$.
2. Calcul du produit $\otimes$ synchrone entre la spécification $\mathcal{S}$ et l'observateur $\mathcal{O}$bs : L'objectif de ce produit est de marquer chaque état $p_{\mathcal{S}}$ de $\mathcal{S}$ avec les états $p_{\mathcal{O}}$ correspondant à $\mathcal{O}$ (application de la règle R1).
3. Extension du produit obtenu en complétant les séquences d'exécution menant à chaque état $(p_S, p_{\mathcal{O}})$ par des suffixes issus de $\mathcal{O}$ et de la forme $\{a.\beta \mid a \in A_u\}$ (règle R4) ou $\{a.\beta \mid a \in A_c \wedge \nexists \alpha \,.\, p_S \stackrel{\alpha.a}{\Longrightarrow}\}$ (règle R3).

NOTE :
Par hypothèse, l'action $!\delta$ appartient toujours à A_u. Dans les représentations graphiques suivantes, nous choisissons toutefois de ne pas représenter l'action $!\delta$ considérant que cette action sera présente dans chaque état du graphe de test et des cas de test produits.

L'architecture de test de l'exemple de la figure 5.5 est : A_u = $\{ !a, !b, !\delta\}$, A_c = $\{ ?c\}$ Le graphe produit entre la spécification et l'observateur contient les séquences de test les plus pertinentes selon les objectifs du test. En particulier, l'action $?c$ présente dans la spécification ne sera rajoutée au graphe de test qu'après l'exécution de $!b$.

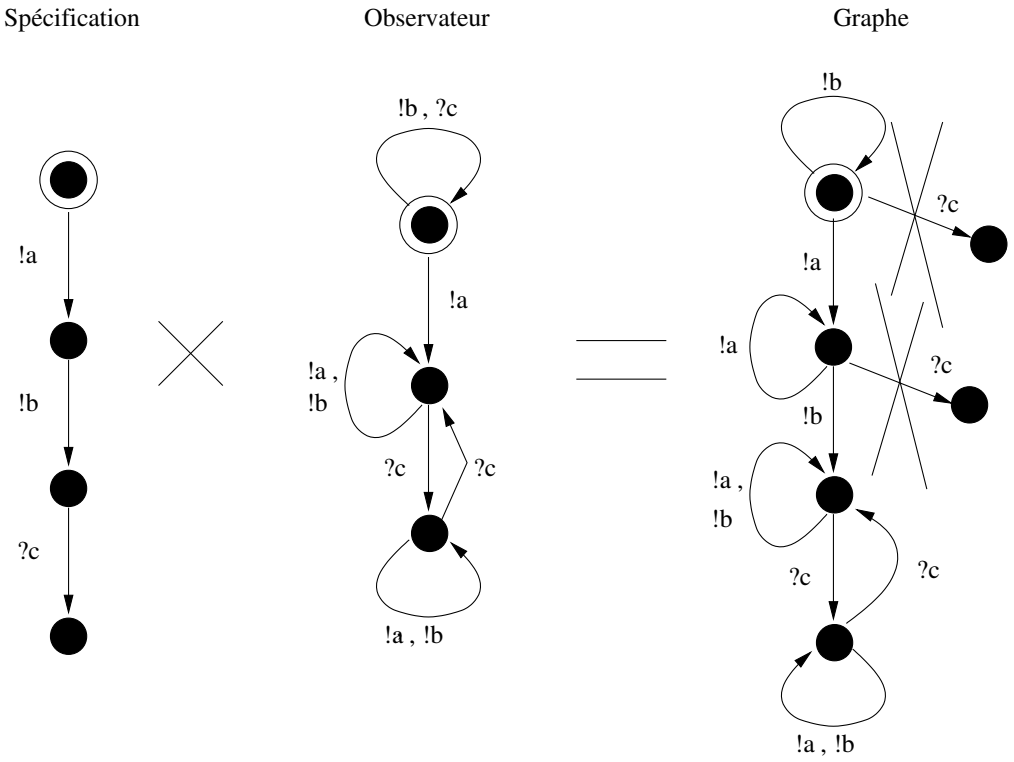

FIG. 5.5 – Graphe de test obtenu à partir d'une spécification et d'un observateur.

Plus formellement, le graphe de test est le résultat (donné par la définition 5.6.1) d'un produit asymétrique $\otimes$ entre $\mathcal{S}$ et $\mathcal{O}$.

Définition 5.6.1 (Graphe de test de propriétés) *Soit $\mathcal{O}bs = (\mathcal{O}, \mathcal{T}_\mathcal{O}, \mathcal{C}_\mathcal{O})$ un observateur avec $\mathcal{O}$ = $(Q_\mathcal{O},\ A_\mathcal{O},\ T_\mathcal{O},\ q_\mathcal{O}^{init})$, $\mathcal{T}_\mathcal{O} = \langle (L_1^\mathcal{O}, U_1^\mathcal{O}), (L_2^\mathcal{O}, U_2^\mathcal{O}), \ldots, (L_n^\mathcal{O}, U_n^\mathcal{O})\rangle$, et $\mathcal{C}_\mathcal{O}$ = $(c_{l_i^\mathcal{O}},\ c_{u_i^\mathcal{O}})$. Soit $\mathcal{AT}$ = $(A_c,\ A_u)$ une architecture de test compatible avec l'observateur, $\mathcal{S}_0$ une spécification et $\mathcal{S}$ = det($\Delta(\mathcal{S}_O,\ A_c \cup A_u)$) l'automate déterministe de suspension associé à $\mathcal{S}_0$.*

Nous définissons l'automate de Rabin paramétré $AGT_r^p = (GT, \mathcal{T}_{GT}, \mathcal{C}_{GT})$ où GT = $(Q_{GT},\ A_{GT},\ T_{GT},\ q_{GT}^{init})$ avec $Q_{GT} \subseteq Q_\mathcal{S} \times Q_\mathcal{O}$, $A_{GT} \subseteq A_{\mathcal{O}bs}$, $q_{GT}^{init} = (q_\mathcal{S}^{init}, q_\mathcal{O}^{init})$, et Q_{GT}, T_{GT} sont obtenues de la façon suivante :

1. *Soit $Q_\otimes$ et $T_\otimes$ les plus petits ensembles satisfaisant les règles R0 et R1 suivantes :*

$$q_{GT}^{init} \in Q_{\otimes} \quad \text{[R0]}$$

$$\frac{(p_S, p_{\mathcal{O}}) \in Q_{\otimes},\ p_S \xrightarrow{a}_{T_S} q_S,\ p_{\mathcal{O}} \xrightarrow{a}_{T_{\mathcal{O}}} q_{\mathcal{O}}}{(q_S, q_{\mathcal{O}}) \in Q_{\otimes},\ (p_S, p_{\mathcal{O}}) \xrightarrow{a}_{T_{\otimes}} (q_S, q_{\mathcal{O}})} \quad [R1]$$

2. *Alors, Q_{GT} et T_{GT} sont les plus petits ensembles vérifiant les règles R2, R3 et R4 suivantes :*

$$Q_{\otimes} \subseteq Q_{GT},\ T_{\otimes} \subseteq T_{GT} \quad \text{[R2]}$$

$$\left.\begin{array}{c} (p_S, p_{\mathcal{O}}) \in Q_{GT}, \\ p_{\mathcal{O}} \xrightarrow{a}_{T_{\mathcal{O}}} q_{\mathcal{O}}, \quad a \in A_c \\ \nexists\, \alpha \in (A_u \cup A_c)^*.(\alpha.a \in \mathit{Trace}(p_S)) \\ \hline (p_S, q_{\mathcal{O}}) \in Q_{GT},\ (p_S, p_{\mathcal{O}}) \xrightarrow{a}_{T_{GT}} (p_S, q_{\mathcal{O}}) \end{array}\right\} \quad \text{[R3]}$$

$$\left.\begin{array}{c} (p_S, p_{\mathcal{O}}) \in Q_{GT}, \\ p_{\mathcal{O}} \xrightarrow{a}_{T_{\mathcal{O}}} q_{\mathcal{O}}, \quad a \in A_u \\ p_S \not\xrightarrow{a} \\ \hline (p_S, q_{\mathcal{O}}) \in Q_{GT},\ (p_S, p_{\mathcal{O}}) \xrightarrow{a}_{T_{GT}} (p_S, q_{\mathcal{O}}) \end{array}\right\} \quad \text{[R4]}$$

3. *L'ensemble des couples $\mathcal{T}_{GT}$ est égal à $\langle (L_1^{GT}, U_1^{GT}), (L_2^{GT}, U_2^{GT}), \ldots, (L_n^{GT}, U_n^{GT}) \rangle$ où L_i^{GT} et U_i^{GT} sont définis comme suit :*

$$\begin{aligned} L_i^{GT} &= \{(p_S, p_{\mathcal{O}}) \in Q_{GT} \mid p_{\mathcal{O}} \in L_i^{\mathcal{O}}\} \\ U_i^{GT} &= \{(p_S, p_{\mathcal{O}}) \in Q_{GT} \mid p_{\mathcal{O}} \in U_i^{\mathcal{O}}\} \end{aligned}$$

4. *L'ensemble des compteurs $\mathcal{C}_{GT}$ est directement hérité de $\mathcal{O}$bs :*

$$\mathcal{C}_{GT} = \mathcal{C}_{\mathcal{O}}$$

Lemme II.2 *Pour tous les graphes de test AGT_p^r construits à partir d'un observateur $\mathcal{O}bs$ nous avons : $\mathcal{L}(AGT_r^p) \subseteq \mathcal{L}(\mathcal{O}bs)$.*

Preuve : *Nous allons montrer :*

Etape 1 $\mathcal{L}(\mathcal{GT}) \subseteq \mathcal{L}(\mathcal{O})$,

Etape 2 *pour tout chemin ρ de GT, de trace $\alpha \in \mathcal{L}(\mathcal{GT})$ vérifiant pour un $i \in \{1, 2, \cdots, n\}$ $\mathcal{H}(\rho, L_i^R, c_{l_i})$ et $\neg\mathcal{H}(\rho, u_i^R, c_{u_i})$, il existe un chemin ρ' de $\mathcal{O}$ de trace $\alpha \in \mathcal{L}(\mathcal{O})$ vérifiant $\mathcal{H}(\rho', L_i^R, c_{l_i})$ et $\neg\mathcal{H}(\rho', u_i^R, c_{u_i})$.*

Etape 1 :

Soit $\alpha \in \mathcal{L}(\mathcal{GT})$ et σ_{GT} la séquence d'exécution de GT de trace α. Nous avons :

$$\sigma_{GT} = (q_0^s, q_0^{Obs}) \xrightarrow{\alpha_0}_{GT} (q_1^s, q_1^{Obs}) \xrightarrow{\alpha_1}_{GT} \cdots \xrightarrow{\alpha_n}_{GT} (q_m^s, q_m^{Obs})$$

Nous pouvons montrer que, par induction sur la longueur des séquences,

$$\sigma_{Obs} = q_0^{Obs} \xrightarrow{\alpha_0}_{Obs} q_1^{Obs} \xrightarrow{\alpha_1}_{Obs} \cdots \xrightarrow{\alpha_n}_{Obs} q_m^{Obs}$$

est une séquence d'exécution de $\mathcal{O}$. En effet, σ_{GT} est obtenue à partir de σ_{Obs} par application des règles R1, R3 et R4. Donc, α =Trace(σ_{Obs}), c'est-à-dire, $\alpha \in \mathcal{L}(\mathcal{O})$.

Etape 2 :

Partant d'une séquence d'exécution σ_{GT}, projetant pour obtenir une séquence σ_{Obs}, et considérant les chemins ρ et ρ' associés, comme à l'étape 1, nous avons qu'un état (q^s, q^{Obs}) est un état distingué de AGT_r^p si et seulement si q^{Obs} est un état distingué de $\mathcal{O}bs$.

5.7 Exemple d'un produit étendu entre une spécification $\mathcal{S}$ et un observateur $\mathcal{O}$bs

L'exemple présenté est l'application des règles précédemment données afin de construire un graphe de test à partir d'une spécification et d'un observateur. Les algorithmes permettant de construire le graphe sont présentés dans la section 7.2.2. L'architecture de test est : A_c = { ?d}, A_u = { !a, !b, !d, !δ}.

Détail de construction selon les règles

(1) : Application des règles R0 et R1 : La première phase du produit est d'obtenir les séquences communes entre la spécification et l'observateur.

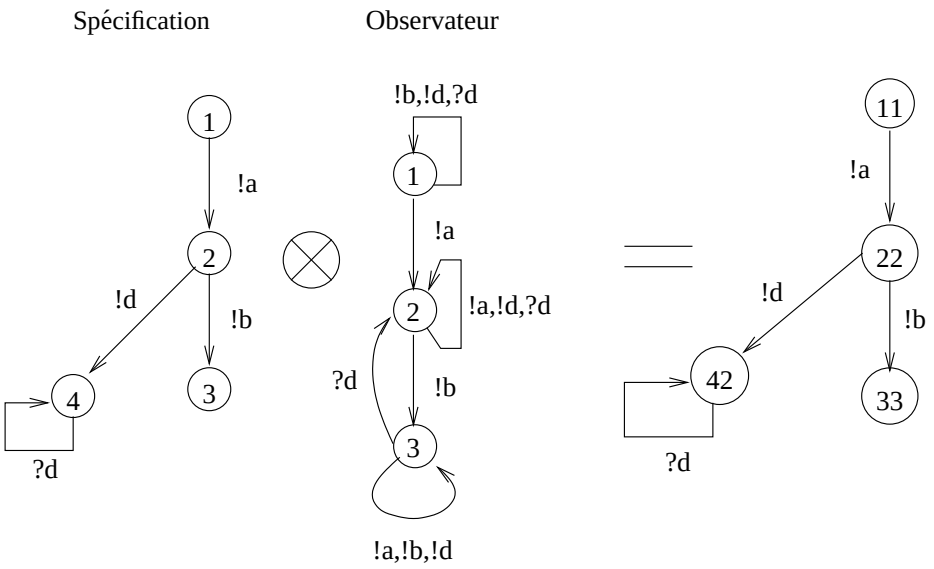

FIG. 5.6 – Produit selon les règles R0 et R1.

(2) : Application des règles R2, R3 et R4 : La deuxième partie du produit permet d'étendre les séquences d'exécution de la spécification.

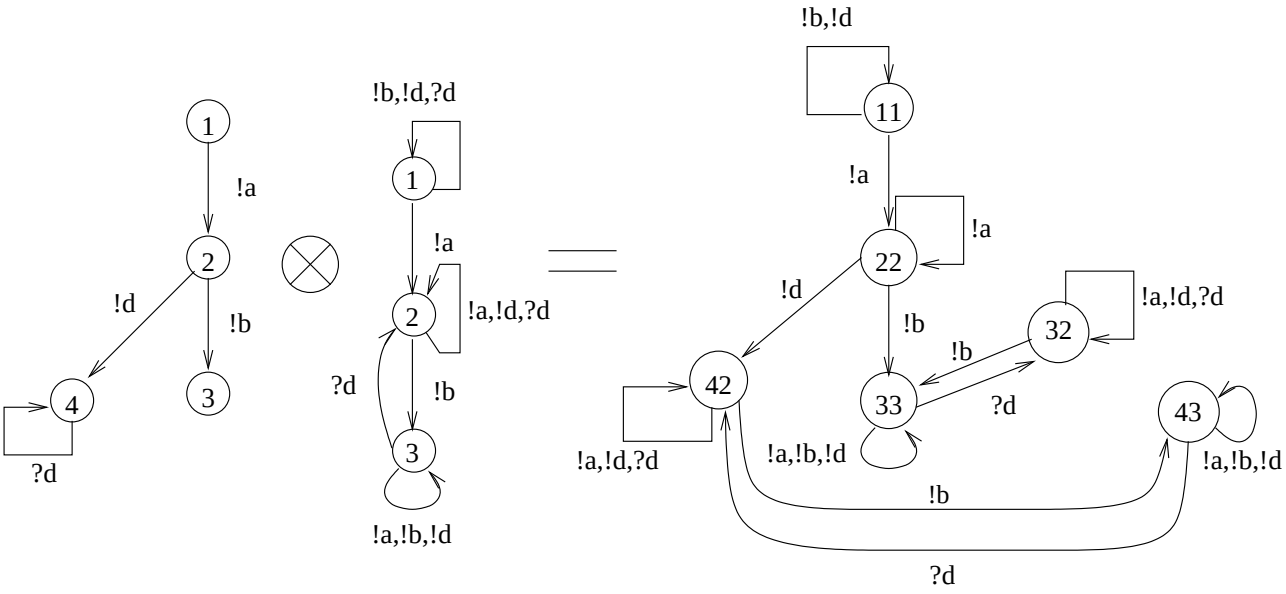

FIG. 5.7 – Produit selon les règles R2, R3 et R4

(3) : Marquage des états U et L. Nous supposons que l'état 1 de l'observateur n'est pas un état distingué, que l'état 2 à le marquage U, et que l'état 3 a le marquage L.

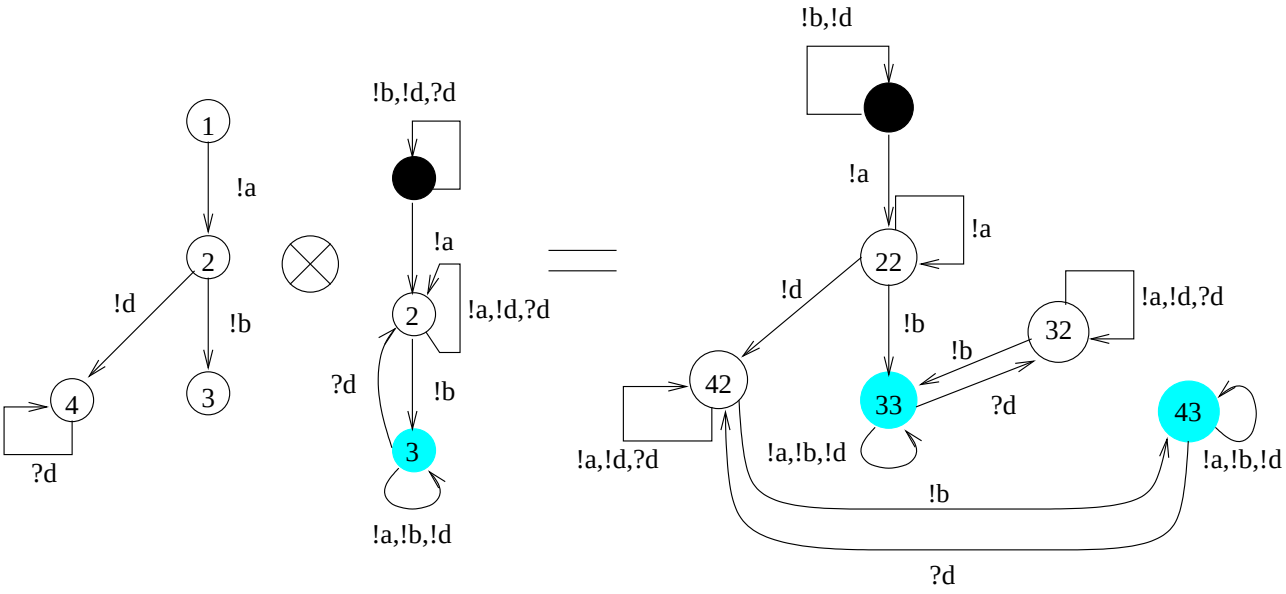

FIG. 5.8 – Marquage des états du produit

5.8 Sélection de cas de test de propriétés

Le but de la sélection d'un cas de test est d'extraire un "sous-graphe contrôlable" du graphe de test contenant au moins une séquence reconnue par l'observateur. À une telle séquence doivent être associés des traces du langage de l'observateur et un chemin atteignant et parcourant une composante fortement connexe contenant un état appartenant à un des ensembles L_i^{GT} (pour un certain i). Réciproquement, si le chemin associé ne mène pas à une composante fortement connexe, la séquence ne peut appartenir à $\mathcal{L}(AGT_r^p)$. Par conséquent, nous définissons d'abord sur le graphe de test GT le prédicat L2L (pour "*leads to L*"), dénotant l'ensemble des états menant à une composante fortement connexe comportant un état de L_i^{GT} :

$$\mathrm{L2L}\,(q) \equiv \exists(q_1, q_2, \omega_1, \omega_2, \omega_3).\ (q \overset{\omega_1}{\Longrightarrow}_{T_{GT}} q_1 \overset{\omega_2}{\Longrightarrow}_{T_{GT}} q_2 \overset{\omega_3}{\Longrightarrow}_{T_{GT}} q_1 \text{ et } \exists i.\ q_2 \in L_i^{\mathrm{GT}})$$

Nous pouvons maintenant définir un sous-ensemble de la relation T_{GT} , contrôlable, et contenant au moins une séquence de $\mathcal{L}(\mathcal{O})$. Ce sous-ensemble contient pour chaque état p du graphe de test T_{GT} des transitions non contrôlables issues de p (marqué par un élément de A_u), et tout au plus une transition contrôlable (aléatoirement choisie) de T_{GT} menant à un état de L2L quand plusieurs transitions existent pour atteindre un tel état. Plus formellement, nous présentons une fonction de

sélection select(p) qui sélectionne un sous-graphe issu de p, en utilisant la fonction *oneof***(V)** qui sélectionne aléatoirement une action de V. Nous noterons :

$$\text{select}(T_{GT}) = \text{select}(q_{\text{GT}}^{init})$$

$$\text{select}(p) = \bigcup_{a \in A_u} \{(p,a,q) \mid (p,a,q) \in T_{\text{GT}}\} \cup \bigcup_{\substack{a \in A_u \\ (p,a,q) \in T_{\text{GT}}}} \text{select}(q) \cup$$

Soit $a = \text{oneof}(Act(p) \cap A_c)$ et q tel que $p \xrightarrow{a}_{T_{\text{GT}}} q$ et L2L (q) dans

$$\{(p,a,q)\} \cup \text{select}(q)$$

Nous assurons ainsi que l'exécution d'un cas de test de propriétés ne sera jamais arrêtée lors d'une réception d'un événement inattendu de l'implantation. La définition d'un cas de test de propriétés est alors la suivante :

Définition 5.8.1 (Sélection d'un cas de test de propriétés) *Soit $AGT_r^p = (GT, \mathcal{T}_{GT}, \mathcal{C}_{GT})$ un automate de Rabin paramétré représentant un graphe de test avec GT = $(Q_{GT},\ A_{GT},\ T_{GT},\ q_{GT}^{init})$ un IOLTS représentant un graphe de test complet et AT = (A_c, A_u) une architecture de test. Un cas de test $ACT_r^p = (CT, \mathcal{T}_{CT}, \mathcal{C}_{CT})$ est un automate de Rabin paramétré avec CT = $(Q_{CT},\ A_{CT},\ T_{CT},\ q_{CT}^{init})$ un IOLTS, tel que : $q_{CT}^{init} = q_{GT}^{init}$, $A_{CT} = A_{GT}$, Q_{CT} est le sous-ensemble de Q_{GT} atteignable par T_{CT} provenant de q_{CT}^{init}, $\mathcal{T}_{CT}$ est la restriction de $\mathcal{T}_{GT}$ et de Q_{CT}, et T_{CT} est défini comme suit :*

$$T_{CT} = \textit{select}\,(T_{GT})$$

5.9 Exécution d'un cas de test et verdicts

Soit IUT = $(Q_{\text{IUT}},\ A_{\text{IUT}},\ T_{\text{IUT}},\ q_{\text{IUT}}^{init})$ une implantation, ACT_r^p = $(\mathcal{CT}, \mathcal{T}_{\text{CT}}, \mathcal{C}_{\text{CT}})$ un cas de test avec $\mathcal{CT}$ = $(Q_{\mathcal{CT}},\ A_{\mathcal{CT}},\ T_{\mathcal{CT}},\ q_{\mathcal{CT}}^{init})$, et $(A_c,\ A_u)$ une architecture de test. Rappelons que $A_{\text{IUT}}^I \subseteq A_c$, $A_{\text{IUT}}^O \subseteq A_u$ et $A_{\text{CT}} \subseteq A_{\mathcal{O}} = A_c \cup A_u$. Nous définissons un verdict pour chaque exécution du cas d'un cas de test. L'exécution du cas de test ACT_r^p sur l'implantation IUT est modélisée par la composition parallèle du cas de test et de l'implantation, avec synchronisation sur le vocabulaire d'action de l'architecture de test $A_c \cup A_u$. Cette exécution peut être décrite par un IOLTS $\mathcal{E}$ = $(Q_{\mathcal{E}},\ A_{\mathcal{E}},\ T_{\mathcal{E}},\ q_{\mathcal{E}}^{init})$, où : $A_{\mathcal{E}}$ = A_{CT}, et $Q_{\mathcal{E}}$ et $T_{\mathcal{E}}$ sont des ensembles définis comme suit :

- Q_ε est un ensemble des *configurations*. Une *configuration* est un triplet $(p_{\text{CT}}, p_{\text{IUT}}, \lambda)$ où : $p_{\text{CT}} \in Q_{\text{CT}}$, $p_{\text{IUT}} \in Q_{\text{IUT}}$ et λ est une fonction à valeurs dans $\mathbb{N}$, définie sur un ensemble de variables : à chaque sous-ensemble L_i^{CT} (resp. U_i^{CT}) est associée une variable x_i (resp. y_i) comptant le nombre de fois qu'un état de L_i^{CT} (resp. de U_i^{CT}) est visité pendant une exécution d'un cas de test.

- T_ε est l'ensemble des transitions $(p_{\text{CT}},\ p_{\text{IUT}},\ \lambda) \xrightarrow{a}_{\varepsilon} (q_{\text{CT}},\ q_{\text{IUT}},\ \lambda')$ tel que :

  – $p_{\text{CT}} \xrightarrow{a}_{\text{CT}} q_{\text{CT}}$, $p_{\text{IUT}} \xrightarrow{a}_{\text{IUT}} q_{\text{IUT}}$ et

  – $$\lambda' = \begin{cases} \lambda & \text{si } q_{\text{CT}} \notin \bigcup_{i \in \{1,\cdots k\}} (L_i^{\text{CT}} \cup U_i^{\text{CT}}) \\ \lambda[\lambda(x_i) + 1/x_i] & \text{si } q_{\text{CT}} \in L_i^{\text{CT}} \\ \lambda[\lambda(y_i) + 1/y_i] & \text{si } q_{\text{CT}} \in U_i^{\text{CT}} \end{cases}$$

    La notation $\lambda[v/x_i]$ (resp. $\lambda[v/y_i]$) désigne la fonction égale à λ sauf au point x_i (resp. y_i) où $\lambda[v/x_i](x_i) = v$ (resp. $\lambda[v/x_i](y_i) = v$).

– La configuration initiale q_{CT}^{init} est $(q_{\text{CT}}^{init}, q_{\text{IUT}}^{init}, \lambda^{init})$, où $\forall i$, $\lambda^{init}(x_i) = \lambda^{init}(y_i) = 0$.

Les VERDICTS : L'exécution d'un cas de test délivre un verdict. Ces verdicts sont formalisés par une fonction **Verdict** sur les séquences d'exécution vers l'ensemble $\{\mathbf{Satisfait}, \mathbf{Ne\ Satisfait\ Pas}\}$. Plus précisément, nous définissons :

– **Verdict**(σ) = **Ne Satisfait Pas** si et seulement si $\rho = Chemin(\sigma)$, il existe $i \in \{1, 2, \ldots, k\}$, $l \in \mathbb{N}$ tel que :

  1. $\rho(l) = (p_l^{\text{CT}}, p_l^{\text{IUT}}, \lambda_l)$, $p_l^{\text{CT}} \in L_i^{\text{CT}}$ et $\lambda_l(x_i) \geq c_{l_i}$, et
  2. pour chaque $m \in [0 \cdots l]$ $\rho(m) = (q_m^{\text{CT}}, q_m^{\text{IUT}}, \lambda_m)$ et $q_m^{\sigma} \in U_i^{\text{CT}}$ implique $\lambda_m(y_i) \leq c_{u_i}$.

– **Verdict**(σ) = **Satisfait** si et seulement si $\rho = Chemin(\sigma)$, pour tout $i \in \{1, 2, \ldots, k\}$ avec $l \in \mathbb{N}$ et ρ le chemin associé à σ :

  1. Soit, $\rho(l) = (p_l^{\text{CT}}, p_l^{\text{IUT}}, \lambda_l)$ et $p_l^{\text{TC}} \in L_i^{\text{CT}}$ implique $\lambda_l(x_i) < c_{l_i}$.
  2. Soit, il y a un $m \in [0 \cdots l]$ $\rho(m) = (q_m^{\text{CT}}, q_m^{\text{IUT}}, \lambda_m)$, $q_m^{\text{CT}} \in U_i^{\text{CT}} et \lambda_m(y_i) > c_{u_i}$.

Dans la pratique, les exécutions de cas de test de propriétés peuvent être améliorées de la façon suivante :

– À chaque étape de l'exécution les conditions de contrôlabilité peuvent donner un choix entre une action observable et une action contrôlable. Dans cette situation le testeur peut soit attendre pour

observer l'action observable produite (cela sera possible en utilisant une temporisation locale), ou alors choisir d'exécuter l'action contrôlable.

– Les paramètres formels de l'ensemble $\mathcal{C}_{\text{CT}}$ sont instanciés en fonction de l'environnement de test. Le verdict **Ne Satisfait Pas** est donné dès qu'une séquence d'exécution incorrecte est trouvée.

NOTE :
Pendant la phase d'exécution, une temporisation est armée au début de chaque exécution (donnée par l'outil d'exécution). Cette temporisation permet d'arrêter l'exécution d'un cas de test dans le cas où une séquence d'exécution rentre dans une boucle ne possédant ni d'états de L_i^{CT} ni d'états de U_i^{CT}. L'exécution s'arrête alors dès que la temporisation expire.

Nous allons montrer maintenant la correction de la méthode.

Définition 5.9.1 (Correction) *Un cas de test est non biaisé si lorsqu'une implantation est rejetée par le cas de test alors la propriété n'est pas satisfaite. Soit IUT l'implantation, ACT_r^p le cas de test et $\mathcal{E}$ l'IOLTS produit entre le cas de test et l'implantation. Un cas de test est non biaisé si et seulement si étant donnée σ une séquence d'exécution du produit et $\alpha = Trace(\sigma)$.*

$$\textbf{Verdict}(\sigma) = \textbf{Ne Satisfait Pas} \Longrightarrow \alpha \in \mathcal{L}(\mathcal{O}bs)$$

Une suite de test ST produite à partir de notre graphe de test (section 5.6) est correcte, si elle ne contient que des tests non-biasés.

Proposition 5.9.1 *Soit IUT l'implantation, ACT_r^p le cas de test et $\mathcal{E}$ l'IOLTS produit entre le cas de test et l'implantation. Soit σ une séquence d'exécution du produit, $\rho = Chemin\sigma)$ et $\alpha = Trace(\sigma)$.*

$$\textbf{Verdict}(\sigma) = \textbf{Ne Satisfait Pas} \Longrightarrow \alpha \in \mathcal{L}(\mathcal{O}bs)$$

Preuve :

Soit $\sigma_{ACT_r^p}$ et σ_{IUT} les séquences d'exécution respectives du cas de test et de l'implantation correspondant à σ. Notons $\rho_{ACT_r^p}$ = Chemin($\sigma_{ACT_r^p}$). **Verdict**(σ) = **Ne Satisfait Pas** $\Longrightarrow$ $\exists i \exists l.\rho(l) = (p_l^{\text{CT}}, p_l^{\text{IUT}}, \lambda_l)\lambda_l(x_i) \geq c_{l_i}$ et pour chaque $m \in [0 \cdots l]$ $\rho(m) = (q_m^{\text{CT}}, q_m^{\text{IUT}}, \lambda_m)$ et $q_m^{\sigma} \in U_i^{\text{CT}}$ implique $\lambda_m(y_i) \leq c_{u_i}$. Il est facile de voir que $\mathcal{H}(\rho_{ACT_r^p}, L_i^{ACT_r^p}, c_{l_i})$ est évalué à vrai et que $\mathcal{H}(\rho_{ACT_r^p}, U_i^{ACT_r^p}, c_{u_i})$ est évalué à faux. Donc, $\alpha \in \mathcal{L}(AGT_r^p)$. Par ailleurs, $\mathcal{L}(AGT_r^p) \subseteq \mathcal{L}(\mathcal{O}\text{bs})$, d'où le résultat.

5.10 Conclusion

Nous venons de proposer une méthode de génération de cas de test orientée test de propriétés et basée sur les modèles. Nous avons proposé une modélisation des propriétés permettant de prendre en compte des propriétés de "vivacité bornée". La génération se fait entre une spécification partielle $\mathcal{S}$ et un observateur $\mathcal{O}$bs (décrivant la négation de la propriété à vérifier). Le résultat de cette

génération est donné sous la forme d'un graphe de test AGT_p^r contenant les séquences "pertinentes" pour vérifier la propriété donnée. Du graphe de test sont extraits les cas de test ACT_p^r. Une exécution du cas de test est réalisée à l'aide d'un testeur en fixant les paramètres de l'observateur pour délivrer les verdicts qui indiquent si les implantations testées satisfont ou non la propriété donnée.

Comparaison de notre méthode avec d'autres méthodes pour le test de propriétés :

Une extension du test de conformité basée sur la relation **ioco** vers du test de propriétés est également proposée dans [RMT$^+$04] (dans le cas de systèmes finis) et [RMJ05] (dans le cas de systèmes infinis, modélisés par des IOLTS symboliques). Toutefois cette extension diffère sur un certain nombre de points de l'approche que nous avons développée dans ce chapitre.

En premier lieu, l'objectif de [RMT$^+$04] et [RMJ05] est d'intégrer le test dans une méthodologie plus générale qui consiste à vérifier tout d'abord une propriété sur une spécification du système puis à générer des tests à partir de cette spécification "correcte" et de la propriété. L'exécution de ces tests pourra alors permettre de détecter non seulement qu'une implantation est non conforme à cette spécification (vis-à-vis de la relation **ioco**), mais également qu'elle ne vérifie pas la propriété considérée. Dans notre cas, l'objectif est différent : nous ne nous intéressons qu'à la correction de l'implantation par rapport à la propriété, et nous n'avons donc aucune hypothèse sur la spécification (qui peut ou non vérifier la propriété).

Une conséquence importante réside dans la nature des propriétés qui peuvent être prises en compte. Dans [RMT$^+$04], les propriétés considérées sont des propriétés linéaires de sûreté, dont la négation est représentée par un observateur muni d'un état d'acceptation (noté *Violate*). Néanmoins, seul un sous-ensemble de ces propriétés pourra être invalidé lors d'un test : celles qui admettent une trace contre-exemple de la forme $\alpha.a$ où α est une trace de la spécification et a une action de sortie. Ces propriétés sont en effet exactement celles qui sont préservées par la relation **ioco**. Dans l'approche que nous proposons les propriétés susceptibles d'être invalidées sont plus générales : leurs séquences contre-exemples sont de la forme $\alpha.\beta$ où α est une trace de la spécification (éventuellement vide) et β un suffixe de la forme $a.\beta'$ où β' est un suffixe quelconque et a est soit une action de sortie, soit une action d'entrée dans le cas où $\alpha.\alpha'.a$ n'est pas une trace de la spécification (pour α' quelconque). De plus, ces propriétés peuvent être exprimées directement comme des approximations de propriétés de vivacité ce qui facilite leur écriture dans le cas de propriété de vivacité bornée (la structure de l'observateur étant indépendante de la valeur des bornes).

Cette différence influe bien entendu sur la technique de génération de test, et notamment sur l'opérateur utilisé entre la spécification et l'observateur pour construire le graphe de test (définition 5.6.1). Dans notre cas, ce graphe de test est obtenu en complétant un produit synchronisé de manière à ce que toutes les traces de l'observateur soient présentes. Dans le cas de [RMT$^+$04] et [RMJ05] ce produit synchronisé est étendu pour contenir uniquement les traces de la forme $\alpha.a$, où α est une trace de la spécification et a une action de sortie menant à l'état *Violate*.

Nous allons voir maintenant comment appliquer notre méthode de test de propriétés pour effectuer

de la génération de cas de test orientée test de robustesse.

Chapitre 6

Génération et exécution de cas de test pour le test de robustesse

6.1 Présentation du test de robustesse

Les travaux présentés maintenant introduisent d'abord la notion de robustesse de programmes, puis une technique de génération et d'exécution de cas de test de robustesse [FMP05, C.P04].

La robustesse et le test de robustesse sont des notions largement répandues dans le domaine matériel. Ces notions restent cependant des activités moins fréquentes dans le domaine du logiciel, et il convient donc de les définir au préalable dans ce domaine. Notre réflexion sur la notion de robustesse a débuté lors d'une action spécifique (AS STIC n°23 à l'initiative du CNRS). Cette action a été réalisée en collaboration avec les laboratoires LAAS (Toulouse), Labri (Bordeaux), IRISA (Rennes), LRI (Orsay-Paris) et Vérimag (Grenoble). Lors des discussions au sein de cette AS, une première définition considérée pour la notion de robustesse a été celle proposée par l'IEEE :

Définition 6.1.1 (La robustesse selon IEEE) *L'IEEE définit la robustesse comme "Le degré selon lequel un système, ou un composant, peut fonctionner en présence d'entrées invalides ou de conditions environnementales stressantes" (IEEE Std 610.12-1990).*

Si nous considérons que la robustesse est établie en présence de fautes, alors cette notion de robustesse est liée à des propriétés de tolerances aux fautes. La situation des fautes par rapport aux frontières des systèmes testés conduit à distinguer les fautes internes aux fautes externes. La définition de la robustesse de L'IEEE met l'accent sur les fautes externes pour décrire les entrées invalides. Il n'y a pas de raison à priori d'interdire les fautes internes, si celles-ci peuvent être testées. La nature des fautes conduit à distinguer les fautes accidentelles et intentionnelles. En incluant les fautes intentionnelles nous pouvons considérer que les tests de pénétration [Wei95] sont des tests de robustesse ciblant les manipulations non-autorisées de l'information. Le fait de considérer des

aléas qui ne sont pas forcément des fautes implique que la notion de robustesse n'est pas limitée à la notion de tolérance aux fautes. S'appuyant sur la définition de l'IEEE, l'action spécifique a proposé une première application de cette définition pour le test de robustesse de programmes :

Définition 6.1.2 (Le test de robustesse selon l'action spécifique AS23) *Le test de robustesse vise à vérifier la capacité d'un logiciel, ou d'un composant logiciel , à fonctionner de façon acceptable en présence de fautes ou de conditions environnementales stressantes.*

Les fautes possibles présentes dans cette définition sont vues comme des fautes externes au système testé. Elles sont provoquées par le propre environnement du système. Typiquement, les fautes internes sont liées à un aspect externe (de la réalité du système) représenté au sein même du modèle (du système). Dans ce cas, les données et/ou les messages échangés entre les composants d'un système sont corrompus. Le modèle de faute est ainsi lié à la description des interfaces entre composants. Cette approche a été utilisée dans un but de caractériser les modes de défaillance de [AFRS02, Rei99], ou ils étudient les mécanismes de propagation d'erreur au sein d'un système, suite à la défaillance (provoquée, simulée) d'un des composants du système testé. Ces modes ont été utilisés pour tester des mécanismes de tolérances aux fautes pour des calculateurs parallèles [BT97], ou encore pour vérifier des propriétés de systèmes répartis en présence de fautes de communications [DJMT96]. Nous complétons les deux premières définitions par :

Définition 6.1.3 (Un fonctionnement acceptable) *L'implantation testée garde un fonctionnement acceptable si elle préserve un ensemble particulier de propriétés (les propriétés de robustesse) lors de son exécution.*

Ces propriétés garantissent à un utilisateur que le comportement d'une application ou d'une fonction particulière reste prévisible, malgré un environnement d'exécution non nominal. De manière générale, les propriétés de robustesse vérifient un aspect bien particulier du mécanisme du système testé. Elles peuvent être de nature différente des propriétés fonctionnelles pour vérifier par exemple : "qu'une connexion réseau est sécurisée (fiable)", "qu'un "buffer" ne déborde jamais" ou "qu'un message d'erreur est correctement émis dans une situation anormale".

Conformité ou robustesse ?

Pour le test de conformité (avec la relation **ioco**), si une sortie du système n'est pas spécifiée, alors l'implantation est non conforme (verdict Fail). La relation exclue l'action de sortie non spécifiée et cette sortie est obligatoirement perçue comme une erreur de construction de l'implantation. Une question se pose, les sorties non spécifiées sont-elles de vraies erreurs dans l'implantation ? Nous pouvons envisager qu'une action de sortie même non spécifiée n'introduise pas nécessairement une faute dans l'exécution du système.

Propriété et/ou robustesse ?

Comme nous l'avons vu, parler de robustesse de programmes implique l'introduction de la notion de fautes. En effet, un programme est robuste si il préserve certaines propriétés de robustesse en présence de fautes. Les fautes ne sont pas forcément dues à une mauvaise écriture de l'implantation. Elles peuvent être induites par les valeurs de paramètres provenant de l'environnement de fonctionnement du composant (communicant) testé. Dans tous les cas, les fautes sont perçues comme des aléas affectant les actions d'entrée et de sortie visibles des implantations. Or, selon la relation **ioco**, l'implantation testée risque de ne pas être conforme à la spécification de référence, car certaines actions (avec des fautes) peuvent être des sorties non spécifiées. Nous n'envisageons donc pas de faire du test de conformité sur des modèles mutés (intégrant certaines fautes), mais nous proposons d'adapter le test de propriétés en prenant en compte des aléas provenant de l'environnement des composants testés.

Une implantation peut-elle être prévue pour devenir robuste ?

Préserver certaines propriétés de robustesse implique qu'une implantation peut tenir compte (dans sa réalisation) de certaines fautes à éviter. De telles implantations ont (peut être) besoin de nouvelles actions (internes et/ou visibles) pour préserver la propriété en présence des fautes identifiées. Les actions ajoutées peuvent être spécifiées ou non (actions incluses ou non incluses dans l'ensemble des actions de la spécification). Dans ces conditions, ces implantations (selon **ioco**) peuvent devenir non conformes. Si nous gardons la relation de conformité, il ne nous est pas toujours possible de tester la robustesse d'une implantation. Pour montrer qu'un comportement peut être robuste, nous devons identifier les actions présentes dans une implantation soumise à un environnement dégradé par rapport à la spécification :

1. Les actions sont inconnues : elles n'appartiennent pas à l'ensemble des actions de la spécification.
2. Elles sont connues, mais mal positionnées (inopportunes pour la spécification) : elles appartiennent à l'ensemble des actions de la spécification, mais elles sont exécutées de manière inopportunes selon les comportements attendus par la spécification.
3. Elles sont corrompues : elles n'appartiennent plus à l'ensemble des actions de la spécification puisque les valeurs de leurs paramètres sont erronées.

Dans tous les cas (et selon la relation **ioco**) si ces actions sont des sorties, alors elles provoquent la fin de l'exécution du test de conformité. L'implantation testée devient mal construite et donc non conforme car elle ne respecte pas la relation (de conformité). Pour tester de telles implantations, nous proposons le test de robustesse.

Le but du test de robustesse.

La méthode de test de robustesse doit permettre de vérifier des programmes en présence des aléas

et des nouveaux comportements induits par des aléas. Nous proposons de répondre aux questions suivantes : Si une implantation fonctionne dans des conditions normales (test de conformité), fonctionne-t-elle encore de manière acceptable :

- Dans un autre environnement (que celui des conditions normales) ?
- En présence d'erreurs (fautes, aléas) dans les actions visibles ?
- Si l'implantation comporte des actions ou des comportements non initialement spécifiés (destinés par exemple à en assurer la robustesse) ?

Nous voulons tester si un programme peut garder un comportement acceptable en présence de fautes ou d'aléas environnementaux. Un comportement est acceptable, si le système testé préserve (satisfait) une propriété de robustesse, lors de son exécution. Le domaine d'application est constitué par les systèmes communicants. L'approche pour générer les cas de test de robustesse est basée sur les modèles (automates étendus communicants et automates de Rabin paramétrés). Les cas de test de robustesse sont construits automatiquement ([C.P03]) à partir des comportements d'une spécification et d'une propriété de robustesse, et sont exécutés par un testeur. Une exécution est une interaction entre le testeur et l'implantation. Le résultat de cette exécution délivre un verdict suivant une relation de robustesse.

Nous proposons deux axes pour générer des cas de test de robustesse :

1. Il faut prendre en compte les actions non spécifiées :
   Une réponse est d'inclure dans les cas de test les actions non prévues par la spécification. Les cas de test vont être construits à l'aide de la spécification et des propriétés de robustesse, une solution est que la propriété de robustesse décrit ou accepte toutes les actions de l'implantation (même celles non spécifiées par la spécification). Nous proposons de représenter la propriété de robustesse (ou sa négation) par un modèle d'automate de Rabin paramétré, comme dans le cas du test de propriétés.

2. Il faut prendre en compte les erreurs ou les fautes :
   Nous avons vu, dans le test de propriétés, qu'une spécification permet de guider la construction des cas de test. Nous proposons dans le cadre du test de robustesse d'intégrer dans le modèle de spécification des fautes modélisant l'environnement de fonctionnement de l'implantation testée.

Le test de propriétés base pour le test de robustesse.

Les changements apportés (suivant nos définitions) par rapport au test de propriétés sont :

1. De représenter des propriétés dites de "robustesse".
2. De prendre en compte les aléas dans le modèle de la spécification. Les aléas sont provoqués par l'environnement de fonctionnement du système. En d'autres termes, les modèles de la

spécification vont être enrichis de ces aléas, cette spécification est nommée : ”spécification mutée”.

3. La définition de la notion de fonctionnement ou de comportement acceptable pour une implantation (malgré certaines fautes). En d'autres termes, une relation de robustesse (équivalente à une relation de satisfiabilité vue pour le test de propriétés) interprétant les réactions d'une implantation lorsqu'elle est soumise aux fautes. Une implantation prévoit-elle des conditions extrêmes et garde-t-elle un fonctionnement acceptable correspondant aux attentes ? C'est à dire satisfait-elle les propriétés de robustesse ?

6.2 Environnement nominal, environnement dégradé

À partir du moment ou un comportement est prévu pour une implantation, il est décrit par une spécification et sous entendu dans un environnement idéal. Nous supposons que les actions et paramètres d'entrée constituent pour une implantation à tester son environnement. Les valeurs de l'environnement ne sont généralement pas explicitées, mais font partie intégrante de la définition de la spécification (dans la suite du document cette spécification contenant un environnement initial est nommée spécification nominale). Maintenant, nous supposons qu'une implantation peut évoluer dans un autre environnement que celui initialement prévu. Nous supposons qu'une implantation est constituée d'un ou plusieurs composants communicants.

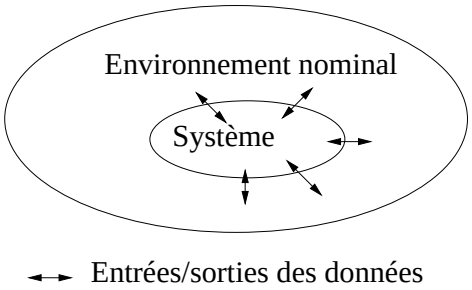

FIG. 6.1 – Configuration d'un système (composant) avec son environnement.

Pour tester un système, un testeur, qui est lui-même un programme, simule une partie de cet environnement avec une séquence composée d'actions de l'interface. Les comportements du système (en réaction à cette simulation) sont alors observés à travers cet environnement.

Pour le test de robustesse, nous proposons de composer avec l'interface initiale du composant testé, et de voir comment l'environnement peut (doit) évoluer (se comporter) lorsque ce composant est soit réutilisé dans un système existant, soit en présence de fautes. Cette évolution d'environnement aura pour impact de changer (supprimer, ajouter ou modifier) des actions ou des paramètres d'entrée et de sortie.

NOTE :
L'environnement de test du test de robustesse pourra prendre en compte les changements d'interface et donc définir de nouveaux points de contrôle et d'observation.

Nous proposons d'identifier par un *environnement nominal* l'interface (au sens du test de conformité) du composant à tester défini implicitement par la spécification initiale. Puis, nous proposons de recomposer l'environnement nominal lorsque le composant à tester fait partie d'un nouveau système. Ce nouvel environnement ainsi identifié sera nommé *environnement dégradé* :

1. L'environnement **nominal** : il décrit les actions d'échanges entre les composants définies de façon idéales. Cet environnement (généralement implicite) est défini initialement et intègre la spécification de référence que nous nommons spécification nominale. Les alphabets et actions de la spécification nominale sont décrits uniquement en fonction des conditions nominales de cet environnement (idéales, optimales).

2. L'environnement **dégradé** : il correspond aux descriptions des conditions réelles de fonctionnement des programmes. Il donne en partie les paramètres et actions échangés pendant l'exécution des implantations. Nous le considérons comme dégradé car cet environnement peut induire des fautes sur l'implantation (par rapport au nominal). Les fautes sont en réalité les nouvelles conditions de fonctionnement, mais aussi des indications sur d'éventuels, pannes, dysfonctionnements, ou élargissements des domaines d'entrée des données. Dans ce cas, un environnement dégradé peut être donné par un utilisateur sous la forme d'un modèle qui sera ensuite intégré aux spécifications nominales. La spécification intégrant l'environnement dégradé sera nommée spécification mutée ou dégradée.

L'exemple présenté dans la section suivante (6.3) illustre les différences entre le test de conformité (en appliquant la relation **ioco**) et le test de robustesse. Pour cet exemple, nous proposons qu'une implantation puisse ne pas être conforme à une spécification, mais, préserve un fonctionnement acceptable vis-à-vis d'une propriété de robustesse.

6.3 Un exemple

Nous considérons trois composants un Client, un Serveur et un Buffer communicants selon la description de la figure 6.2 :

FIG. 6.2 – Un système communicant

Nous testons uniquement le comportement du composant Client qui a pour spécification formelle :

1. Réception d'un entier 1 ou d'un entier 2 provenant du serveur.
2. Envoi de cet entier 1 ou 2 vers le Buffer.

L'environnement du Client est composé d'un serveur envoyant des entiers 1 ou 2, et d'un Buffer réceptionnant les entiers. À la fin d'une exécution le Buffer contient des 1 et des 2.

Pour cet exemple nous allons représenter :
- Une spécification (figure 6.4) : modélisant les comportements attendus du Client qui accepte et retransmet des entiers.
- Un environnement nominal : constitué des actions acceptant les entiers 1 ou 2 provenant du serveur, via un signal s, et un canal c et des actions de retransmissions des entiers 1 ou 2 en direction du Buffer, via un signal buff, sur un canal d. Dans cet environnement les canaux sont garantis sans corruption de données (canaux fiables).
- Une propriété : "Si l'entier 1 est reçu par le Client l'entier 1 est placé dans le Buffer dans un délai raisonnable sinon la transaction est abandonnée".
- Un environnement dégradé (de fonctionnement) : qui indique que le canal c peut éventuellement comporter des corruptions de données.
- Une Implantation 1 (figure 6.5) : un processus (Client) avec communication fiable et conforme à l'environnement nominal.
- Une Implantation 2 (figure 6.6) : un processus (Client) inclut une simulation de communication non fiable car induite par l'environnement dégradé.

La propriété de robustesse que nous souhaitons vérifier est la suivante : " Inévitablement si un entier 1 est reçu par le Client $c\,?s(1)$ un entier 1 est placé dans le buffeur $d\,!buff(1)$ dans un délai raisonnable sinon la transaction est abandonnée $c\,!$abort". Nous représentons la négation de cette propriété par l'automate de Rabin paramétré suivant :

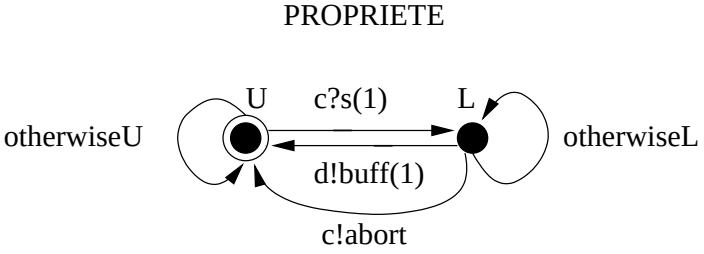

FIG. 6.3 – Propriété de robustesse

Dans la propriété, l'action "$otherwiseU$" (resp. "$otherwiseL$") permet d'accepter toutes les actions d'entrées et de sorties autres que $c\,?s(1)$ (resp. $d\,!buff(1)$ ou $c\,!abort$). Les états L et U représentent les états distingués de l'automate de Rabin paramétré (des paramètres cL et cU seront associés aux états U et L lors de l'exécution du cas de test produit à partir de cette propriété).

La figure 6.4 est la spécification du composant Client. Elle contient deux canaux c et d, les signaux s et $buff$, la variable x , et $c\,?s(x)$ une entrée, $d\,!buff(x)$ une sortie :

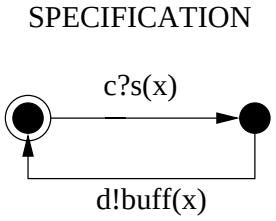

FIG. 6.4 – Spécification

À partir de cette description, l'environnement nominal est constitué des actions $c?s(x)$ et $d!buff(x)$ (et toutes les expressions pouvant intervenir dans ces actions) et implicitement notifie que les canaux c et d sont fiables (soit pas de corruption de données lors des échanges). Plusieurs implantations peuvent être envisagées pour représenter le Client. Pour avoir un peu plus de lisibilité dans les implantations nous proposons également une description du composant serveur envoyant des entiers 1 puis 2.

Une première implantation 1 est de respecter les conditions de l'environnement nominal. L'implantation 1 (Client de la figure 6.5) est une représentation naïve des actions de la spécification (avec implicitement des canaux fiables). Le serveur qui représente une partie de l'environnement nominal est supposé envoyer un entier 1 puis un entier 2. Le Client qui reçoit les entiers 1 et 2 les retransmet directement (un à un) au Buffer. À la fin d'une exécution le Buffer contient les entiers 1 et 2.

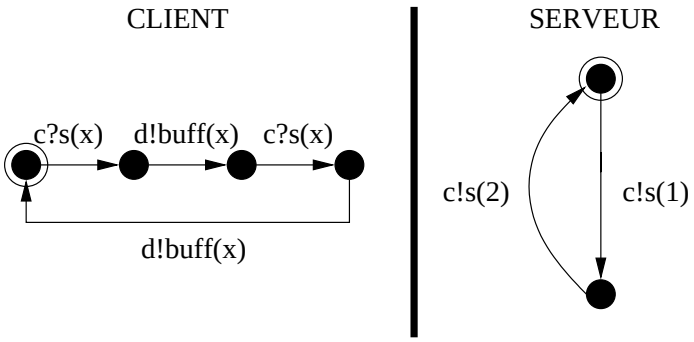

FIG. 6.5 – Implantation 1

Nous supposons maintenant disposer d'un environnement dégradé. Il contient une faute qui est : "Les canaux deviennent non fiables, impliquant des pertes de données possibles pendant une communication entre le Client et le Serveur". L'implantation 2 (figure 6.6) est conçue (élaborée robustesse) pour garantir un fonctionnement acceptable même en présence de pertes de données. Compte tenu de la dégradation de l'environnement, le Client (par un dialogue avec le serveur) vérifie auprès du serveur si la donnée reçue est bien une donnée à retransmettre au Buffer. Cette vérification est faite à l'aide de deux nouvelles actions (au niveau Client) $c!s(x)$ et $c?s(e)$, pour $x \in \{1,2\}$ et

$e \in \{\text{ok}, \neg\text{ok}\}$ et une vérification avec les conditions $e \neq$ "ok" et $e ==$ "ok". Si les pertes de données sont persistantes (au bout d'un nombre d'essais de confirmation fixé à $Cpt < 10$, avec Cpt un entier représentant un décompte des actions rejetées) un abandon de transaction est signalé par l'action c !abort. Cette implantation 2 respecte la propriété, indiquant qu'à la fin d'une exécution "réussie" le Buffer contient les entiers 1 et 2.

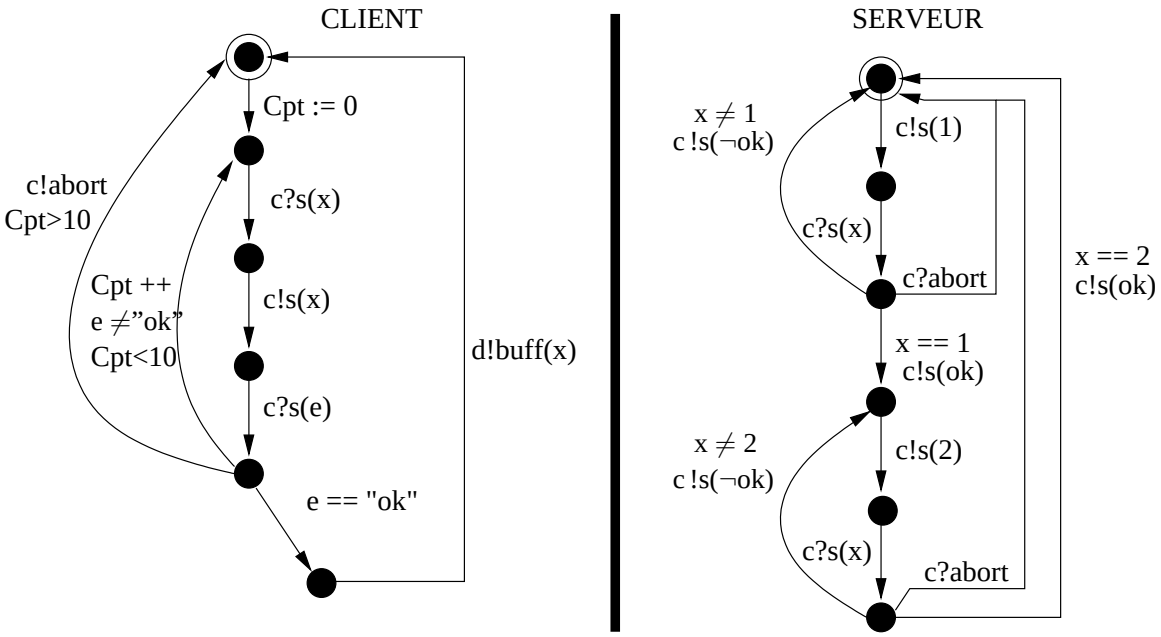

FIG. 6.6 – Implantation 2 élaborée robustesse

Èvolution de l'environnement en fonction de l'implantation 2 :

FIG. 6.7 – Un système communicant suivant l'implantation 2

- Nous ne décrivons pas la génération et l'exécution de cas de test de conformité, mais nous pouvons conclure pour chaque implantation par :

Conforme : Selon la relation **ioco** l'implantation 1 "serait" conforme. En fait, nous pouvons l'affirmer car le modèle de représentation de l'implantation 1 est identique à la description de la spécification. Donc tous les cas de test produits aboutiraient à un verdict Pass. En d'autres termes les traces de l'implantation sont incluses dans celles de la spécification : (STrace(implantation 1) $\subseteq$ STrace(S)$\Longrightarrow$ implantation 1 **ioco** S). Pour vérifier la propriété, il faudrait utiliser par exemple un model-ckecker.

Non conforme : Selon la relation **ioco** et les cas de test produits, l'implantation 2 est non conforme, soit à cause des sorties non spécifiées par la spécification nominale, soit à cause d'une absence de réponse lors de l'exécution des cas de test. Dans ce cas (et pour un bon nombre d'outils d'exécution de test de conformité) un timeout expire provoquant un verdict Fail. Exemple si $c\,?s$(x).$c\,!s$(x).$c\,?s$(x) $\not\subseteq$ $c\,?s$(x).$d\,!buff(x)$

- Nous ne donnons pas la génération et l'exécution de cas de test de robustesse, mais nous pouvons conclure pour chaque implantation par :

  Robuste : L'implantation 1 et l'implantation 2 sont "supposées" robustes pour la propriété de robustesse donnée. En effet pour l'implantation 1, il y a alternance des entiers 1 et 2 sans perte de données, donc si un entier 1 est transmis au Client, un entier 1 est transmis au Buffer. Dans l'implantation 2, il y a vérification des données donc si un entier 1 est transmis après moins de 10 tentatives ($Cpt < 10$) d'échanges de confirmation, un entier 1 est contenu dans le Buffer. Sinon un message d'abandon ($c\,!abort$) est transmis si les tentatives pour transmettre le bon entier dépassent les 10 essais ($Cpt > 10$), ceci respectant bien la propriété donnée.

En Résumé :

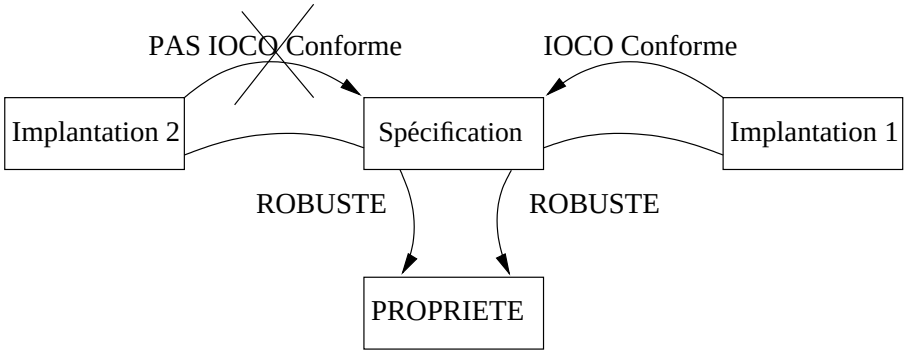

FIG. 6.8 – Implantation 2 non conforme et robuste - Implantation 1 conforme et robuste

L'implantation 2 n'est plus conforme à la spécification, mais satisfait une spécification partielle

Nous montrons par cet exemple, qu'une implantation non conforme à une spécification donnée peut être robuste par rapport à certaines propriétés.

La méthode de test de robustesse que nous allons présenter enrichit la spécification nominale avec de nouveaux paramètres de fonctionnement de l'implantation testée. Nous disposons alors d'indications supplémentaires dans la spécification pour générer et exécuter des cas de test tout en gardant une description partielle du fonctionnement de l'implantation. Nous allons voir maintenant comment générer des cas de test de robustesse.

6.4 Le test de robustesse basé sur les modèles

Les différentes méthodes de test présentées jusqu'à maintenant ont des caractéristiques communes. Comme pour le test de conformité, et maintenant le test de propriétés, nous proposons pour le test de robustesse de stimuler l'implantation à tester en lui fournissant des entrées, et d'observer ses réponses. Les verdicts (suivant une relation de robustesse) sont délivrés en fonction des observations pendant l'exécution d'un cas de test. La mise en place du test de robustesse (figure 6.9) implique de disposer de plusieurs éléments clés pour générer les cas de test :

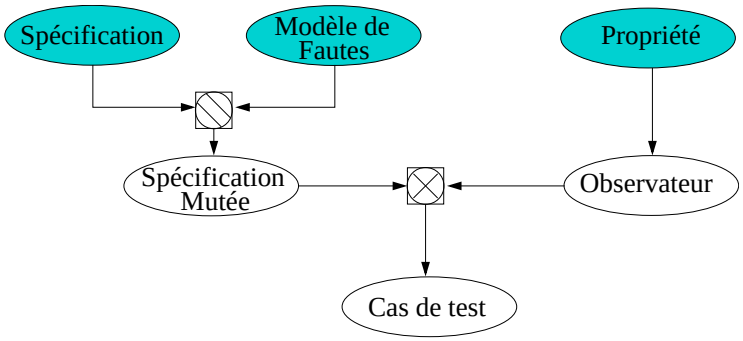

FIG. 6.9 – Génération de cas de test pour le test de robustesse

Nous allons définir plus formellement les ingrédients suivants :
- le modèle de faute,
- La mutation et les spécifications Mutées $\mathcal{S}_m$,
- Les propriétés de robustesse,
- La relation de robustesse.

NOTE :
Les modèles de la spécification, de l'observateur et le principe d'exécution des cas de test restent les mêmes que vus dans la partie test de propriétés (chapitre 5).

- La génération de cas de test de robustesse est réalisée avec un produit synchrone entre un observateur $\mathcal{O}$bs et une spécification mutée $\mathcal{S}_m$: $\mathcal{O}$bs $\otimes$ $\mathcal{S}_m$,
- Le verdict résultant de l'exécution est une fonction de l'ensemble des $\mathcal{O}$bservations dans celui des verdicts : $\mathcal{O}$bservations $\longrightarrow$ `verdict`. Cette fonction permet de définir les implantations robustes ou non robustes. Le domaine de `verdict` reste que celui du test de propriétés {**Satisfait**, **Ne Satisfait Pas**}.

6.5 Mutation d'une spécification

6.5.1 Techniques de test basées sur la mutation

Plusieurs approches existent pour représenter les mutations, nous en citons quelques unes :

1. Soit les fautes sont intégrées directement dans les modèles de spécification et l'application d'un test fonctionnel sert à vérifier l'impact des mutations.
2. Soit les fautes sont injectées pendant l'exécution (pas besoin de modèles d'automates) et le test réalisé est un test d'injection de fautes [BDD$^+$91, LAC$^+$96].
3. Soit les fautes sont extraites du code des implantations, et dans ce cas, les machines de mutations permettent de représenter de façon compacte un ensemble de machines d'implantation possibles (et potentiellement erronées) d'une machine de spécification donnée, en fonction de l'hypothèse de fautes.

Les mutations et tests par mutations

Le principe des mutations est d'altérer les données existantes (introduction de fautes dans des modèles [BA85] ou modélisation des fautes par un ensemble de machines d'implantation pour des FSM[1] [Pet01, KPY99, PY92]). Dans [Bud80, DGK$^+$88], l'ensemble des fautes possibles d'une implantation (pour une spécification donnée) est décrit par un ensemble de mutants, appelé domaine de faute. Il existe plusieurs méthodes pour muter des données, par exemple :

– [MR01] produit des cas de test et remplace systématiquement les données élémentaires pertinentes du cas de test par une donnée élémentaire altérée.

– [DMM96] propose de dégrader les interfaces, telles que les appels aux méthodes, les paramètres, ou les variables globales (contenues dans le code des programmes et ciblant les actions entre composants).

[1] Machine d'états finis

Généralement appliqué pour vérifier le fonctionnent d'un logiciel, le test de mutation est une analyse des résultats (domaine de sortie) en fonction des fautes et des sorties prédéfinies. L'intérêt des théories du test des mutations, est que les mutants peuvent être produits pour toutes sortes de programmes à moindre coût.

Nous proposons, dans cette section, les modèles de représentation (spécification, observateur . . .), une proposition de classification de fautes, puis les opérations permettant d'intégrer les fautes au modèle.

6.5.2 Spécification

Formellement les mutations sont définies comme des transformations sur la syntaxe d'une spécification. Nous donnons donc une description de cette syntaxe sous forme d'une grammaire abstraite et nous proposons un ensemble de fonctions de mutations qui opèrent sur cette grammaire.

Nous proposons qu'une spécification est un ensemble d'automates étendus communicants. La syntaxe est constituée des types de données, de déclarations, des signaux de communication, des canaux de communication avec une déclaration de leur attribut, des variables (éventuellement partagées), et des déclarations de processus. Un signal est constitué d'un nom unique et d'un ensemble de paramètres typés. Les signaux sont les objets transportés d'un processus à l'autre à travers des canaux.

NOTE :
Nous présentons cette grammaire simplifiée (grammaire abstraite pour les automates étendus communicants qui sera comparable avec celle du langage IF de la section 8.3.2 que nous utiliserons pour définir les formats des spécifications utilisées).

Spécification : : =
TYPE : $\text{type}_1 \ldots \text{type}_n$
SIGNAL : $\text{signal}_1(\text{type}_1 \ldots) \ldots \text{signal}_n(\text{type}_n \ldots)$
CANAL : $\text{canal}_1 \ldots \text{canal}_n$
ATTRIBUT_{CANAL} : ((unicast | multicast) , (fiable | nonfiable))
VARIABLE : $\text{var}_1 \ldots \text{type}_n$
PROCESSUS : $\text{processus}_1 \ldots \text{processus}_n$

Un processus est identifié par un nom unique PROCESSUS_{id}. Il est constitué d'un ensemble d'états, de déclarations de variables et de déclarations de canaux. Un processus représente l'ensemble de transitions qui permettent de passer d'un état à l'autre en évaluant les gardes et en exécutant des actions. La valeur d'une garde conditionne l'exécution de l'action.

PROCESSUS : : =
PROCESSUS_{id}
ETAT : $\text{état}_1 \ldots \text{état}_2$

VAR_DECLARATION : var_declaration$_1$ ... var_declaration$_n$
CANAL_DECLARATION : canal_declaration$_1$ ... canal_declaration$_n$
TRANSITION$_{processus}$*

Chaque variable déclarée comporte un nom IDF[2] et un type unique. Chaque canal représente le lien de communication entre les processus. Il est défini par un nom unique et possède des attributs. Un attribut précise le nombre de communications du même type (en fonction des destinataires) et la fiabilité des communications.

VAR_DECLARATION : :=
IDF : TYPE

CANAL_DECLARATION : :=
CANAL : (PROCESSUS$_{id}$ TO PROCESSUS$_{id}$, ATTRIBUT$_{CANAL}$)

Plusieurs transitions sont possibles dans un même processus et sont représentées par :

TRANSITION$_{processus}$: : =
FROM état$_{id}$
GARDE
ACTION
TO état$_{id}$

Les actions associées aux transitions sont respectivement des affectations, des émissions de signaux, des réceptions de signaux. Une affection consiste à attribuer la valeur d'une expression à une variable initiale. Les émissions et les réceptions précisent donc le signal émis ou attendu, le canal concerné et le paramètre transmis. Les expressions "EXPRESSION" sont classiques et sont construites à partir des variables et des opérations possibles sur les variables.

ACTION : : =
AFFECTATION | EMISSION | RECEPTION

GARDE : := EXPRESSION
AFFECTATION : := var$_{id}$:= EXPRESSION
EMISSION : := canal$_{id}$ signal$_{id}$(EXPRESSION*)
RECEPTION : := canal$_{id}$ signal$_{id}$(var$_{id}$*)

[2] IDentiFicateur

6.5.3 Modèle de faute

Les Fautes

Selon IEEE ([IEE90]), une faute est une instruction ou une donnée supposée incorrecte dans un programme. Les programmes que nous testons sont des systèmes contenant des composants interagissant entre eux. Pour un composant, toutes les données échangées proviennent de son environnement. Nous proposons de faire une classification des ”fautes” (possibles). Les fautes traduisent des pannes (arrêt définitif d'une communication), des coupures (arrêt temporaire d'une communication), des dysfonctionnements (une condition est mal évaluée) ou des élargissements des données. La liste des fautes n'est pas exhaustive et l'ensemble peut évoluer suivant les caractéristiques des environnements de fonctionnent et dépendre de l'architecture des applications (par exemple, le manque de fiabilité entre les liaisons). Les fautes peuvent aussi être données par un tiers extérieur. Nous proposons d'identifier deux types de fautes : (1) Les fautes portant sur des données, (2) Les fautes portant sur le contrôle. Pour le test de robustesse tel que nous le présentons, nous ne nous intéressons qu'aux fautes que nous allons pouvoir provoquer au moment du test. Nous parlons alors de modèle de faute :

Définition 6.5.1 (Modèle de faute) *Le modèle de faute (nommé $\mathcal{MF}$) représente la liste des fautes :*

1. *De données :*

   – *Des* **élargissements** *de données (traduits par des changements de types des données) : Une variable peut prendre un domaine de valeurs différent que celui décrit initialement. Bien que nous le notions ”élargissements”, le nouveau domaine de valeur peut être plus grand ou plus petit que le domaine existant. Ce type de faute peut être testé, car il affecte les valeurs des actions échangées entre les composants d'un système.*

   – *Des* **dysfonctionnements** *(significatifs au niveau des valeurs et variables échangées) : Contrairement aux élargissements de données (qui sont des erreurs globales au programme), un dysfonctionnement est ponctuel à un paramètre d'une action. Les dysfonctionnements regroupent particulièrement les corruptions ou pertes de données échangées entre les composants.*

2. *De contrôle :*

   – *Des* **coupures** *(perte locale de communication entre composants) : Une communication est interrompue pendant une courte durée.*

   – *Des* **pannes** *(arrêt définitif de la communication entre composants) : Un lien ou un composant est définitivement défaillant, plus aucune communication n'est possible.*

NOTE :
Nous provoquons les fautes (intégrées dans un environnement dégradé) en fonction des paramètres différents de ceux décrits par un environnement nominal. Au niveau des automates, les fautes affectent les données, les émissions, les réceptions, et les gardes.

À partir des fautes suggérées et la grammaire proposée, nous présentons un domaine de faute portant sur l'ensemble des types et des transitions, nommé : dMF-Transition. Le domaine de faute propose pour chaque type ou action d'une transition la mutation possible entre les éléments initiaux et la faute provoquée. Ce domaine est constitué des sous-domaines de faute en fonction du type de faute. Nous proposons dMF-Transition = dMF-changement-type $\cup$ dMF-perte-donnée $\cup$ dMF-corruption-reception $\cup$ dMF-corruption-emission $\cup$ dMF-contrôle-externe $\cup$ dMF-coupure-reception $\cup$ dMF-coupure-emission $\cup$ dMF-panne-reception $\cup$ dMF-panne-emission $\cup$ dMF-panne-composant où chaque sous domaine est représenté de la façon suivante :

1. Pour les changements de type de données : dMF-changement-type : {(IDF $\times$ TYPE $\times$ TYPE)*} où (idf, type, type$'$) décrit la nouvelle déclaration de la variables idf avec type$'$ la mutation du type de la variable.
2. Pour une perte de données : dMF-perte-donnée : {(CANAL $\times$ ATTRIBUT(CANAL) $\times$ ATTRIBUT(CANAL$'$))*} où (canal,attribut(canal),attribut(canal$'$) décrit la nouvelle déclaration du canal avec attribut(canal$'$) la mutation attribut(canal).
3. Pour la corruption de données :
   - dMF-corruption-reception : {(SIGNAL(IDF) $\times$ IDF$'$)*} où (signal(idf), idf$'$) décrit le nouveau signal(idf$'$) avec idf$'$ la mutation de l'idf initial.
   - dMF-corruption-emission : {(SIGNAL(EXPRESSION) $\times$ EXPRESSION$'$)*} où (signal(exp), exp$'$) décrit le nouveau signal(exp$'$) avec exp$'$ la mutation de l'exp initiale.
   - dMF-contrôle-externe : {(GARDE $\times$ GARDE$'$)*} où (garde, garde$'$) décrit que la nouvelle garde$'$ est la mutation de garde.
4. Pour les coupures :
   - dMF-coupure-reception : {(SIGNAL(IDF) $\times$ coupure $\times$ GARGE$_a$ $\times$ GARGE$_b$)*} où (signal(idf), coupure, garde$_a$, garde$_b$)} décrit qu'un niveau signal de réception garde$_b$coupure est possible et que l'ancien signal de réception est gardé : garde$_a$signal(idf)
   - dMF-coupure-emission : {(SIGNAL(EXPRESSION) $\times$ coupure, $\times$ GARGE$_a$ $\times$ GARGE$_b$)*}, où (signal(exp), coupure, garde$_a$, garde$_b$) décrit qu'un niveau signal de réception garde$_b$coupure est possible et que l'ancien signal d'émission est gardé : garde$_a$signal(idf).
5. Pour les pannes :
   - dMF-panne-reception : {(SIGNAL(IDF) $\times$ panne)*} où (signal(idf), panne) décrit que le signal(idf) de réception n'est plus possible et qu'un nouveau signal de réception de panne le remplace.
   - dMF-panne-emission : {(SIGNAL(EXPRESSION) $\times$ panne)*} où (signal(exp), panne) décrit que le signal d'émission n'est plus possible et qu'un nouveau signal de réception de panne le remplace.
   - dMF-panne-composant : {(ETAT $\times$ panne $\times$ ETAT)*} où (état, panne, état$'$)} décrit une création d'une nouvelle action panne au niveau de état permettant d'aller à état$'$.

Les tableaux suivants représentent les différentes mutations envisagées (selon les sous-domaines de mutation présentés). Dans le tableau, la partie ”Ancien noeud” décrit la transition initiale à muter à partir d'un état de l'arbre, et le ”Nouveau noeud” est le résultat de la mutation de ”Ancien noeud” selon le ”Domaine de faute”.

Identification des fautes provoquant des mutations de données :

Type de Faute	Ancien Nœud	Domaine de Faute	Nouveau Nœud
Elargissement des entrées			
changement de types	var x : T	$\{(x,\text{T},\text{T}')\}$	var x : T$'$
Dysfonction−−nement			
pertes de données	c :(p_1 to p_2,Fiable) c :(p_1 to p_2,Multicast)	$\{(c,\text{Fiable},\neg\text{Fiable}),$ $(c,\text{Multicast},\text{Unicast})\}$	c :(p_1 to p_2,$\neg$Fiable) c :(p_1 to p_2,Unicast)
corruption en réception	$c?s(x)$ $q \longrightarrow q'$	$\{(s(x),x')\}$	$\begin{cases} c?s(x) \\ x := x' \end{cases}$ $q \longrightarrow q'$
corruption en émission	$c!s(e)$ $q \longrightarrow q'$	$\{(s(e), e')\}$	$c!s(e')$ $q \longrightarrow q'$
fonction externe de contrôle	$[b]$ $q \longrightarrow q'$	$([b],b')$	$[b']$ $q \longrightarrow q'$

Un élargissement affecte le type d'une donnée numérique. Au niveau de la variable ses nouvelles valeurs constituent un intervalle numérique différent de celui prévu intialement (environnement nominal).

Les dysfonctionnements sont des pertes de données, et correspondent à un nouveau comportement :

1. Les composants perdent des liens extérieurs, et passent de multicast à unicat ou perdent des messages et passent d'un canal fiable à un canal non fiable. Nous ne proposons pas l'inverse,

car il serait difficile d'envisager d'augmenter l'environnement d'un composant par simple mutations. Il faut pour cela reprendre toute l'architecture du système et dans ce cas l'environnement disponible décrirait ces nouveaux changements.

2. Les corruptions en émission, en réception et les fautes des fonctions externes de contrôle, sont locales et portent sur une transition précise.

Indentification des fautes provoquant des mutations de contrôle :

Type de Faute	Ancien Nœud	Domaine de Faute	Nouveau Nœud
coupure			
d'un canal de réception	$q \xrightarrow{c\,?s(x)} q'$	$\{(s(x)$,coupure,$[a]$,$[b])\}$	$q \xrightarrow{[b]\,?\text{coupure}} q'$ $q \xrightarrow{[a]\,c\,?s(x)} q'$
d'un canal d'émission	$q \xrightarrow{c\,!s(e)} q'$	$\{(s(e)$, coupure, $[a]$, $[b])\}$	$q \xrightarrow{[b]\,?\text{coupure}} q'$ $q \xrightarrow{[a]\,c\,!s(e)} q'$
Panne			
d'un canal de réception	$q \xrightarrow{c\,?s(x)} q'$	$\{(s(x)$, panne$)\}$	$q \xrightarrow{?\text{panne}} q'$
d'un canal d'émission	$q \xrightarrow{c\,!s(e)} q'$	$\{(s(e)$,panne$)\}$	$q \xrightarrow{?\text{panne}} q'$
d'un composant	q	$\{(q$, panne$)\}$	$q \xrightarrow{?\text{panne}} q'$ "q$'$ est un nouvel état"

Les coupures sont temporaires et ponctuelles sur les émissions ou les réceptions existantes du système. Pour nous permettre de provoquer une coupure, une garde $[a]$ est ajoutée à l'action qui peut subir la coupure. L'évaluation de cette condition $[a]$ nous permet d'envisager ou non de prendre la transition existante pendant le test. Nous provoquons une situation de coupure en ajoutant une

nouvelle transition munie d'une garde $[b]$ donnée en parallèle à la transition existante (à couper). L'action de coupure donnée correspondant à une réception et donc une action contrôlable par le testeur. Les gardes $[a]$ et $[b]$ des transitions sont fixées en fonction de la coupure souhaitée (la garde $[a]$ doit être le complémentaire de $[b]$, car si $[a] \vee [b] \neq \emptyset$, la coupure n'est plus contrôlée par l'environnement).

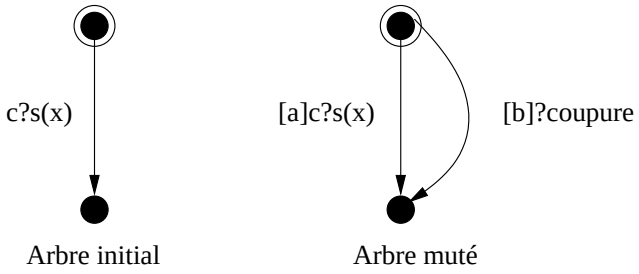

FIG. 6.10 – Automate muté avec une coupure.

Une panne d'émission ou de réception est définitive, bloquante et locale à un état, car l'action de panne remplace l'action existante sortant de l'état. Une panne d'un composant est une action définitive et aboutit à **un nouvel état puits**.

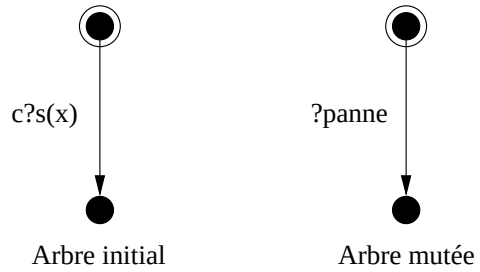

FIG. 6.11 – Automate muté avec une panne.

6.5.4 Règles de mutations

À chaque terme de la grammaire est associé un arbre abstrait. Les règles de mutation vont transformer cet arbre abstrait (associé à une spécification). Nous avons défini par les tableaux ci-dessus un ensemble de transformations possibles, une par mutation souhaitée. Nous définissons maintenant une fonction Apply qui permet de transformer un arbre abstrait à l'aide des transformations (mutations) souhaitées :

Définition 6.5.2 (Fonction de mutation) *Soit la fonction Apply qui prend comme paramètres* Δ *: la fonction qui permet de muter un nœud d'un arbre abstrait, f(*$t1, \ldots, t_n$*) un arbre abstrait, avec* $t1, \ldots, t_n$ *les nœuds de l'arbre, et MF le modèle de faute, nous avons :*

$$Apply\ (\Delta, f(t_1, t_2, \ldots, t_n), MF) =$$
$$f(Apply(\Delta_1, t_1, MP), Apply(\Delta_2, t_2, MP) \ldots, Apply(\ \Delta_n, t_n, MF)\ ;$$

Définition 6.5.3 (Mutation d'un modèle) *La mutation est une transformation d'arbre. Soit* a_1 *et* a_2 *deux arbres :*

Mut(a_1*, MF) =* a_2 *est possible en appliquant à l'arbre* a_1 *les mutations suivantes :*

1. *Apply(*Δ*changement-type,* a_1*, MF),* Δ*changement-type : la mutation d'un changement de type.*
2. *Apply(*Δ*perte-donnée,* a_1*, MF),*Δ*perte-donnée : la mutation d'une perte de donnée.*
3. *Apply(*Δ*corruption,* a_1*, MF) :*
   - Δ*corruption-reception : la mutation d'une corruption d'un paramètre d'un signal de réception.*
   - Δ*corruption-emission : la mutation d'une corruption d'un paramètre d'un signal d'émission.*
4. *Apply(*Δ*contrôle-externe,* m_1*, MF),* Δ*contrôle-externe : la mutation d'une fonction de contrôle externe.*
5. *Apply(*Δ*coupure,* m_1*, MF),*
   - Δ*coupure-reception : la mutation d'une coupure d'un signal de réception.*
   - Δ*coupure-emission : la mutation d'une coupure d'un signal d'émission.*
6. *Apply(*Δ*panne,* m_1*, MF),*
   - Δ*panne-reception : la mutation d'une panne d'un signal de réception.*
   - Δ*panne-emission : la mutation d'une panne d'un signal d'émission.*
   - Δ*panne-composant : la mutation d'une panne d'un composant.*

Les mutations présentes dans la définition se décomposent de la façon suivante. Nous ne détaillons pas toutes les mutations, et présentons les transformations de type d'Elargissement, de Dysfonctionnement, de Coupure et de Panne :

1. **Elargissement des entrées / Changement de types :**
   Δchangement-type(n) :
   - si n = var_declaration(x,T) et (x,T,T$'$) $\in$ dMF-changement-type :
     alors n = var_declaration(x, T$'$)
   - sinon n

2. **Dysfonctionnement :**

– Perte de donnée = Δperte-donnée(n) :
  – si n = canal_declaration($canal$, p_1 to p_2, T) et (p_1 to p_2,T,T$'$) ∈ dMF-perte-donnée :
    alors n = canal_declaration($canal$, p_1 to p_2, T$'$)
  – sinon n

– Corruption de réception = Δcorruption-reception(n) :
  – si n = reception(c,s,x) et (c,s,x,v) ∈ dMF-corruption-reception :
    alors n = reception(c,s,x) ; affectation(x,v)
  – sinon n

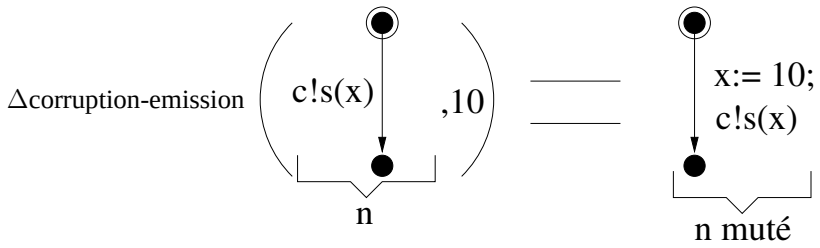

FIG. 6.12 – Mutation d'une émission.

– Fonction externe de contrôle = Δcorruption-controle(n) :
  – si n = garde([b]) et ([b],t) ∈ dMF-contrôle-externe :
    alors n = garde(t)
  – sinon n

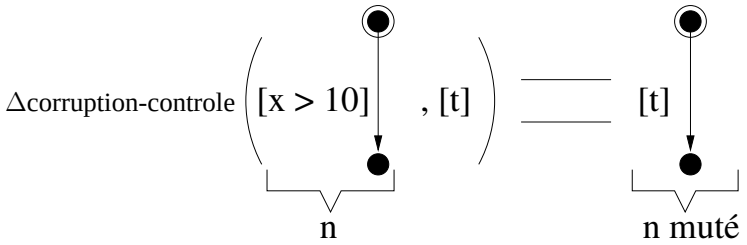

FIG. 6.13 – Mutation d'un noeud garde.

3. **Coupure :**
  – Coupure de réception = Δcoupure-reception(n) :
    – si n = reception(c,s,x) et (c,s,x,coupure,[a], [b]) ∈ dMF-panne-reception :
      alors n = [a]reception(c,s,x), [b]reception(coupure)
    – sinon n

4. **Pannes :**
   - Panne de réception = Δpanne-reception(n) :
     - si n = reception(c,s,x) et (c,s,x,panne) $\in$ dMF-panne-reception :
       alors n = reception(panne)
     - sinon n

   La mutation est effectuée à partir de l'état initial d'une transition. Le nouveau état créé doit être un état différent des états présents dans l'ensemble des états du processus et doit intégrer cet ensemble.

Nous proposons avec l'exemple suivant de muter une spécification en appliquant les régles de mutations proposées ci dessus :

Exemple 9 — **[Mutation d'une spécification S $\Longrightarrow$ spécification mutée S$_m$]**
Pour cet exemple nous proposons une spécification S, réalisant des actions visibles d'entrée et sortie sur le canal c avec, s le signal, a,b,x des variables, e,e' des expressions. À partir d'un modèle de faute MF (donné dans le tableau suivant), nous proposons une spécification mutée S$_m$ en appliquant les règles de la fonction de mutation : Apply (Δ, S, MF) = S$_m$.

Soit une spécification S :

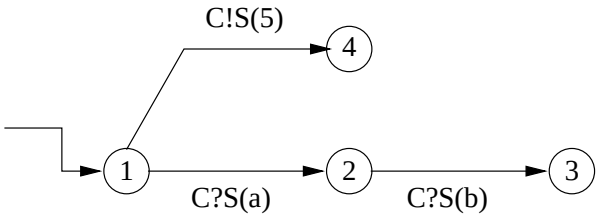

FIG. 6.14 – Une spécification S

Un modèle de faute MF :

Type de Faute	Ancien Nœud	Domaine de faute	Nouveau Nœud
corruption en émission	$1 \xrightarrow{c!s(e)} 4$	$\{(s(e),x+10)\}$	$1 \xrightarrow{c!s(x+10)} 4$
Panne d'un composant	2	$\{$(panne)$\}$	$2 \xrightarrow{?panne} 5$

Dans le tableau ci-dessus l'état 5 est un nouvel état non contenu dans l'IOLTS de la spécification initiale.

La spécification mutée S_m en appliquant les règles de mutation est :

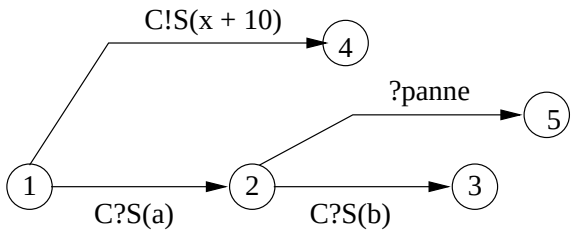

FIG. 6.15 – Spécification Mutée

6.6 Les propriétés de robustesse

Les propriétés ou contraintes de robustesse d'un système peuvent être définies comme des besoins de sûreté, de temps de réponse, d'espace mémoire, etc. Nous proposons que les propriétés de robustesse expriment des propriétés "non fonctionnelles" spécifiant comment un système doit se comporter pendant une exécution. Dans ce cas, une propriété de robustesse peut exprimer l'utilisation d'un composant ou d'une méthode particulière.

Dans le contexte du test de robustesse, une propriété représente soit une garantie d'obtenir un résultat, soit une condition transitoire pour retrouver une situation initiale par exemple :

- **Retour en situation initiale sans conséquences, après une situation d'erreur** $\Rightarrow$ Cette propriété permet un fonctionnement acceptable de l'implantation après une situation d'erreur reconnue. La propriété nécessite une connaissance des actions menant à une situation d'erreur et aux séquences sortantes de cette situation d'erreur.

NOTE :
De manière informelle, les propriétés de robustesse peuvent être écrites en langage naturel. Nous choisissons pour le test de robustesse de modéliser la négation de la propriété par un automate de Rabin paramétré (définition 5.3.4) définissant ainsi un observateur.

6.7 La relation de robustesse

Similaire à la relation de satisfaction (définition 5.4), la relation de robustesse est établie entre l'automate paramétré de Rabin représentant la négation de la propriété de robustesse *P* et l'IOLTS d'une implantation *IUT*. Elle vise à s'assurer qu'une implantation conserve un comportement acceptable (à une exécution nominale) vis à vis d'une propriété (à satisfaire). Nous formalisons la notion de robustesse d'une implantation préservant la condition de robustesse φ par :

Définition 6.7.1 (Relation de robustesse) *La relation de robustesse est une rélation de satisficabilité avec : IUT =* $(Q_{IUT},\ A_{IUT},\ T_{IUT},\ q_{IUT}^{init})$ *un IOLTS représentant une implantation,* $\mathcal{P}$ *une propriété de robutesse à tester sur IUT et* $\mathcal{O}bs = (\mathcal{O},\ \mathcal{T}_{\mathcal{O}},\ \mathcal{C}_{\mathcal{O}})$ *un observateur de* $\neg\mathcal{P}$ *représenté par un automate de Rabin paramétré tel que* $\mathcal{O} = (Q_{\mathcal{O}},\ A_{\mathcal{O}},\ T_{\mathcal{O}},\ q_{\mathcal{O}}^{init})$ *soit un IOLTS* déterministe, complet *et* $\mathcal{L}(\mathcal{O}bs)$ *et le langage reconnu par* $\mathcal{O}bs$*, avec* $A_{\mathcal{O}} \subseteq A_{IUT} \cup \{\delta\}$*. Nous disons que IUT est* robuste $\mathcal{P}$ *si et seulement si :*

$$\mathcal{L}(\Delta(IUT))\downarrow \mathcal{A}_{\mathcal{O}} \cap \mathcal{L}(\mathcal{O}bs) = \emptyset.$$

En d'autres termes, si IUT satisfait $\mathcal{P}$*, et si aucune trace de l'IUT n'est reconnue par l'observateur* $\mathcal{O}bs$*.*

6.8 Architecture de test, Graphe de test, Sélection et cas de test de robustesse

Nous ne développons pas maintenant en détail les étapes pour générer et exécuter les cas de test de robustesse. Puisque les étapes restent les mêmes que pour le test de propriétés et que nous allons les présenter plus en détails dans le chapitre 7. À partir de la spécification mutée et de l'observateur nous produisons un graphe de test, puis un cas de test (sélectionné dans le graphe de test), et finalement un testeur interagit avec l'implantation en fonction du cas de test donné.

Architecture de test : L'abstraction considérée est l'ensemble (A_c,A_u).

Graphe de test de robustesse : Le graphe de test de robustesse GT est le produit entre la spécification mutée et l'observateur (négation de la propriété de robustesse à tester) : GT = $S_m \otimes \mathcal{O}$bs décrit par la définition 5.6.1.

Sélection de test de robustesse : Comme pour le test de propriétés, les cas de test de robustesse sont extaits du graphe de test en applicant la fonction de sélection décrite dans la section 5.8.

Cas de test de robustesse : Par rapport au test de propriétés, les verdicts des cas de test deviennent : "**NonRobuste**" et "**Robuste**". Notre méthode permet de construire des cas de test vérifiant une propriété de la façon suivante :

– Si il existe un test sur une implantation tel que son exécution produit un verdict **Non Robuste**, cela veut dire que nous avons pu exécuter une séquence de l'ensemble $\mathcal{L}$(SMutée) $\cap$ $\mathcal{L}$(IUT) $\cap$ $\mathcal{L}(\neg P)$, à partir de la spécification mutée et de la négation de la propriété donnée (et l'implantation est donc non robuste définition6.7.1).

– Par contre, nous ne générons pas tous les cas de test permettant de vérifier la non robustesse d'une implantation vis-à-vis d'une propriété puisque la construction d'un cas de test est limitée par la spécification mutée. En particulier, les séquences de l'ensemble $\mathcal{L}$(IUT) $\cap$ $\mathcal{L}$(SMutée) - $\mathcal{L}(\neg P)$, n'appartiendront pas à l'ensemble des cas de test générés (bien qu'elles violent la propriété de robustesse).

Exécution des cas de test de robustesse : Les cas de test sont exécutés avec un testeur sur les implantations. Les implantations sont déclarées robustes ou non selon les verdicts obtenus.

6.9 Conclusion

Nous venons de présenter une méthode de test basée sur les modèles orientée test de robustesse. Nous avons défini une méthode complète pour générer et exécuter des cas de test de propriétés, puis, nous l'avons adaptée au test de robustesse. Nous avons proposé de tester des propriétés de sûreté et de vivacité bornée, modélisées par des automates de Rabin paramétrés. Nous avons pris en compte dans nos modèles d'éventuels aléas induits par un environnement ou un utilisateur. La condition de robustesse vise à s'assurer qu'une implantation conserve un comportement acceptable vis-à-vis d'une propriété de robustesse.

Nous allons voir, dans la section suivante, les différents algorithmes permettant de produire un graphe de test et de sélectionner un cas de test de robustesse.

Troisième partie

Un prototype pour générer et exécuter des cas de test de robustesse pour des programmes Java

Chapitre 7

Une plate-forme pour le test de robustesse

Ce chapitre présente une plate-forme pour le test de robustesse utilisant la théorie vue dans les chapitres précédents. La plate-forme propose une chaîne complète d'outils pour générer automatiquement des cas de test de robustesse CT, à partir d'une architecture de test AT, d'une spécification S, d'un modèle de faute MF et d'un observateur $\mathcal{O}$bs (la négation d'une propriété donnée). Deux étapes sont nécessaires pour obtenir un cas de test CT. Premièrement, nous construisons le graphe de test GT et deuxièmement, nous sélectionnons dans ce graphe un sous-graphe définissant un cas de test. Pour réaliser ce travail, nous détaillons dans ce chapitre les algorithmes de construction du graphe de test, et la sélection (dans le graphe) des cas de test.

7.1 Présentation Globale

La plate-forme (figure 7.1) **:**

1. Les données (les éléments "de couleur" présents sur la figure 7.1 sont les données initiales) :
   - Une architecture de test : (A_u, A_c).
   - Une spécification : S.
   - Un modèle de faute : MF (qui représente les fautes et les actions provenant de l'environnement des composants du système testé).
   - Une propriété = Un Observateur : $\mathcal{O}$bs (qui représente la négation de la propriété).

2. Les éléments produits (les éléments ovoïdes présents sur la figure 7.1 sont les données produites) :
   - Une spécification mutée : S_m.
   - Un IOLTS associé à la spécification mutée,déterminisé, minimal et comportant les blocages (IOLTS de suspension+δ) : $det(\Delta(S_m))_{mini}$.

– Un graphe de test : GT (IOLTS).
– Un cas de test : CT (IOLTS).

3. Les opérateurs (Les rectangles présents sur la figure 7.1 indiquent les différents opérateurs de composition) :
   – La mutation : Apply(Δ, S, MF) = S_m (Δ est un opérateur de mutation).
   – La simulation : Transformation du modèle d'automates communicants en modèle IOLTS.
   – La suspension + δ, la déterminisation, la minimisation.
   – Le produit : $det(\Delta(S_m)) \otimes \mathcal{O}$bs = GT.
   – La sélection : sélection(GT) = CT.

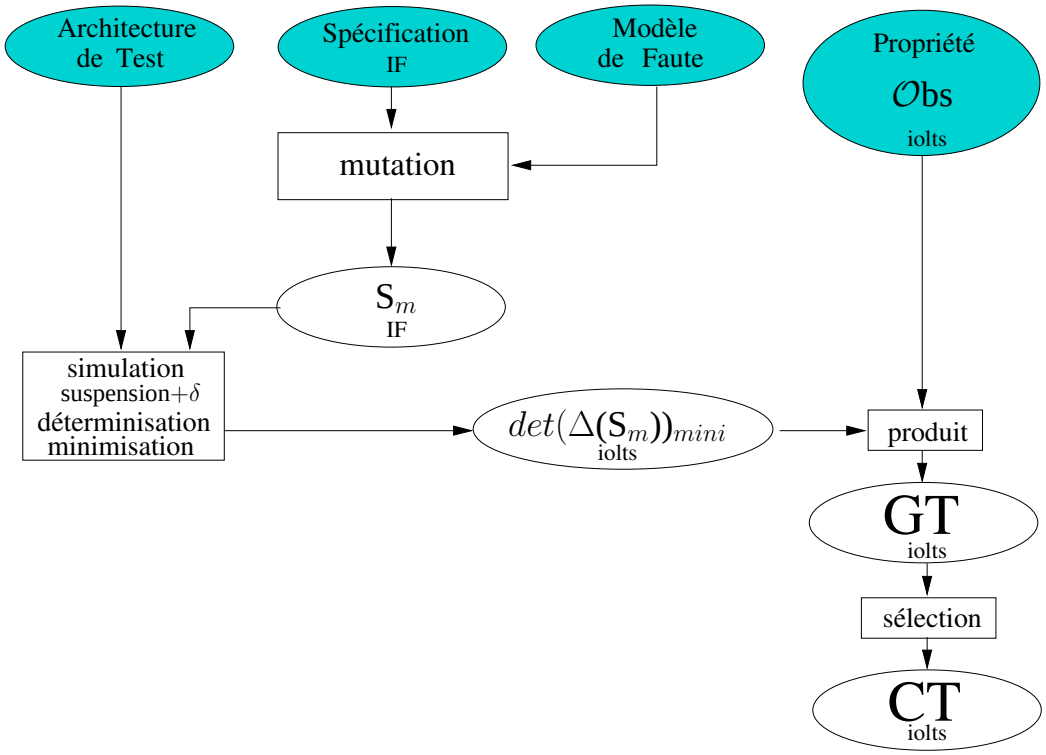

FIG. 7.1 – Plate-forme pour générer un cas de test de robustesse

Dans cette plate-forme, les algorithmes que nous utilisons sont, la simulation (boîte à outils IF[1]) la déterminisation et la minimisation (boîte à outils CADP[2]). Les autres algorithmes que nous avons

[1]Intermediate format
[2]Construction and Analysis of Distributed Processes, formely known as CAESAR/ALDEBARAN Development Package

développés sont : la suspension+δ, le produit, la sélection. Ces algorithmes ont été implémentés avec les langages C ou C++.

L'objectif de la plate-forme est de fournir un cas de test de robustesse exécutable par un testeur. Nous proposons de découper en trois travaux distincts la génération de cas de test de robustesse :

1. Le premier travail est d'obtenir une spécification mutée, déterminisée en tenant compte des blocages éventuels. Pour ce travail, nous procédons en plusieurs étapes. Il faut muter la spécification S en appliquant les opérateurs Δ de mutation (vus dans la section 6.5.4) avec le modèle de faute MF.
2. Le deuxième travail se réalise avec la spécification mutée S_m, et l'architecture de test. Trois phases sont nécessaires pour ce travail :
   (a) La première phase, "la simulation", permet d'obtenir le IOLTS associé à la spécification S_m (S_m initialement décrit par des automates étendus communicants). L'outil pour effectuer cette simulation est disponible dans la boîte à outils IF [BGM02].
   (b) La deuxième phase est d'appliquer l'opérateur de "suspension + δ" pour définir et représenter (sur le IOLTS S_m) les blocages éventuels (cet outil a été réalisé en partie en s'inspirant des travaux de [Mor00]. Cet outil, intégré dans un développement existant, a été réalisé en fonction de la structure de donnée, d'où sa réécriture en utilisant des algorithmes existants dans la littérature [Mor00, HU79, Tar72].
   (c) La troisième phase est de déterminiser ([HU79]) puis de minimiser (avec une relation de bisimulation forte) l'IOLTS $(\Delta(S_m))$ = det($\Delta(S_m)$) et minimal : det$(\Delta(S_m))_{mini}$, en utilisant l'outil "Aldebaran" disponible dans la boîte à outils CADP [3].
3. Le deuxième travail est de construire le graphe de test de robustesse GT, suivant les régles du produit synchrone étendu (décrit dans la section 5.8) entre le $det(\Delta(S_m))_{mini}$ et l'observateur $\mathcal{O}$bs : GT = $det(\Delta(S_m))_{mini} \otimes \mathcal{O}$bs.
4. Le dernier travail est d'extraire du graphe GT (par une méthode de sélection) un cas de test CT.

7.2 Mise en œuvre des algorithmes

Nous présentons maintenant, les algorithmes disponibles dans les boîtes à outils et détaillons nos algorithmes, de suspension+δ, de construction du graphe de test et de sélection d'un cas de test.

NOTE :
Pour la phase de mutation, il suffit d'appliquer la fonction Apply(Δ, S, MF) (vue et détaillée dans la section 6.5.4) sur la spécification S en relation avec le modèle de faute MF. Cette phase est en cours d'implémentation en collaboration avec un étudiant et les langages de programmation utilisés sont Lex, Yacc et C++.

[3] Caesar Aldebaran Development Package

7.2.1 Étapes de construction d'un det($\Delta(S_m)$)$_{mini}$

La figure 7.2 présente l'enchaînement des opérations pour obtenir un det($\Delta(S_m)$) minimal : det($\Delta(S_m)$)$_{mini}$.

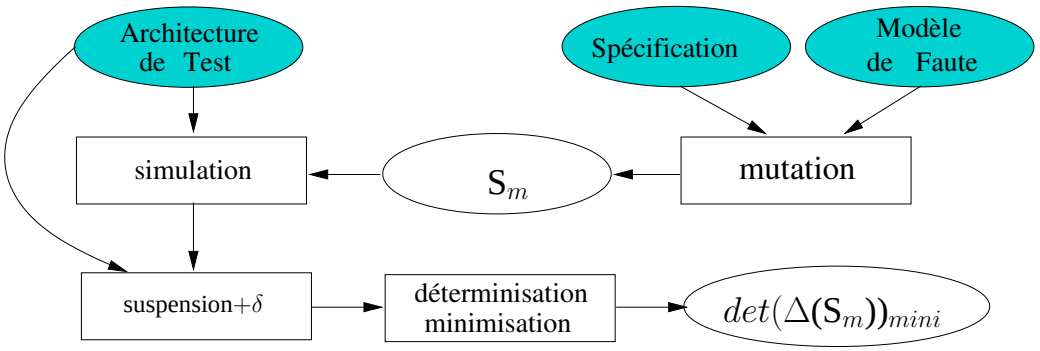

FIG. 7.2 – Chaîne d'opérations pour obtenir : det($\Delta(S)$)$_{mini}$ = une spécification mutée, déterministe, minimale et prenant en compte les blocages

Phase de simulation [BGM02]

L'algorithme de simulation est disponible dans la boîte à outils IF. Il permet d'obtenir un IOLTS à partir d'un programme IF (application des règles de sémantique vues au premier chapitre). Nous verrons dans la partie III comment décrire une spécification dans le langage IF.

ALGORITHME 1 : Simulation

La fonction **simulation(S,AT)** transforme une spécification S, décrit sous forme d'un modèle d'automates étendus communicants en un IOLTS.

– Entrées : Une spécification S (automates étendus communicants), une architecture de test AT.
– Sortie : Un IOLTS.

Phase de suspension+δ d'une spécification [Mor00]

La fonction de suspension+δ va nous permettre de définir les actions de blocages. Cette fonction a pour but de fournir un IOLTS équivalent au sens des traces au IOLTS obtenu à partir de la spécification mutée, auquel nous ajoutons les différents types de blocages (identifiés par une ac-

tion !δ). Un état du IOLTS produit est un ensemble d'états obtenu par τ-fermetures. La τ-fermeture correspond à l'ϵ-clôture pour les automates finis [HU79].

L'algorithme choisi est un parcours en profondeur d'un IOLTS, à partir de son état initial. La terminaison de l'algorithme est assurée lorsque tous les états du IOLTS ont tous été visités. L'algorithme de [Tar72] se prête tout à fait à ce calcul. L'information de livelock est attribuée à toutes les composantes fortement connexes non triviales, composées d'états par des τ-transitions (action τ au sens du IOLTS). Les blocages de sortie correspondent à des composantes fortement connexes triviales (des singletons d'états) qui n'ont aucune composante fille et qui ne possèdent pas de transitions tirables internes ou d'émissions sur l'alphabet de la spécification mutée. Un deadlock est le raffinement d'un blocage de sortie où aucune action observable n'est tirable. Tous les blocages ont une importance dans la construction du cas de test. Les livelock sont, *a priori*, des états terminaux de la spécification et le test doit se terminer après l'expiration d'une temporisation armée au début d'une exécution d'un cas de test. Les livelock n'impliquent pas un arrêt complet des émissions de l'implantation, mais rien ne permet de connaître la durée de ce blocage.

ALGORITHME 2 : Suspension + δ

La fonction **suspension+δ(S, AT)** applique un algorithme de réduction sur un S = $(Q_S,\ A_S,\ T_S,\ q_S^{init})$ et ajout d'action de blocage δ de la façon suivante :

1. $\forall$ q $\in$ Q_S tel que q $\not\longrightarrow$, alors nous ajoutons la transition δ tel que q $\xrightarrow{\delta}$ q.
2. $\forall$ q $\in$ Q_S si q $\stackrel{\tau^+}{\Longrightarrow}$ q, suppression des τ-transitions et nous ajoutons la transition δ tel que q $\xrightarrow{\delta}$ q.
3. $\forall$ q et q$'$ $\in$ Q_S et a $\in$ A_S (et a $\in$ AT) si q $\stackrel{\tau^+a}{\Longrightarrow}$ q$'$, alors nous la remplaçons par une transition q $\xrightarrow{a}$ q$'$.

– Entrées : Un IOLTS S, une architecture de test AT.
– Sortie : Un IOLTS τ^+a = $(Q_{\tau^+a},\ A_{\tau^+a},\ T_{\tau^+a},\ q_{\tau^+a}^{init})$ "réduit" et comportant éventuellement des boucles de δ.

Principe : L'algorithme de "suspension + δ" transforme l'IOLTS donné en un IOLTS sans actions internes. l'algorithme garde suffisamment d'information sur chaque état visité pour éviter de traverser plusieurs fois les mêmes parties du IOLTS et permettre de remplacer par des δ les éventuels blocages provoqués par les actions internes. Il est noté que les états puits du IOLTS initial comporteront des actions de δ.

L'IOLTS est obtenu par les transformations suivantes :
– $q_{\tau^+a}^{init}$ = $\tau^+a(q_S^{init})$ et q_{τ^+a} = $\tau^+a(q_S)$ | q_S^{init} et $q_S \in Q_S$, représentent les états $\in Q_{\tau^+a}$ du IOLTS τ^+a par la τ^+a-fermeture de l'ensemble Q_S du IOLTS initial S.

– Les boucles de τ (correspondant à des blocages) sont réduits sous la forme d'une action δ appartenant à l'ensemble des actions A_{τ^+a} du IOLTS τ^+a . $A_{\tau^+a} = A_S \cup \delta$, signifie que l'ensemble des actions de A_{τ^+a} est composé de l'ensemble d'actions de A_S augmenté de l'action δ de blocage.

– $\forall a \in A_{\tau^+a}, \forall E, E' \in Q_S, E \xrightarrow{a}_{\tau^+a} E' \Leftrightarrow \exists q \in E, \exists q' \in Q, q \xrightarrow{a} q'$ et $E' = \tau^+a(q')$, signifiant que tout état successeur E' d'un état E du IOLTS réduit est la τ^+a-fermeture d'un état successeur à un état E dans le IOLTS S initial.

Phase de Déterminisation de $\Delta(S_m)$ {aldebaran [Fer88]}

Le fait de supprimer certaines actions internes τ peut introduire du non détermisme dans les IOLTS. Nous déterminisons les $\Delta(S_m)$ suivant la définition 3.4.1 :

ALGORITHME 3 : Déterminisation

La fonction **déterminisation($\Delta(S_m)$)** supprime le non déterminisme observable. Le principe de la déterminisation a été vu dans la section 3.4.1.

– Entrée : Un IOLTS $\Delta(S_m)$.
– Sortie : Un IOLTS déterminisé det($\Delta(S_m)$).

Phase de minimisation de det($\Delta(S_m)$) {**aldebaran [Fer88]**}

La phase de minimisation (en appliquant les règles de bisimulation forte) n'est pas indispensable, souvent la déterminisation produit un automate déterministe minimal, mais cette phase peut dans certains cas réduire la taille du IOLTS donné. La minimisation permet de réduire les dépliages redondants pouvant être crées après la τ-réduction et/ou la déterminisation du IOLTS.

ALGORITHME 4 : Minimisation

La fonction **minimisation(det($\Delta(S_m)$))** :
Si le IOLTS det($\Delta(S_m)$) reconnaît une langage A, le IOLTS résultant de la minimisation est l'IOLTS det($\Delta(S_m)_{mini}$ = ($Q_{\det(\Delta(S_m)_{mini}}$, $A_{\det(\Delta(S_m)_{mini}}$, $T_{\det(\Delta(S_m)_{mini}}$, $q^{init}_{\det(\Delta(S_m)_{mini}}$) reconnaissant le même langage A.

– Entrée : Un IOLTS det($\Delta(S_m)$).
– Sortie : Un IOLTS det($\Delta(S_m)_{mini}$.

7.2.2 Construction d'un graphe de test de robustesse : GT

Le graphe de test est obtenu en effectuant un produit synchrone étendu (en appliquant les règles de construction définies dans la définition 5.6.1) entre un IOLTS (spécification mutée suspendue déterminisée minimale = det(Δ(S$_m$)$_{mini}$) = det(suspension+δ(S$_m$,AT))$_{mini}$ avec AT = $A_c \cup A_u$ une architecture de test donnée) et un automate de Rabin paramétré complet (l'observateur repésentant la négation de la propriété de robustesse à tester) : GT = det(Δ(S$_m$))$_{mini} \otimes \mathcal{O}$bs.

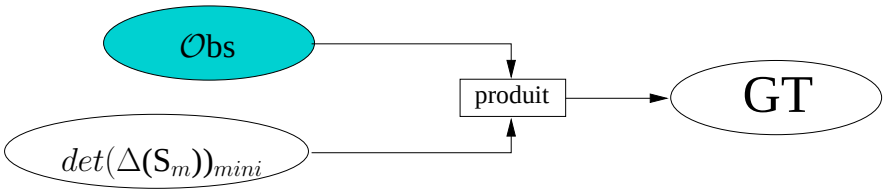

FIG. 7.3 – Graphe de test de robustesse obtenu par produit synchrone étendu entre un observateur et l'IOLTS d'une spécification mutée (déterminisé, minimale avec détection des blocages).

L'algorithme proposé est un parcours en profondeur avec pour données initiales :
– $\mathcal{O}$bs = $\{\mathcal{O},\ \mathcal{T}_\mathcal{O},\ \mathcal{C}_\mathcal{O}\}$ avec $\mathcal{O}$ = $(Q_\mathcal{O},\ A_\mathcal{O},\ T_\mathcal{O},\ q_\mathcal{O}^{init})$ et $\mathcal{T}_\mathcal{O}$ = le marquage des ensembles d'états distingués U et L $\in Q_O$, et $\mathcal{C}_\mathcal{O}$ = les couples de paramètres.
– det(Δ(S$_m$))$_{mini}$ = $(Q_{\text{det}(\Delta(S_m))_{mini}},\ A_{\text{det}(\Delta(S_m))_{mini}},\ T_{\text{det}(\Delta(S_m))_{mini}},\ q^{init}_{\text{det}(\Delta(S_m))_{mini}})$.

Le graphe produit est GT = $(Q_{GT},\ A_{GT},\ T_{GT},\ q_{GT}^{init})$ tel que :
– L'ensemble des actions A$_{GT} \subseteq$ A$_O$.
– Un état $q \in$ Q$_{GT}$ est un couple (état de la spécification, état de l'observateur).
– L'état initial de GT est : q_{GT}^{init} = $(q^{init}_{det(\Delta(S_m))_{mini}}, q^{init}_{Obs})$.
– L'ensemble Q$_{GT}$ dépend de l'ensemble T$_{GT}$ construit lors du produit, plusieurs cas sont possibles pour obtenir une action $a \in$ A$_{GT}$, une transition $\longrightarrow \in$ T$_{GT}$ et $q \in$ Q$_{GT}$:

1. Si q = (S,O) $\in$ Q$_{GT}$ est un état du graphe de test et a une action telle que : $a \in$ A$_{det(\Delta(S_m))_{mini}}$ et si S$'$ $\in$ Q$_{det(\Delta(S_m))_{mini}}$ est un état de la spécification tel que S $\xrightarrow{a}$ S$'$, alors $\exists\ a \in$ A$_{Obs}$ une action (observateur complet) et O$'$ $\in$ Q$_{Obs}$ tel que O $\xrightarrow{a}$ O$'$. Alors q' = (S$'$,O$'$) $\in$ Q$_{GT}$ est un état du graphe, $a \in$ A$_{GT}$ est une action du graphe et $\xrightarrow{a} \in$ T$_{GT}$ est une transition du graphe avec : $q \xrightarrow{a} q'$.

   À partir du graphe produit avec les régles R0 et R1 selon la définition 5.6.1, nous rajoutons les extensions si nécessaires avec en fonction des actions d'entrée ou de sortie :

2. Si q = (S,O) $\in$ Q$_{GT}$ est un état du graphe, $a \in$ A$_c$ est une action d'entrée telle que S $\not\xrightarrow{a}$ et il n'existe pas de séquence σ de S issue de q tel que $a \in$ Trace(σ) alors, q' = (S,O$'$) $\in$ Q$_{GT}$ est un état du graphe, $a \in$ A$_{GT}$ est une action du graphe et $\xrightarrow{a} \in$ T$_{GT}$ est une transition du graphe avec : $q \xrightarrow{a} q'$. Nous synthétisons d'abord cette information sur la spécification.

3. Si q = (S,O) $\in$ Q$_{GT}$ est un état du graphe, et si a est une action de sortie $\in$ A$_u$ et non présente

dans la spécification, alors $q^{'}$ = (S,O$^{'}$) $\in$ Q_{GT} devient un état du graphe, $a \in A_{GT}$ devient une action du graphe et $\xrightarrow{} \in T_{GT}$ devient une transition du graphe avec : $q \xrightarrow{a} q^{'}$.

Le graphe de test GT obtenu contient l'IOLTS associé à la spécification mutée étendu des actions de la négation de la propriété (si celles-ci ne faisaient pas parties des actions du IOLTS de la spécification).

NOTE :
Nous rappelons que dans cet algorithme l'IOLTS det($\Delta(S_m)$)$_{mini}$ sert de guide de construction et l'observateur sert à placer les verdicts. En effet, dès que le graphe GT est construit les états $q \in Q_{GT}$ comportant des états de l'observateur $\in Q_{Obs}$ marqués U ou L sont des états distingués par le même marquage (marquage servant à placer les verdicts).

Remarque : Dans les algorithmes suivants :
- *Les structures de données et les déclarations de variables sont effectuées en début du programme.*
- *Les commentaires sont encadrés par les symboles /* */.*
- *Les fonctions appelées ne sont pas toutes décrites. Seules les fonctions non explicites et nécessitant des explications sont écrites.*

ALGORITHME 5 : Génération d'un graphe de test

La fonction **produit(S, $\mathcal{O}$bs)** construit un graphe de test GT (automate de Rabin paramétré). Les états de GT sont des couples (état$_S$, état$_O$). Les transitions du graphe GT sont obtenues en appliquant les règles du produit synchrone étendu (définit par la définition 5.6.1). Les états de GT sont marqués (L ou U ou pas de marquage) suivant le marquage de l'état$_O$.

- Entrées : Un IOLTS S, un automate de Rabin paramétré $\mathcal{O}$bs,
- Sorties : Un automate de Rabin paramétré GT qui est le résulat du produit synchrone étendu entre S et $\mathcal{O}$bs. GT est marqué des états distingués état_U et état_L selon les états de $\mathcal{O}$bs.

Principe : L'algorithme de **produit(S, $\mathcal{O}$bs)** est un produit synchrone étendu entre la spécification et l'observateur en appliquant un parcours en profondeur après synthèse des actions contrôlables de la spécification.

---

visited	/* ensemble des états visités */
T_{GT}	/* ensemble de transitions */
Q_{GT}	/* ensemble d'états */
$\mathcal{T}_{GT}$	/* ensemble des états marqués */

controlable(q) $\longrightarrow 2^{A_c}$ fonction qui, pour chaque état, donne l'ensemble des actions contrôlables, exécutables, soit dans l'état q, soit dans un état atteignable à partir de q

/* initialisation des données */
visited $\longleftarrow \emptyset$;
$T_{GT} \longleftarrow \emptyset$;
$Q_{GT} \longleftarrow \emptyset$;
$\mathcal{T}_{GT} \longleftarrow \emptyset$;

synthèse($etat_S^{init}$) ;
visited $\longleftarrow \emptyset$;
produit-DFS($etat_S^{init}$, $etat_O^{init}$) ;

ALGORITHME 6 : produit-DFS

La fonction **produit-DFS($etat_S$, $etat_O$)** est une fonction récurvive construisant les ensembles de transitions, d'états et des marquages du graphe de test.

- Entrées : Un couple d'état
- Sorties : L'ensemble des transitions sortantes de ($etat_S$, $etat_O$), l'ensemble des états atteints par les transitions et l'ensemble des marqages des états.

visited $\longleftarrow$ visited $\cup \{(etat_S, etat_O)\}$;
forall (a,$etat'_S$) **such that** $etat_S \xrightarrow{a} etat'_S$ **do**
  Let $etat'_O$ **such that** $etat_0 \xrightarrow{a} etat'_0$ **then**
    $T_{GT} \longleftarrow T_{GT} \cup \{ (etat_S, etat_0) \xrightarrow{a} (etat'_S, etat'_0)\}$;
    $Q_{GT} \longleftarrow Q_{GT} \cup \{ (etat_S, etat_0)\}$;
    if ($etat_0 \in \mathcal{T}_O$ **) then**
      $\mathcal{T}_{GT} \longleftarrow \mathcal{T}_{GT} \cup \{ (etat_S, etat_0) \}$;
    end_if ;
  if ($(etat'_S, etat'_O) \notin$ visited) **) then**
    produit-DFS($etat'_S$, $etat'_O$) ;
  end_if ;
end_forall ;
forall (a,$etat'_0$) **such that** $etat_0 \xrightarrow{a} etat'_0$ **&&** $a \notin$ controlable ($etat_S$) **do**
  $T_{GT} \longleftarrow T_{GT} \cup \{ (etat_S, etat_0) \xrightarrow{a} (etat_S, etat'_0)\}$;
  $Q_{GT} \longleftarrow Q_{GT} \cup \{ (etat_S, etat_0)\}$;
  if ($etat_0 \in \mathcal{T}_O$ **) then**
    $\mathcal{T}_{GT} \longleftarrow \mathcal{T}_{GT} \cup \{ (etat_S, etat_0)\}$;
  end_if ;
  if ($(etat_S, etat'_O) \notin$ visited) **) then**
    produit-DFS($etat_S$, $etat'_O$)} ;
  end_if ;
end_forall ;

Nous détaillons à partir de cet algorithme 5 la fonction de **synthèse(etat$_S$)**.

ALGORITHME 7 : Synthèse de la spécification

La fonction **synthèse(etat$_S$)** permet de synthétiser à chaque état de la spécification S les actions contrôlables.
– Entrée : Un IOLTS
– Sortie : Un IOLTS en associant aux états la synthèse des états contrôlable.

Principe : L'algorithme de **synthèse(etat$_S$)** est un parcours en profondeur du IOLTS donné. Il s'agit d'associer à chaque état du IOLTS des actions contrôlables exécutables soit dans cet état soit dans un état atteignable à partir de cet état.

visited /* ensemble des états visités */
A /* ensemble des synthèses */

/* initialisation des données */
visited $\longleftarrow \emptyset$ **;**
A $\longleftarrow \emptyset$ **;**
/* appel de la fonction de construction des synthèses */
controlable(etat$_S^{init}$) $\longleftarrow$ synthèse-DFS(etat$_S^{init}$) **;**

ALGORITHME 8 : Synthèse-DFS

La fonction **synthèse-DFS(etat$_S$)** permet de synthétiser à chaque état etat$_S$ un ensemble d'actions contrôlables.
– Entrée : Un état etat$_S$
– Sortie : Un ensemble d'actions contrôlables associé à etat$_S$.

visited $\longleftarrow$ visited $\cup$ {etat$_S$} **;**
A $\longleftarrow$ A$_c$ $\cap$ Act(etat$_S$) **;**
forall (a,etat$_S^{'}$) **such that** etat$_S$ $\xrightarrow{a}$ etat$_S^{'}$ **do**
  if ({etat$_S^{'}$} $\notin$ visited) **) then**
    A $\longleftarrow$ A $\cup$ synthèse-DFS(etat$_S^{'}$) **;**
  end_if ;
end_forall ;
controlable(etat$_S$) $\longleftarrow$ A **;**
return A **;**

À partir du graphe de test GT, nous devons maintenant sélectionner les cas de test.

7.2.3 Algorithme de sélection de cas de test

Parmi tous les chemins possibles obtenus grâce au produit synchrone étendu, nous voulons sélectionner les chemins comportant des boucles état_L. Nous définissons par boucle état_L (resp. boucle état_U) toutes les composantes connexes non triviales contenues dans un graphe comportant au moins un état distingué L $\in$ état_L (resp. U $\in$ état_U) et ne comportant pas d'état distingué U $\in$ état_U (resp. L $\in$ état_L). Nous rappelons que les états distingués L $\in$ état_L et U $\in$ état_U sont les marquages des états de l'automate de Rabin paramétré (définition 5.3.4).

L'algorithme 10 de sélection d'un cas de test (CT) se déroule en trois phases :
- La première phase consiste à isoler des chemins élémentaires contenant des état_U (sans boucles état_U).
- La deuxième phase consiste à rechercher les boucles état_L.
- La troisième phase construit le cas de test. Un cas de test est l'union des chemins (contenant ou non des état_U) menant à une boucle état_L plus la boucle état_L. CT = $\bigcup$ chemin $\cup$ composante_état_L.

Nous proposons un algorithme basé sur l'algorithme de Wolper [CVWY92] (pour retrouver les boucles état_L) et l'algorithme de Jéron [Jér91] pour éviter les boucles état_U. L'intérêt pratique de ces algorithmes est leur faible coût en mémoire. Ils utilisent chacun des piles (récursivité), dont les tailles maximales sont égales au nombre d'états atteignables du graphe de test (nécessaire pour tout parcours en profondeur et la longueur de la plus longue séquence acyclique est suffisante).

ALGORITHME 9 : Wolper[CVWY92]

L'algorithme (Wolper [CVWY92]) **composante(G)** permet à partir d'un graphe G d'isoler une composante fortement connexe de ce graphe.

- Entrée : Un graphe G (automate de Büchi).
- Sortie : Une composante fortement connexe contenant au moins un état distingué (état infiniment répété de l'automate de Büchi) si elle existe, sinon rien.

L'algorithme de [CVWY92] prend en entrée un automate de Büchi G = (Q, Σ, T, q_G^{init},etat_F) avec :

- Q est l'ensemble des états,
- Σ l'alphabet des actions du graphe,
- T est la fonction de transition : Q $\times$ Σ $\longrightarrow$ 2^Q,
- q_G^{init} est l'état initial,
- etat_F est l'ensemble des états distingués infiniment répétés, etat_F $\subseteq$ Q,

Le principe de l'algorithme est de trouver en deux parcours en profondeur, un chemin dans l'automate d'entrée comportant une composante fortement connexe non triviale (composante contenant au moins un état distingué $\in$ etat_F).

Les données de l'algorithme sont les suivantes :

/* P_1 est la pile initiale pour le premier parcours concernant la recherche des états $f \in$ état_F postfixés */
/* P_2 est la pile pour la recherche d'une composante fortement connexe */
/* DéjàVisitéM_1 est l'ensemble des marquages des états visités lors du premier parcours */
/* DéjàVisitéM_2 est l'ensemble des marquages des états visités lors du deuxième parcours */

$P_1 \leftarrow \emptyset$;
$P_2 \leftarrow \emptyset$;
DéjàVisité$M_1 \leftarrow \emptyset$
DéjàVisité$M_2 \leftarrow \emptyset$
empile(S_{init},P_1) ;
while $P_1 \neq \emptyset$ **do**
  x $\leftarrow$ recupere_elt(P_1) ;
  if $\exists$ y $\in$ succ(x) avec DéjàVisitéM_1(y) = 0 **then**
    y $\leftarrow$ extrait_un_successeur_non_visite(v) ;
    DéjàVisitéM_1(y) $\leftarrow$ 1 ;

empile(y,P_1) ;

  end_if ;
  else
    depile(x,P_1) ;
    if x $\in$ état_F **then**
    empile(x,P_2) ;
    while $P_2 \neq \emptyset$ **do**
      v $\leftarrow$ recupere_elt(P_2) ;
      if x $\in$ succ(v) **then** STOP ;**end_if ;**
      if DéjàVisitéM_2(w) = 1 $\forall$ w $\in$ succ(v) **then**
        depile(v,P_2) ;
      else
        w $\leftarrow$ extrait_un_successeur_non_visite(v) ;
        DéjàVisitéM_2(w) = 1 ;

empile(w,P_2) ;

      end_if ;
    end_while ;
  end_if ;
end_while ;

Cet algorithme est composé de 2 parcours :

1. Le premier détermine l'ensemble $\{f_1, f_2, \ldots, f_n\} \in$ Q, c'est à dire les états de etat_F = $\{f_1, f_2, \ldots, f_n\}$ qui sont atteignables à partir de $q_G^{init} \in$ Q par ordre postfixé.
2. Le deuxième parcourt les éléments de etat_F dans l'ordre postfixé, les place dans une pile P2 et s'arrête dès qu'un cycle est trouvé. C'est à dire dès que l'état initial f de pile P2 est atteint.

Deux critères d'arrêts sont possibles :

1. Tous les parcours sont finis, tous les marquages sont faits, les piles de construction sont vides, et il n'y a aucune de composante fortement connexe dans l'automate.
2. Soit l'algorithme trouve une composante fortement connexe non triviale, c'est à dire si en partant d'un état $f_i \in$ etat_F, d'une paire f_i, f_j, avec $i < j$ il y a un chemin de f_j à f_i, alors l'état f_i est contenu dans une composante fortement connexe non triviale.

Considérons le premier parcours où : Soit A un ensemble des états antérieurs (selon l'ordre postfixé) à f_i, soit B l'ensemble des états constitués des états f_i, et C l'ensemble des états après f_i. Si il existe un chemin venant de f_i (un état de B) à f_j (un état de C avec $j > i$) alors le chemin doit passer par des états de A. Et donc, f_i peut être atteint par un état de A et donc appartenir à une composante fortement connexe.

Considérons le deuxième parcours où : Une composante fortement connexe contenant f_j est trouvée, implique alors qu'un arbre de racine f_j avec une transition de retour sur f_j est construit (donc un arbre contenant un cycle). Soit $f_j \in$ etat_F, l'état le plus petit de tous les états de la composante fortement connexe, et considérons p le chemin de f_j à f_j. Nous supposons que les états de p sont atteignables à partir de f_i avec $i < j$. Si f_i est dans p alors f_j est atteint en partant de f_i, ce qui serait contradictoire avec le critère d'ordre fixé. Donc, pour le deuxième parcours aucun état de p est placé quand l'algorithme empile f_j dans P2 et donc ce parcours a pour but de trouver uniquement la transition amenant à f_j et donc de trouver une composante fortement connexe.

La figure 7.4 représente la détection d'une composante fortement connexe non triviale à partir de l'algorithme [CVWY92] contenant l'état f_i. Le premier parcours contenu dans [CVWY92] range tous les états de la composante fortement connexe de f_i en ordre postfixé dans un ensemble etat_F à partir de l'état initial. Supposons maintenant que dans la représentation 7.4, f_i soit le plus petit état de l'ensemble etat_F en ordre postfixé et qu'il appartient à la composante fortement connexe non triviale, alors les états f_j plus petits que f_i ne peuvent pas appartenir à la composante connexe de f_i, car si f_i avait été atteignable à partir de f_j alors $f_j \in$ etat_F et il serait placé avant f_i. Si il existe un chemin entre f_j et f_i en ordre postfixé alors cet état f_j appartiendrait à la composante fortement connexe de f_i, ce qui est contradictoire.

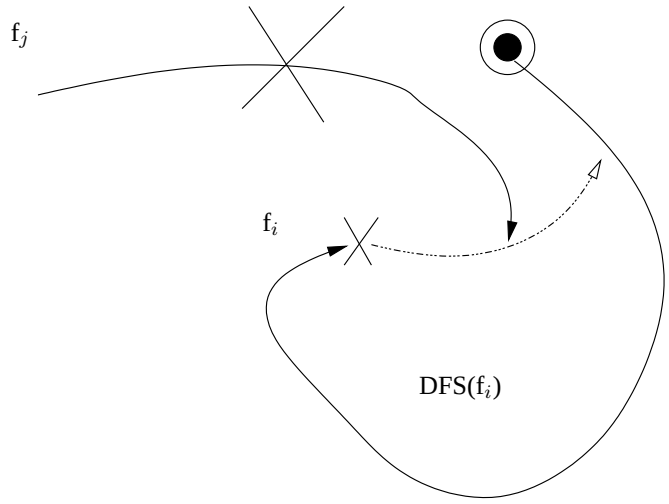

FIG. 7.4 – Détection d'une composante connexe non triviale.

Dans son principe notre algorithme 10 est une adaptation de l'algorithme 8 [CVWY92] et de l'algorithme de [Jér91]. Le graphe que nous utilisons en entrée est un automate de Rabin paramétré, il est composé comme l'automate de Büchi d'états infiniment répétés $\in$ état_L, mais aussi d'états répétés un nombre fini de fois $\in$ état_U. Ce que nous voulons, c'est construire un sous-graphe de l'automate de Rabin paramétré (cas de test CT) CT = chemin $\cup$ composante_état_L ou *chemin* ne comporte pas de boucles d'état_U et que *composante_état_L* soit une boucle contenant au moins un état distingué $\in$ état_L et ne contenant pas d'état $\in$ état_U. Si nous adaptons l'algorithme 8 pour détecter les boucles état_L, il nous faut en même temps isoler les états_U pour les soustraire à d'éventuelles boucles. Nous choisissons d'adapter l'algorithme de [Jér91] qui permet d'isoler les états distingués (et dans notre cas les états $\in$ état_U) dans un ensemble Frontière.

Le principe de l'algorithme 10 est le suivant :

1. L'ensemble pour garder les état_U est nommé Frontière. Les état_U sont isolés dans cet ensemble dès leur détection grâce à un parcours en profondeur du graphe. Le chemin de parcours est gardé dans une pile P1. Initialement le premier élément de l'ensemble Frontière est toujours le premier état du graphe GT, et le premier élément de la Pile P1 est toujours le premier élément de Frontière. L'ensemble Frontière est au fur et à mesure complété dès la détection d'un état_U (l'état_U détecté n'est pas empilé dans P1).
2. Pour tout état_U isolé le chemin courant du graphe GT menant à cet état est gardé.
3. Un deuxième parcours en profondeur permet de trouver les boucles état_L(sur le principe de l'algorithme 8). Ce parcours est effectué dès qu'un état_L est détecté en dépilant P1. Cet élément est initialement placé en sommet d'une pile P2 (élément racine de la boucle). Le nouveau parcours du graphe est gardé par la pile P2 jusqu'à la détection de état_L initial (si cela est possible) ou tant que tous les états atteignables à partir des états empilés ne sont pas tous visités :
   (a) Si une boucle état_L est trouvée l'exécution s'arrête (c'est dire si le parcours retourne sur l'état_L présent en début de pile P2). Le résultat produit est un cas de test composé

du chemin menant à la boucle état_L et la boucle d'état_L CT = chemin ∪ composante_état_L ou *chemin*.

(b) Si aucune boucle n'est trouvée les parcours s'arrêtent dés que tous les états sont visités. Le résultat ne produit aucun cas de test.

NOTE :
Avec l'algorithme 10 présenté, toutes les étapes de parcours se font en une seule exécution. La première boucle état_L détectée stoppe le programme. Pour nous permettre de trouver tous les cas de test comportant toutes les boucles état_L, il faut contrôler les informations de marquage des état_L (données par le produit). En effet, si nous voulons détecter une autre boucle état_L, il suffit d'isoler le premier marquage état_L qui a permis de détecter la boucle état_L, et ainsi de suite pour tous les état_L présents dans le graphe de test GT. Dans le programme, une structure indépendante des état_L du graphe GT gère le marquage et donc un utilisateur peut en modifiant les valeurs obtenir tous les cas de test (toutes les boucles_L). Ce choix a été fait pour limiter la taille des données, des structures de données rentrées et gardées, et le nombre de parcours du graphe.

ALGORITHME 10 : Sélection d'un cas de test de robutesse CT à partir du graphe de test GT

La fonction **selection(GT)** extrait du graphe GT un cas de test, si il existe. Un cas de test comporte au moins une boucle état_L et ne comporte pas de boucle état_U.

- Entrée : Un graphe de test GT (automate de Rabin paramétré).
- Sortie : Un cas de test CT (sous-graphe de GT) contenant une boucle état_L, si il existe, sinon rien.

/* P_1 est la pile initiale pour le premier parcours concernant la recherche des état_U */
/* P_2 est la pile pour la recherche des boucles état_L */
/* DéjàVisitéM_1 est le marquage des états visités lors du premier parcours de recherche des état_U pour chaque q*/
/* DéjàVisitéM_2 est le marquage des états visités lors du parcours de recherche des boucles état_L pour chaque q*/
/* table$_{sequence}$ est un tableau de chemin du graphe GT */
/* Frontiere est un tableau d'états */
/* S_0 est le premier état du graphe GT */
/* S_{init},x ,y, v, w sont les éléments de Pile (P1 ou P2)*/

DéjàVisitéM_1 ⟵ ∅ /* tous les marquages sont vides */
DéjàVisitéM_2 ⟵ 0
Frontiere ← S_0 /* initialisation de l'ensemble Frontière par le premier état du graphe, un CT commence toujours par l'état initial du graphe */

```
while Frontiere ≠ ∅ ;
    P1 ← ∅ ;                                          /* Initialisation des Piles */
    P2 ← ∅ ;
    Sinit ← extrait_un_elt(Frontiere) ;
    empile(Sinit,P1) ;                                /* Premier élément de Pile P1 */

    while P1 ≠ ∅ do
        x ← recupere_elt(P1) ;
        if ∃y ∈ succ(x) avec DéjàVisitéM1(y)=0 then   /* y visité sucesseur de x */
            y ← extrait_un_successeur_non_visite(x) ;
            DéjàVisitéM1(y) ← 1 ;
            if y ∈ etat_U
                Frontiere ← Frontiere ∪ {y}
                garder_sequence_menant_a_y(P1,y,tablesequence) ;
            else
                empiler(y,P1) ;
            end_if
        else
            depiler(x,P1) ;
            if x ∈ etat_L then
                empile(x,P2) ;                        /* Premier élément de Pile P2 */

                while P2 ≠ ∅ do                       /* Recherche des boucles état_L */
                    v ← recupere_elt(P2) ;
                    if x ∈ succ(v) then
                        return sequence_prefixe_cycle(tablesequence) ∪ cycle_L ;   /* Cas de test */
                        stop ; end_if
                    if DéjàVisitéM2(w) = 1 ∀ w ∈ succ(v) then
                        depile(v,P2) ;
                    else
                        w ← extrait_un_successeur_non_visite(v) ;   /* w sucesseur de v */
                        DéjàVisitéM2(w) ← 1 ;
                        if w ∉ etat_U
                            empiler(w,P2) ;
                        end_if
                    end_if
                end_while                             /* Fin de parcours Pile P2 */

            end_if
        end_if
    end_while                                         /* Fin de parcours Pile P1 */

end_while                                             /* Fin Frontière vide*/
```

Pour les fonctions suivantes, nous ne donnons que leur spécification.

ALGORITHME 11 : Traitement des états du graphe

La fonction **extrait_un_elt(Frontière)** permet d'extraire un élément contenu dans un ensemble Frontière.
- Entrée : Un ensemble d'éléments.
- Sortie : Retourne le premier élément de l'ensemble Frontière. et cet élément est supprimé de l'ensemble Frontière.

La fonctions **recupere_elt(P)** permet de copier un élément contenu dans une pile P donné en paramètre. Elle ne modifie pas la pile P.
- Entrée : Une pile P
- Sortie : Retourne une copie du dernier élément de la pile P.

La fonction **garder_sequence_menant_a_y(P,y)** donne le chemin allant de l'état initial de la pile P à l'état y successeur du dernier état de P. Il s'agit d'un simple parcours de la pile contenant les états en gardant les transitions permettant de passer d'un état à l'autre.
- Entrées : Une pile, un état.
- Sortie : Une liste (un chemin p constitué des états et des transitions permettant d'aller de l'état initial de la pile à l'état y)

La fonction **sequence_prefixe_cycle(table$_{sequence}$)**
La fonction garde tous les bouts de chemins allant de l'état initial du graphe GT à l'état accédant à la boucle état_L à partir de la table des chemins gardés jusqu'au traitement de la boucle.
- Entrée : Les différentes séquences gardées.
- Sortie : Retourne un chemin partant d'un état initial allant à l'état accédant à la boucle état_L en utilisant les transitions entre les états.

L'algorithme 10 prend en entrée un automate de Rabin paramétré G = (Q, Σ, T, q_G^{init},etat_L,etat_U) avec :
– Q est l'ensemble des états,
– Σ l'alphabet des actions du graphe,
– T est la fonction de transition : Q $\times$ Σ $\longrightarrow$ 2^Q,
– q_G^{init} est l'état initial,
– etat_L est l'ensemble des états distingués infiniment répétés, etat_L $\subseteq$ Q,
– etat_U est l'ensemble des états distingués répétés un nombre de fois fini, etat_U $\subseteq$ Q.

Le principe de l'algorithme est de trouver en deux parcours en profondeur, un chemin dans l'automate de Rabin paramétré comportant une composante fortement connexe non triviale avec au moins un état $\in$ etat_L et pas d'état $\in$ etat_U dans la composante.

Théorème 7.2.1 *Si les parcours s'arrêtent, cela signifie qu'aucun chemin comportant une composante connexe avec au moins un état_L et pas d'état_U existe. Si il n'existe pas de composante contenant un état_L et pas d'état_U alors les parcours s'arrêtent dés que tout l'automate a été visité.*

Pour éviter qu'il existe une composante avec un état_U, dès qu'un état_U est atteint, il est placé dans l'ensemble Frontière et la transition menant à cet état est supprimée du graphe. Dans ce cas, si cet état est dans une composante connexe, le fait de supprimer la transition "casse" la composante.

Si cet algorithme "casse" toutes les composantes contenant un état_U toutes les composantes restantes comportent soit aucun état_L soit au moins un état_L.

NOTE :
Le premier parcours en profondeur produit un graphe "faiblement" connexe G$'$. À partir de G$'$ (à l'aide d'un deuxième parcours en profondeur) nous recherchons toutes les composantes fortement connexes contenant au moins un état_L et pas d'état_U.

Preuve :

Considérons le premier parcours en profondeur avec le graphe de test GT = (G, $\mathcal{T}_G$, $\mathcal{C}_G$) G = (Q_{G}, A_{G}, T_{G}, q_{G}^{init}).

Ce premier parcours construit un graphe "faiblement" connexe GT$'$(G$'$, $\mathcal{T}_G{}'$, $\mathcal{C}_G{}'$) tel que G$'$ = (Q'_{G}, A'_{G}, T'_{G}, $q_{\mathrm{G}}^{init'}$) avec :
– T'_{G} = T_{G} / { $\forall \xrightarrow{a}$ | $\forall q, q' \in G_{\mathrm{G}}$ & $q \xrightarrow{a} q'$ & q' est un état_U },
– A'_{G} = A_{G} / {$\forall a \mid q \xrightarrow{a} q'$ & q' est un état_U},
– Q'_{G} = Q_{G}, $q_{\mathrm{G}}^{init'}$ = q_{G}^{init},
– $\mathcal{T}'_G$ = $\mathcal{T}_G$,
– $\mathcal{C}'_G$ = $\mathcal{C}_G$.

Pour ce parcours nous avons besoin d'une pile (P1), d'un ensemble pour marquer les états visités (DéjàVisitéM1), et d'un ensemble pour isoler les etat_U (Frontière) : initialement

(G$'$ = (Q'_{G}, A'_{G}, T'_{G}, $q_{\mathrm{G}}^{init'}$), $\mathcal{T}_G$, $\mathcal{C}_G$) = (G = (Q_{G}, A_{G}, T_{G}, q_{G}^{init}), $\mathcal{T}_G$, $\mathcal{C}_G$) ;

Le parcours débute par l'état q^{init}, puis chaque état accessible (non état_U) est placé dans la Pile P1 et l'état est placé dans l'ensemble DéjàVisitéM1 :

$$\exists \xrightarrow{a} \in T_{\mathrm{G}}, \exists q, q' \in G_{\mathrm{G}} \mid q \xrightarrow{a} q' \text{ \& DéjàVisitéM1}(q') = 0 \text{ \& } q' \text{ n'est pas un état\_U} \Longleftrightarrow \text{DéjàVisitéM1}(q') = 1.$$

$$\text{P1} = \text{P1} \cup q'.$$

Pendant le parcours du graphe, dès qu'un état $q \in$ état_U est trouvé, nous gardons la séquence menant à cette q avec la fonction : **garder_sequence_menant_a_y(P1, q, table$_{sequence}$)**. Ce chemin est construit avec le chemin contenu dans la pile P1 $\cup \{\xrightarrow{a} q\}$.

$$\text{table}_{sequence} = \text{Chemin\_P1} \cup \{\xrightarrow{a} q\}$$

NOTE :

Soit table$_{sequence}$ part de l'état initial et atteint, si elle existe, un état_U, soit elle contient le chemin élémentaire entre deux état_U. Le chemin élémentaire final est l'union des séquences allant de l'état initial au premier état_L de la composante connexe trouvée : $\bigcup$ chemin.

Si pendant le parcours du graphe un etat_U est trouvé, il appartient à un ensemble Frontière, la transition menant à cet état n'appartient plus au graphe G, l'état est marqué et la séquence menant à cet état_U est gardé.

$$\forall q \in G_{\mathrm{G}} \mid q \text{ est un état\_U} \in \mathcal{T}_G \Longleftrightarrow \text{Frontière} = \text{Frontière} \cup q.$$

$$\exists \xrightarrow{a} \mid q \xrightarrow{a} q' \text{ pour } q' \text{ un état\_U} \in \mathcal{T}_G \text{ alors } T'_{\mathrm{G}'} = T'_{\mathrm{G}'} \,/\, \xrightarrow{a} \text{ \& } A'_{\mathrm{G}'} = A'_{\mathrm{G}'} \,/\, \{a\} \text{ \&}$$

$$\text{DéjàVisitéM1}(q') = 1.$$

Tant que les états dans la pile P1 ont des successeurs le parcours continu : Si une transition $\xrightarrow{a}$ est tirable à partir d'un état $q \in$ Sommet(P1), alors l'état q' (q' = succ(q)) qui est le successeur de q par $\xrightarrow{a}$ n'appartient pas à l'ensemble Frontière. Dés qu'un état n'a plus de successeur nous dépilons les états jusqu'à obtenir un nouvel état ayant un successeur non visité.

Dés que la pile est vide, si l'ensemble Frontière $\neq \emptyset$, le premier état $q \in$ Frontière est placé en sommet de la pile P1 (sommet(P1)), et le parcours du graphe continu tel qu'il a été décrit avant, et Frontière = Frontière / $\{q\}$.

À la fin de ce parcours, nous obtenons un graphe "faiblement" connexe ne comportant pas de composantes connexes non initiales avec un/des état_U.

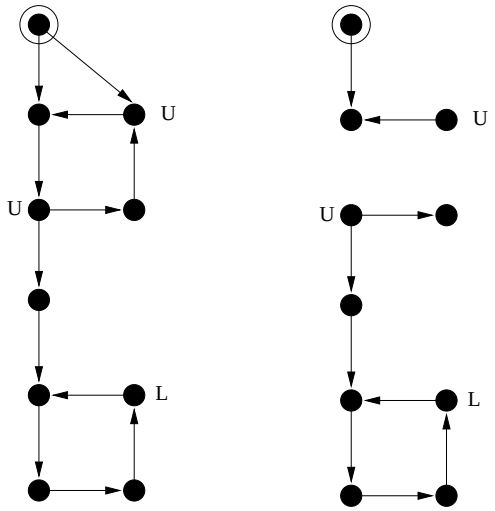

FIG. 7.5 – Graphe connexe GT (gauche) et graphe "faiblement" connexe GT$'$ (droite).

Théorème 7.2.2 *Si le deuxième parcours s'arrête (avant que la pile P2 soit vide) cela signifie qu'une composante connexe d'état contenant au moins un état_L et pas d'état_U est trouvée. Supposons qu'il y a une composante connexe et qu'un état_L est dans cette composante connexe et qu'aucun état_U appartient à la composante connexe alors le programme s'arrête.*

Preuve :

Soit GT$'$ = (G$'$, $\mathcal{T}_{G}{}'$, $\mathcal{C}_{G}{}'$) le graphe "faiblement" connexe construit à partir du premier parcours, l'algorithme proposé permet à partir de G$'$ de retrouver une composante fortement connexe contenant un état distingué (boucles état_L). Nous rappelons que le premier parcours s'arrête sur un état si tous les états sont dans DéjàVisitéM1. À partir de cet instant les états contenus dans la pile P1 sont dépilés. Si un état q de P1 est dépilé et que cet état $q \in$ état_L alors, q est empilé dans une pile P2, et devient racine d'une future composante connexe. Selon l'algorithme 8, à partir de cet état $q \in$ état_L, nous recherchons, si elle existe, une composante connexe contenant l'état_L (et pas d'état_U).

La composante connexe cherchée ne comporte aucun état_U :

1. Tant que q' = succ(q) et que q' n'est pas un état_U, alors q' est un état potentiel de la composante recherchée.
2. si q' = succ(q) et q' est un état_U alors la transition entre q et q' n'est pas possible.

Une composante est trouvée si q' = succ(q) et q' n'est pas un état_U et q' = q, à ce moment, il existe une composante connexe contenant un état_L et pas d'état_U et l'algorithme s'arrête grâce à l'instruction **stop ;**. Un cas de test est trouvé et il est constitué du chemin élémentaire (pour nous, un chemin élémentaire est une séquence d'états contenant ou non des état_U) soit $\bigcup$ chemin menant à la composante et de la composante elle-même : CT = $\bigcup$ chemin $\cup$ composante_état_L. Dans tous les autres cas, il n'existe pas de cas de test.

7.3 Conclusion

Nous venons de présenter une plate-forme pour générer un cas de test de robustesse, elle comporte :

1. Une phase de mutation de la spécification en fonction d'un modèle de faute.
2. Une phase de simulation pour transformer l'automate de la spécification mutée en un IOLTS.
3. Une phase de déterminisation, minimalisation et de suspension pour ajouter (éventuellement) des actions de blocages de l'IOLTS.
4. Puis, nous avons construit un graphe de test (avec un produit synchrone) pour intégrer les comportements de l'observateur aux comportements de la spécification.
5. Et finalement, nous sélectionnons à partir de ce graphe produit un cas de test à exécuter.

Nous pourrions améliorer l'algorithme de sélection en faisant un troisième parcours du graphe de test GT. En effet, avec l'algorithme présenté, nous n'obtenons qu'un cas de test contenant un seul chemin menant à une boucle état_L. Un dernier parcours permettrait de donner tous les chemins élémentaires (ne contenant pas de boucles état_U et de boucles état_L) allant de l'état initial du graphe $GT^{'}$ à l'état_L (de la boucle). Dans ce cas, nous pourrions constituer plusieurs cas de test contenant chacun un chemin élémentaire (en appliquant par exemple la règle de contrôlabilité).

Les algorithmes présentés ont donné lieu à un prototype présenté dans le chapitre 8 suivant.

Chapitre 8

Mise en œuvre

Nous présentons, dans cette troisième, et dernière partie, une chaîne complète d'outils. Le but de ce prototype est, d'une part de proposer une mise en œuvre des principes et algorithmes de génération de cas de test (développés dans les deux premières parties), et d'autre part d'utiliser et d'adapter des outils d'exécution de cas de test pour vérifier la robustesse de programmes écrits en Java (en fonction de propriétés de robustesse données). Les différentes phases de fonctionnement de cette chaîne d'outils sont illustrées au moyen d'un exemple de présentation.

Le plan de cette partie est le suivant : Les deux premières sections présentent l'architecture générale de la chaîne d'outils et son principe de fonctionnement. La section suivante donne le formalisme utilisé pour décrire les spécifications (le langage IF). Cette introduction permet d'établir le parallèle entre les modèles (présentés dans le document) et les formats utilisés pour le prototype. A travers un exemple, nous présentons ensuite les étapes de construction pour obtenir une spécification de référence, un graphe de test et un cas de test pour une propriété et une implantation données. Pour conclure, nous proposons une exécution d'un cas de test sur une implantation (écrite en Java), en donnant pas à pas les étapes nécessaires à l'utilisation et à l'adaptation d'un outil d'exécution (Spider).

8.1 Une chaîne d'outils pour le test de robustesse de programmes Java

Le schéma de la figure 8.1 représente une chaîne d'outils permettant d'exécuter des cas de test de robustesse sur des implantations Java. Cette chaîne d'outils est construite à partir d'un ensemble de composants existant contenu dans des boîtes à outils IF (pour l'outil simulateur), CADP (pour les outils de déterminisation et minimisation), TGV et AGEDIS (pour exécuter les cas de test) et d'opérateurs (suspension + δ, produit, ou sélection) développés en C et/ou C++. La théorie de ces opérateurs est celle vue précédemment. Pour présenter le fonctionnement de l'outil nous utilisons un exemple, celui d'un distributeur de tickets : La Machine Ticket détaillée dans la suite. Sur la figure 8.1 de l'outil, les données initiales sont "les ovales de couleur", les opérateurs (adaptés ou créés) sont "les rectangles" et les résultats des opérations sont "les ovales".

NOTE :
Les outils réalisés sont : "suspension+δ", "produit", "sélection", les outils adaptés sont : "translateur" (translateur a été réalisé en partie par un stagiaire étudiant), "Spider" et les outils réutilisés sont : "simulation", "déterminisation", "minimisation". La mutation reste encore une activité "manuelle". Elle est en cours de réalisation pour muter de façon automatique une spécification avec le modèle de faute.

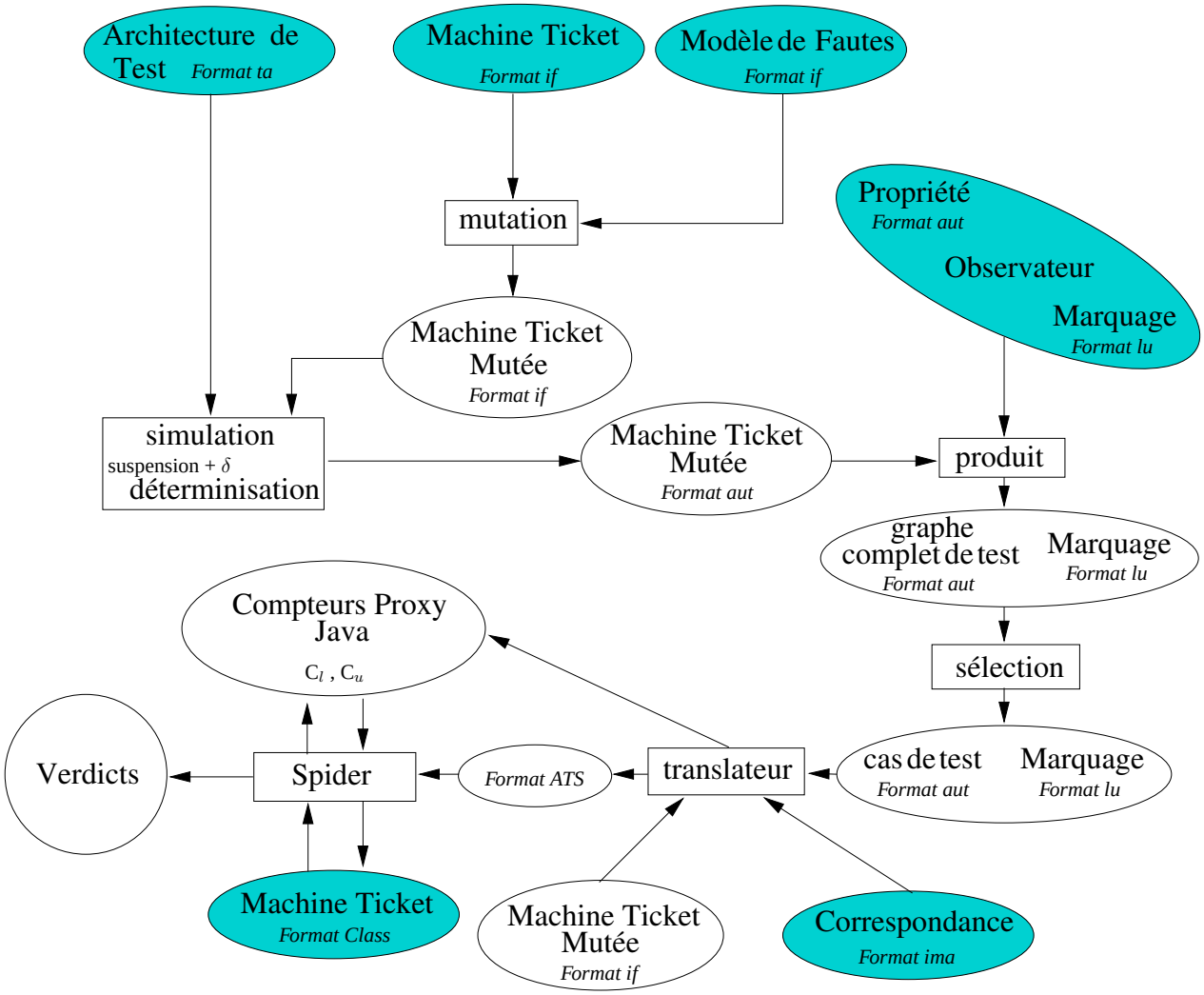

FIG. 8.1 – Représentation d'une chaîne complète d'outils pour le test de robustesse des programmes Java (incluant les formats d'entrée).

Les différents éléments de la chaîne d'outils :

1. Les données fournies en entrée sont :

   – Une spécification : décrite en IF (voir la section 8.3 pour la présentation de se langage, [Boz99, BGMS98]) elle présente les comportements attendus de l'implantation dans un contexte et un environnement nominal.
   – Une architecture de test : il s'agit d'une liste d'actions visibles correspondant aux appels des méthodes publiques fournies par l'implantation Java.
   – Une correspondance : donne, dans un fichier, l'équivalence entre les noms des méthodes de l'implantation Java et les actions visibles de la spécification.
   – Un modèle de faute : il est donné sous la forme d'un ensemble (tableau des fautes de la section 6.5.3) de mutations à appliquer sur la spécification nominale.
   – Un observateur : il représente la traduction de la négation d'une propriété de robustesse donnée. Cet observateur est décrit par un automate de Rabin paramétré (IOLTS paramétré avec un fichier de marquage). Dans notre cas, le marquage (donné par un fichier annexe) représente la liste des état_U et des état_L présent dans la description de l'observateur.
   – Les compteurs proxy Java : ce fichier fixe les valeurs des compteurs c_l et c_u. Il introduit la relation entre les état_U et les état_L présents dans le cas de test en fixant les fonctions d'incrémentations, dans le code de l'implantation à tester. Les valeurs d'incrémentations des compteurs ont un rôle dans le verdict de robustesse.
   – Une implantation : ce fichier est le code du programme à tester écrit en Java.

2. Les données produites sont :

   – Une spécification mutée.
   – Un graphe complet de test (GT accompagné d'un fichier de marquage).
   – Un cas de test (CT accompagné d'un fichier de marquage).

3. L'outil Spider permettant d'exécuter les cas de test de robustesse est un composant de l'outil développé par IBM Haifa dans le projet AGEDIS. Pour utiliser l'outil Spider, nous adaptons les formats d'entrée et les étapes d'exécution avec :

   – Les cas de test générés (IOLTS) sont transformés (avec l'outil translateur aut2ats) dans un format ATS (Abstract Test Suite).
   – Le testeur (Spider) effectue l'exécution du cas de test (ats) en interaction avec l'implantation Java donnée et met à jour les compteurs c_l et c_u.
   – Le résultat d'une exécution est un verdict de robustesse obtenu en interprétant les valeurs les compteurs c_l et c_u.

Remarque : *Nous avons choisi :*

1. *De faire un fichier de marquage indépendant pour ne pas modifier le format des automates, car ce format est utilisé comme paramètre pour les outils existants. Les fichiers de marquages sont calculés et disponibles si nécessaire.*
2. *Le langage IF, car il existe dans sa boîte à outils des opérateurs permettant de manipuler et de modifier les données pour effectuer des analyses statiques et gérer, le cas échéant des problèmes liés à l'explosion des états des automates (des outils de slicing par exemple).*

NOTE :
Les formats de fichier suivants sont décrits dans la suite du document : – Les données (spécification) IF : .if – Les IOLTS (det($\Delta(S_m)$), observateur, GT, CT en sortant du produit) : .aut – La liste des actions visibles (architecture de test) : .ta – La liste des correspondances entre implantation et spécification : .ima – Les objectifs de test : .td – Les listes des marquages : .lu – Les cas de test en entrée de l'outil Spider : .ats

8.2 Principe de l'outil de test de robustesse

L'outil est constitué de plusieurs composants. L'exécution des composants non liés entre eux peut se faire de manière indépendante. Il faut toutefois garder un ordre dans la construction des cas de test et leur exécution.

Les étapes d'une génération de cas de test sont :

1. Mutation de la spécification $\Longrightarrow$ Cette opération a besoin comme données d'entrée, de la spécification (IF) et du modèle de faute, et produit une spécification mutée (IF).
2. Simulation $\Longrightarrow$ Cette opération disponible dans la boîte à outils IF a besoin comme données d'entrée, d'une architecture de test (ensemble des actions visibles), d'une spécification (IF), le résultat est un automate IOLTS représentant la spécification. Cette opération est une transformation, traduisant une spécification vers un modèle IOLTS.
3. Réduction et détection des blocages $\Longrightarrow$ Cette opération a besoin comme données d'entrée, d'une spécification (IOLTS), et d'une architecture de test, le résultat est la suppression des actions non visibles et la transformation des boucles d'actions internes en une action δ.
4. Déterminisation et minimisation $\Longrightarrow$ Cette opération est disponible dans la boîte à outils CADP (option "det" et "min" de l'outil Aldebaran [Fer88]). À la fin de toutes ces opérations nous obtenons un automate IOLTS déterministe avec éventuellement des actions δ de blocage (det$\Delta(S_m)$) et minimal : det($\Delta(S_m)$)$_{mini}$.
5. Produit synchrone $\Longrightarrow$ Cette opération a besoin comme données d'entrée, d'un det($\Delta(S_m)$)$_{mini}$ et de l'observateur, et produit un graphe de test (automate IOLTS) accompagné d'une table des marquages (pour distinguer les états du graphe). Un état du graphe est constitué d'un état de la spécification et d'un état de l'observateur. Le marquage d'un état est en relation avec l'état "distingué" de l'observateur (contenu dans cet état).

6. Sélection $\Longrightarrow$ Cette opération a besoin comme données d'entrée, du graphe de test et des marquages, le résultat est un automate IOLTS représentant un cas de test et ses marquages.

Les étapes d'une exécution sont :

1. Traduction $\Longrightarrow$ Cette opération est disponible dans la boîte à outils AGEDIS (Spider). Les données d'entrée sont un cas de test (accompagné des marquages), une table de correspondances, l'architecture de test. Le résultat produit par l'outil Spider est un cas de test au format ATS prêt à exécuter, ainsi qu'un programme Java (proxy) permettant la gestion des compteurs du cas de test.
2. Exécution $\Longrightarrow$ Cette opération est disponible dans la boîte à outils AGEDIS (Spider), et a besoin comme données d'entrée, du cas de test, du proxy Java et de l'implantation Java. Le testeur exécute le cas de test sur l'implantation, et incrémente les compteurs lorsqu'un état_U ou un état_L du cas de test est rencontré. Le résultat d'une exécution est un verdict donné en fonction des valeurs des compteurs.

Nous présentons maintenant, le modèle (langage) IF servant à décrire les spécifications, puis, l'exemple de la Machine Ticket utilisé pour dérouler une exécution complète de notre chaîne d'outils.

8.3 IF : Langage de description

Nous représentons les spécifications sous une forme intermédiaire IF (*Intermediate Format*) [BFG+99b]. IF est un langage permettant la description d'arbres abstraits [Boz99, BGMS98], l'expression de spécifications de plus haut niveau comme SDL ou UML. Ce langage, développé au sein du Laboratoire Vérimag, est une représentation à base d'automates temporisés communicants, conçu pour décrire et valider formellement des systèmes asynchrones. Le langage IF possède une sémantique opérationnelle complètement définie en termes de LTS. Concrètement, la sémantique permet, à partir d'un ensemble de règles, de construire tous les comportements d'une exécution d'un programme sous la forme d'un LTS. Le langage IF permet de représenter l'ensemble des composants et les liens d'un système. Il devient facile avec ce langage de reproduire les interactions d'un protocole avec son environnement (vu comme un processus). Il est également possible pour des processus de décrire les comportements internes et inter-processus. La modélisation explicite des actions va nous permettre de distinguer plus finement les transitions à garder pour constituer les cas de test (simulation avec l'architecture de test).

IF (dans sa version évoluée IF-2.0 [BGMO04]) permet de modéliser des systèmes à plusieurs composants. Dans ce cas, les programmes IF sont composés d'objets actifs (perçus comme des instances de processus) exécutés en parallèles. Les interactions entre les composants sont des messages transmis via des files d'attente de communication. Le modèle IF permet de définir les voies de transmission de messages (appelé "signalroute") et les adressages entre les processus, et gère de façon dynamique l'évolution, la création et la destruction des processus intermédiaires nécessaires à une exécution (et donc des "signalroute" entre processus dynamiques). En résumé, un programme IF est constitué de composants génériques incluant des instances dynamiques comme les processus, les

signalroutes, les signaux et des composants plus statiques et parfois partagés comme les variables, des types de données, des valeurs de constantes et des procédures externes.

Nous choisissons le modèle (langage) IF, car il est suffisamment expressif pour rester en amont du problème d'explosion d'états et pour simuler simplement des concepts existants dans les formalismes de spécification. Les programmes représentés sous la forme IF gardent de manière explicite un certain nombre d'informations, comme par exemple le parallélisme, le temps ou encore les données manipulées restent également disponibles. De nombreux outils [BGOS04] ont été développés autour du modèle IF et servent généralement en amont ou pendant la génération de cas de test pour optimiser la taille des modèles produits. En particulier, ils offrent la possibilité d'effectuer des analyses statiques telles que l'analyse des variables actives [BFG99a] ou le $slicing$ [BFG00] (utilisant des théories de [Wei79, Tip95, Gou97]). De manière générale, tous les travaux sont issus du domaine de l'optimisation de code.

8.3.1 Définitions globales

Un programme IF est un système contenant des **Définition_globale** et un ensemble de **Processus**. Chaque processus $\in$ **Processus** décrit un comportement séquentiel comprenant les transformations de données, les communications avec d'autres processus, la création et la destruction dynamiques de processus. Syntaxiquement une spécification IF est constituée :

1. D'un nom unique de système : **system**$_{id}$.
2. D'une liste des éléments statiques de communications et des définitions de types : "Définition_globale".
3. De la description de chacun des processus : Processus.

system : :=
  system **system**$_{id}$;
    Définition_globale
    {**Processus**}*****
  endSystem ;

Une **Définition_globale** est un tuple (**Type_donnée**, **Signaux**, **SignalRoute**, **Variables**, **Procédures externes**). Ces définitions servent pour les interactions des automates communicants (les messages, les canaux . . .) :

– **Type_donnée** est un ensemble de types de donnés tels que les entiers, les booléens, les tableaux, les enregistrements et des types abstraits : Type : := Type$_{id}$ **;**

– **Signaux** est l'ensemble des signaux paramétrés représentant les messages transmis entre processus. Les signaux peuvent être adressés directement au processus (par l'utilisation de son numéro identifiant pid) et/ou avec un itinéraire. Le processus de destination stocke le signal reçu dans

une file d'attente. Les signaux dans la file d'attente d'entrée sont consommés en utilisant le mode "fifo" : **signal** : := action **;**

– **SignalRoute** dénote les canaux des communications entre processus. Nous les utilisons également comme un lien entre des processus pour assurer et réaliser l'envoi de messages. Les comportements des canaux sont définis en utilisant une politique de raccordement (comme peer to peer, ou multicast) : **signalRoute** : := signalRoute$_{id}$ **From** processus$_{id}$ **to** processus$_{id}$ **with** signal$_{id}$, . . . , signal$_{id}$ **;**

– **Variables** est un ensemble de définitions des variables globales : Variables : := **type** Variables$_{id}$: Type$_{id}$ **;**

– **Procédures externes** représentent des transformations de données écrites en langage C. Elles sont utilisées par appel de méthode.

8.3.2 Processus

Un processus est un automate d'états finis étendu avec des données qui possède des variables, des états de contrôle et une file d'attente.

Le comportement d'un processus est spécifié par un ensemble de **Transitions**. Chaque transition $\in$ **Transitions** permet de passer d'un état de contrôle à un autre en exécutant une action (qui peut être gardée). L'exécution d'une transition est atomique (elle correspond à la transition des modèles LTS). Plusieurs transitions peuvent être permises en même temps, dans ce cas le choix de la transition est non déterministe. Les transitions peuvent être déclenchées par des signaux contenus dans les files d'attente ou par des actions spontanées.

Une instance d'un processus peut être créée ou détruite de façon dynamique pendant l'exécution d'une action.

Chaque processus est constitué d'un identifiant et d'une instance de processus. La spécification du processus (= **Corp_Processus**) est constituée d'un ensemble de variables locales, d'un ensemble d'états de contrôle et d'un ensemble de transitions.

Un processus **P** $\in$ Processus est un tuple = (Identifiant de processus, Nombre initial d'instance, Corp_Processus) avec Corp_Processus = (Variables, Etats, Transitions) :

Process : :=
Process **Identifiant de processus** (**Nombre initial d'instance**) ;
{**Corp_Processus**}*
endProcess ;

– **Identifiant de processus** représente l'identificateur unique du processus.

– **Nombre initial d'instance** est une constante donnant le nombre d'instances de processus créé initialement. Chaque instance est identifiée par un numéro unique (pid).

– **Variables** est l'ensemble des déclarations des types de variables ou des paramètres locaux.

– **Etats** est l'ensemble des états de contrôle du processus. Un attribut #**start** peut être donné à un état. Il indique que l'exécution du processus peut démarrer à partir de cet état (un seul attribut #**start** est présent par processus). Aux états peuvent être associés des filtres sur les files d'attente, qui peuvent retarder la consommation de certains signaux (*save*(q)) pour des moments ultérieurs, ou alors provoquent la destruction de signaux considérés inutiles (*discard*(q)).

– **Transitions** est l'ensemble des transitions du processus. Une transition est permise dans un état si son signal de déclenchement est présent (dans la file d'attente par exemple) et que sa garde est évaluée, en respectant la condition de transition, à vrai. Le corps d'une transition est un programme et si q, q$'$ $\in$ **état de contrôle** une transition est exprimée par :

$$q \xrightarrow{[garde]\ declencheur +\ corps} q'$$

– **[garde]** est un prédicat représentant la condition de la transition qui dépend des variables visibles dans le processus.

– **declencheur** est une opération élémentaire de réception : : **input** signal$_{id}$ **;**

– **corps** est une séquence d'opérations élémentaires (affectations de variables, messages envoyés, créations/destructions de processus, etc..) :

- Les émissions : **output** signal$_{id}$ **via** signalroute$_{id}$ **to** processus$_{id}$ **;**
- Les affectations simples : **TASK** variable$_{id}$:= 0 **;**

NOTE :
Les corps de transitions (signaux) peuvent inclure des appels à des fonctions et/ou des procédures externes, écrites dans un langage de programmation externe (comme les langages C/C++) : **call** + nom de la fonction.

8.4 Présentation de l'exemple : Un distributeur de Tickets

Dans cette section, nous illustrons le fonctionnement de la chaîne de génération et d'exécution de test de robustesse sur un exemple (un distributeur de tickets). Cet exemple "prototype" permet également de donner les différents formats d'échanges entre les outils. Dans la suite du document, nous nommons l'exemple du distributeur de tickets : Machine Ticket.

Présentation fonctionnelle de l'exemple

Nous proposons l'exemple d'un distributeur, cas classique d'un système en interaction avec son environnement. Ici, nous adaptons cet exemple pour mettre en valeur les aspects liés à la robustesse.

La Machine Ticket délivre un ticket (papier) sur demande d'un utilisateur si celui-ci fournit une somme au moins égale à celle nécessaire pour un ticket. Tant que la somme fournie est insuffisante, la distribution d'un ticket est impossible. Si la somme est supérieure au prix demandé, un rendu de monnaie est effectué.

La figure 8.2 présente l'architecture principale de la Machine Ticket, plus précisément, il s'agit d'un système formé de trois composants (processus) qui communiquent tous entre eux. Ce système comporte :

1. Un chargeur de pièces qui emmagasine ou restitue les pièces données par un utilisateur, et indique les valeurs des pièces introduites à un contrôleur. À tout moment, il connaît le nombre de pièces dont il dispose et peut transmettre cette information à un contrôleur qui le demande.
2. Un contrôleur qui dialogue avec le chargeur de pièces en fonction des demandes de l'utilisateur. Il gère la distribution ou l'annulation des tickets et calcule le rendu de monnaie en fonction des pièces disponibles dans le chargeur.
3. Un utilisateur qui introduit et récupère des pièces auprès du chargeur, imprime et retire un ticket auprès du contrôleur et qui peut à tout moment annuler une transaction.

Les deux composants, chargeur de pièces et contrôleur communiquent par échange de messages via deux canaux C et A. La communication entre l'utilisateur et les deux autres composants est possible via le canal U.

NOTE :
Le chargeur de pièces peut être partagé par plusieurs contrôleurs.

Architecture de la Machine Ticket

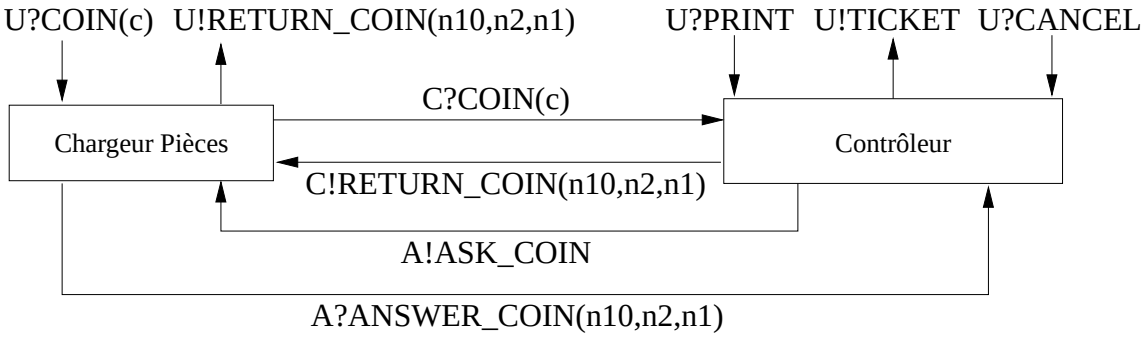

FIG. 8.2 – Représentation des composants Contrôleur et Chargeur de pièces de la Machine Ticket en interaction avec un utilisateur.

Fonctionnement de la Machine Ticket

L'utilisation nominale de la Machine Ticket est la suivante :

1. Les valeurs des pièces admises par le chargeur sont (1, 2, 10).
2. Le contrôleur reçoit pour chaque pièce introduite dans le chargeur sa valeur au moyen de l'action A ?COIN(c) (où c représente la valeur de la pièce introduite).
3. L'utilisateur peut à tout moment demander un ticket en donnant l'information au contrôleur avec U ?PRINT, ou annuler une transaction avec U ?CANCEL.
4. Si l'action U ?PRINT est demandée, le contrôleur vérifie le crédit de l'utilisateur. Si le crédit est suffisant le contrôleur délivre un ticket avec U !TICKET. Dans le cas contraire, aucun ticket n'est délivré et le contrôleur reste en attente d'une nouvelle manipulation de l'utilisateur.

   – Dans le cas où un ticket est délivrable par le contrôleur, deux étapes deviennent possibles :
     (a) Le crédit donné par l'utilisateur est strictement égal au prix du ticket. Alors, le contrôleur distribue le ticket et clôture la transaction.
     (b) Le crédit donné est supérieur au prix du ticket. Le contrôleur calcule la monnaie à rendre ainsi que le nombre de pièces à sélectionner dans le chargeur pour y parvenir en fonction des pièces disponibles dans le chargeur. Pour effectuer cette sélection le contrôleur a besoin de connaître la quantité de pièces dans chaque catégories (1,2,10) auprès du chargeur. Un dialogue est donc nécessaire entre le contrôleur et le chargeur de pièces :
        i. Le contrôleur demande, par A !ASK_COIN, au chargeur le nombre de pièces disponibles dans chaque catégorie de pièces.
        ii. Le contrôleur se met en attente d'une réponse du chargeur.
        iii. Après calcul du nombre de pièces restantes, le chargeur le transmet avec C !ANSWER_COIN(n10,n2,n1).
        iv. Le contrôleur calcule dans chaque catégorie le nombre de pièces à rendre, et le transmet au chargeur avec C !RETURN_COIN(n10,n2,n1).
        v. Le contrôleur délivre un ticket avec U !TICKET.
        vi. Le chargeur donne la monnaie à l'utilisateur dès réception de C !RETURN_COIN(n10,n2,n1).

   – Si un ticket n'est pas délivrable :
     (a) Le contrôleur se met en attente de nouvelles actions de la part de l'utilisateur ou du chargeur de pièces.

5. A n'importe quel instant l'utilisateur peut annuler une transaction en cours en utilisant U ?CANCEL. Dés la réception de cette action (un nouvel échange de messages est réalisé entre le contrôleur et le chargeur), le contrôleur transmet au chargeur la quantité de pièces à rendre. Le chargeur rend alors les pièces avec C !RETURN_COIN(n10,n2,n1) à l'utilisateur.

Remarque : *L'exemple du calcul du rendu de monnaie est réalisé lors d'un dialogue entre le chargeur et le contrôleur. Ce dialogue est indispensable car le chargeur peut être partagé par plusieurs contrôleurs et seul le chargeur est capable (dans notre protocole) de gérer et le nombre de pièces dans chacune des catégories de pièces. Dans ce cas un dialogue est exécuté de manière atomique.*

Sans donner tous les détails d'un programme IF correspondant au système de la Machine Ticket nous proposons un aperçu de la spécification du programme écrit en IF. La Machine Ticket IF comporte des signaux (COIN, CANCEL, ...), les "signalroute" (les liens entre les composants, contrôleur $\longleftrightarrow$ chargeur de pièces, ...) et trois processus "controller", "coinsTray", et "user" (ce dernier processus pouvant être un processus d'environnement dans un programme IF).

```
system MachineTicket ;
type CoinValue = range 1..11 ;
...
signal UTICKET() ;
signal UPRINT() ;
signal UCOIN(CoinValue) ;
...
signalroute UserController(1)
            from user to controller with UPRINT URESET ..... ;
...
procedure ComputeNbcoin ;
      fpar /* Liste des paramètres d'entrées ou de sorties */
{#
... /* Corps d'une procédure C */
#}
endprocedure ;
...
process controller(1) ;
var c integer ;
...
state état0 #start ;
      input UCOIN(c) ;
   nextstate état1 ;
endstate ;
state état1 ;
      call CompChange(c, n10, n2, n1) ; /* Appel d'une procédure C */
   nextstate état0 ;
endstate ;
...
endprocess ;
...
endsystem ;
```

Propriété formelle de la Machine Ticket

"Une distribution est possible si le chargeur de pièces contient deux unités de la monnaie en vigueur".

8.5 La spécification de référence utilisée par la chaîne d'outils

Nous ne modélisons pas l'intégralité de la Machine Ticket (figure 8.2) mais uniquement le fonctionnement du contrôleur (figure 8.3). Cette spécification suffit pour montrer les principes de l'outil et vérifier des propriétés de robustesse de ce composant. Nous pouvons facilement imaginer que ce composant soit utilisé dans d'autres machines de distribution et c'est pour cette raison que le test de robustesse d'un composant à un sens. L'environnement de ce composant est constitué du chargeur de pièces et de l'utilisateur. Ces deux composants formaient implicitement l'environnement nominal, maintenant ils forment l'environnement dégradé du contrôleur à tester. Notre spécification de référence devient :

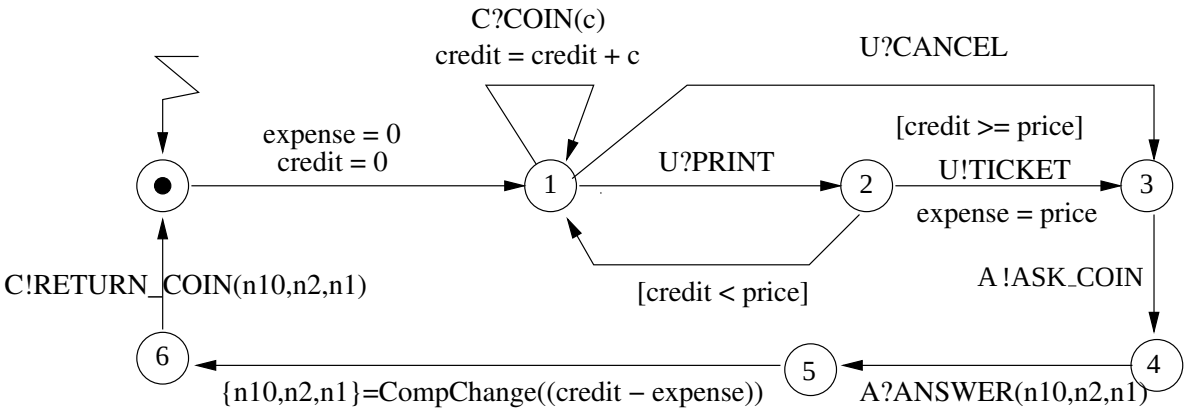

FIG. 8.3 – Spécification représentant le comportement du contrôleur de la Machine Ticket.

- Les variables sont : expense, credit, price.
- La fonction de calcul est ; CompChange().
- Les signaux d'échanges sont : ?PRINT, ?CANCEL, !TICKET sur le canal U avec l'utilisateur, ?COIN(), !RETURN_COIN() sur le canal C, puis ?ANSWER, !ASK_COIN sur le canal A avec le chargeur de pièces.

8.6 La mutation : Spécification / Modèle de faute

Nous avons présenté une Machine Ticket comportant trois composants communicants entre eux. Le test que nous décidons de réaliser va se porter sur le composant contrôleur, et les fautes proviennent du chargeur de pièces ou de l'utilisateur.

Modèle de faute : Nous proposons deux fautes éventuelles :

1. La première porte sur l'action d'envoi (A !ASK_COIN) du contrôleur au chargeur de pièces, et représente une coupure de réception notée : [t] ?coupure.
2. La deuxième faute est une panne de composants, aucune communication n'est possible avec le contrôleur, notée : ?panne avec un nouvel état.

Les fautes décrites ci dessus sont données par le tableau du modèle de faute suivant :

Coupure	Ancien Nœud	Domaine de faute	Nouveau Nœud
d'un canal de réception	$3 \xrightarrow{\text{A !ASK\_COIN,}} 4$	{ASK_COIN, coupure,[f],[t])}	$3 \xrightarrow{\text{[f] ?coupure}} 4$ $3 \xrightarrow{\text{[t]A !ASK\_COIN}} 4$
Panne			
d'un composant	1	{(1, panne)}	$1 \xrightarrow{\text{?panne}} 7$

Remarque : *Nous supposons ces fautes car les situations engendrées peuvent être bloquantes pour les calculs des fonctions du contrôleur. Elles ont été choisies (parmi d'autres possibles) uniquement pour illustrer le fonctionnement de l'outil.*

Spécification mutée

Nous disposons maintenant d'une spécification et d'un modèle de faute. La première opération est la mutation, en appliquant la fonction Apply(Δ, Spec Contrôleur, Modèle de faute). L'opération de mutation produit la spécification mutée suivante :

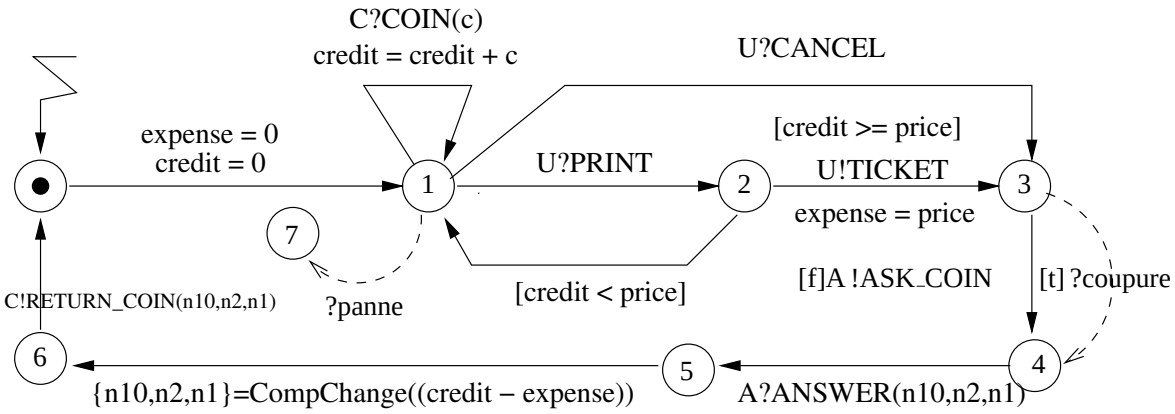

FIG. 8.4 – Spécification mutée

Au niveau de la spécification IF, deux nouveaux signaux panne() et coupure() sont placés dans l'ensemble des signaux, un nouvel état est placé dans l'ensemble des états du processus "controller", et de nouvelles transitions apparaissent dans le processus "controller".

Les transitions ajoutées sont :

(1) une action de panne représentée par ?panne entre l'état 1 et un nouvel état 7

```
Signal panne() ;        /* Ajout du signal panne */
Signal coupure() ;      /* Ajout du signal coupure*/
. . .
process controller(1) ;
. . .
state état1 ;
/* Ajout d'une transition pour effectuer la panne */
   Input panne() ;
     nextstate état7 ;
. . .
endprocess ;
```

(2) une action de coupure représentée par [t] ?coupure entre l'état 3 et l'état 4.

NOTE :
Des études sont en cours pour réaliser une fonction automatique de mutation des programmes IF en relation avec le modèle de faute. La mutation présentée actuellement reste une étape manuelle.

8.7 Simulation, suspension, déterminisation, minimisation de la spécification mutée

Un composant de la boîte à outils IF permet de produire un IOLTS (.aut) en partant d'un fichier IF. Pour faire cette simulation, l'outil a besoin d'une architecture de test (pour définir les actions visibles). Dans cette boîte nous avons ajouté un algorithme permettant de calculer l'automate de "suspension + δ". Nous obtenons après ces deux étapes un IOLTS ne comportant que les actions visibles. Puis, l'outil Aldébaran de la boîte à outils CADP, nous permet de déterminiser et minimiser ce IOLTS.

Architecture de test .ta

Ce fichier est la liste des actions visibles choisie par un utilisateur (actions provenant du fichier IF). Cette architecture de test se représente de la façon suivante :

```
observe :          /* Déclaration des actions observables */
IF_return_COIN
IF_return_PRINT
...
control :          /* Déclaration des actions contrôlables */
IF_call_COIN
IF_call_PRINT
...
```

Après toutes ces étapes nous avons une spécification det(Δ(S$_m$)) minimale : det(Δ(S$_m$))$_{mini}$.

8.8 Propriété, observateur et fichier de marquage .lu pour test la robustesse du contrôleur de la machine Ticket

La propriété de robustesse

La propriété que nous voulons vérifier est : "Inévitablement si la valeur c d'une pièce est transmise au contrôleur par l'action C ?COIN(c), le contrôleur doit fournir au chargeur, avec l'action C !RETURN_COIN(n10,n2,n1), les valeurs de n10, n2 ,n1 correspondant au nombre de pièces à rendre". Dans la conception du contrôleur, si il n'y a pas de pièces à rendre, les valeurs des n10, n2, n1 sont égales à zéro. Dans ce cas les valeurs même nulles sont obligatoirement transmises au chargeur de pièces. La négation de la propriété de robustesse est donnée par l'automate de Rabin paramétré suivant :

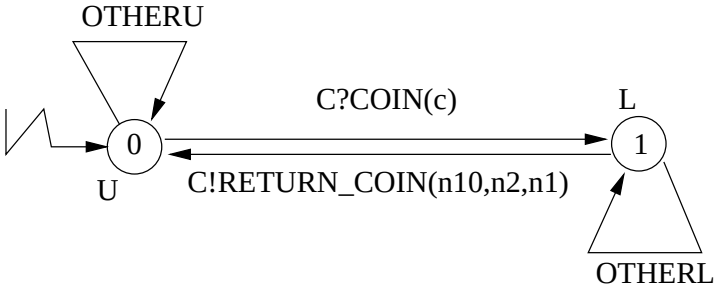

FIG. 8.5 – Observateur de la propriété de robustesse à vérifier.

L'automate de la propriété doit être complet pour accepter toutes les entrées et sorties de l'implantation. Nous représentons la complétude avec les actions "otherU" et "otherL" (dénotant toutes les actions non présentes dans les états de l'automate de la propriété).

L'observateur en format IOLTS .aut

Nous n'avons pas encore donné le format d'un fichier IOLTS. Nous profitons de cette partie pour le faire. La description d'un automate reste classique, nous précisons pour la structure de données l'état initial, le nombre de transitions, le nombre d'états de l'automate. Les transitions entre chaque état sont écrites par un ensemble représentant, un état$_{cible}$, l'identifiant du processus, l'action à effectuer et l'état état$_{source}$, tout cela se retrouve dans :

des(N° état initial,Nombre de transition, Nombre d'état)
(état$_{cible}$, "Processus + action", état$_{source}$)
. . .

Nous donnons un aperçu de la structure modélisant l'observateur. Il faut placer les actions complémentaires à l'action C ?COIN au niveau de l'état 0. Nous procédons de la même façon pour représenter l'action "otherL" sur l'état 1 (dans ce cas ce sont les actions complémentaires à RETURN_COIN qu'il faut placer

```
des(0,4,2)
(0, " otherU", 0)
(0, " C ?COIN ", 1)
(1, " C !RETURN_COIN ", 0)
(1, " otherL ", 1)
```

Fichier de Marquage .lu

Le format .aut dont nous disposons ne permet pas de représenter les informations associées aux états de l'automate. Nous choisissons de représenter un marquage indépendant pour les état_U et les état_L. Nous utilisons pour cela un fichier annexe nommé Marquage (de format .lu). Ce fichier est donné sous la forme d'un tableau contenant initialement les états de l'observateur et leurs marquages U et L respectifs. Ce fichier, disponible à tout moment, est modifié en fonction de l'avancement dans la chaîne d'outils. Il correspond successivement au marquage des états de l'observateur puis à celui des états du graphe de test et enfin à celui des états des différents cas de test.

des(Nombre de description**)**
Numéro de l'état **:** L,
Numéro de l'état **:** U,

Format générique d'une table de marquage.

des(2**)**
0 **:** U,
1 **:** L,

Format de la table de marquage initiale pour l'observateur.

Dans le tableau "**des(Nombre de description)**" indique le nombre d'état marqué. Chaque ligne du tableau, mise à part la première est constituée d'un état et d'un marquage.

> **Remarque :** *Les fichiers de marquage ne sont jamais vides. Même si l'automate de l'observateur décrit une propriété de sûreté, l'observateur est paramétré par des état_U. La raison pour garder le marquage est qu'un observateur (et donc son marquage) sert d'oracle pour le test de robustesse.*

8.9 Graphe de test, sélection et cas de test

Le graphe de test est le produit synchrone entre la spécification mutée (figure 8.4) et l'observateur (figure 8.5). La construction de ce graphe ne pose pas de problème et respecte les règles de construction données en deuxième partie du document. La sélection a pour but de donner des séquences d'exécution comportant des boucles contenant au moins un état_L. Les cas de test produits peuvent

être par exemple les suivants :

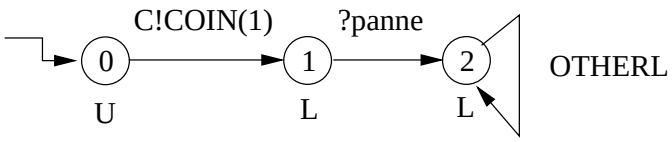

FIG. 8.6 – Cas de test de robustesse pour exprimer la faute de panne.

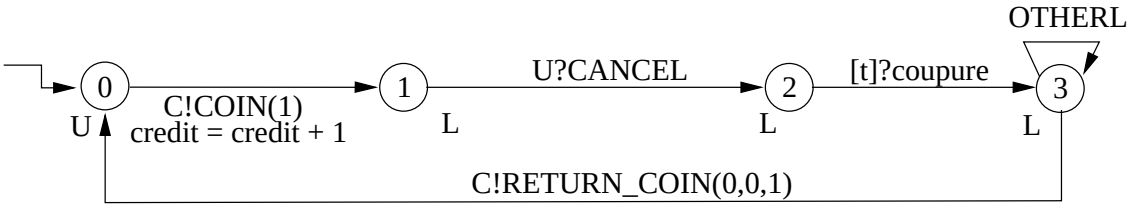

FIG. 8.7 – Cas de test de robustesse pour exprimer la faute de coupure.

8.10 Proposition d'une implantation pour le composant contrôleur de la Machine Ticket

Une implantation naïve du composant contrôleur

Une implantation naïve est la reproduction exacte des comportements de la spécification fonctionnelle nominale.

Exécuter les cas du test sur une telle implantation peut ne pas satisfaire la propriété de robustesse donnée et amener à un verdict **NonRobuste**. En effet, avec les deux cas de test produits, la valeur limite du compteur associé à un état_L serait rapidement atteinte par le fonctionnement de l'implantation.

Une implantation du composant contrôleur orientée robustesse

Nous proposons de donner une implantation un peu plus élaborée. Cette implantation doit dans sa réalisation essayer d'être robuste sur deux points :

1. Dans le fonctionnent de la Machine, si un utilisateur introduit un certain nombre de pièces, il doit demander un ticket. Deux cas se présentent. Soit l'utilisateur ne donne pas assez de

pièces pour le prix d'un ticket. Soit l'utilisateur rentre un grand nombre de pièces sans demander un ticket. Dans les deux cas l'implantation rend les valeurs de pièces. Pour cela une temporisation est réarmée à chaque réception de pièces.

2. Comme l'implantation doit rendre la valeur des pièces, le contrôleur ne connais pas exactement le nombre de pièces restant dans le chargeur. Si le chargeur ne transmet pas la quantité de pièces disponibles, un calcul au niveau du contrôleur est toujours possible. Ce calcul est réalisé en utilisant les dernières quantités de pièces reçues, et donc un rendu de monnaie est peut être possible. Le calcul est traduit par une fonction CompDefChange(n10,n2,n1). La fonction est déclenchée après une expiration d'une temporisation.

Nous ne donnons pas le code de l'implantation Java, mais nous la représentons sous sa forme d'automate :

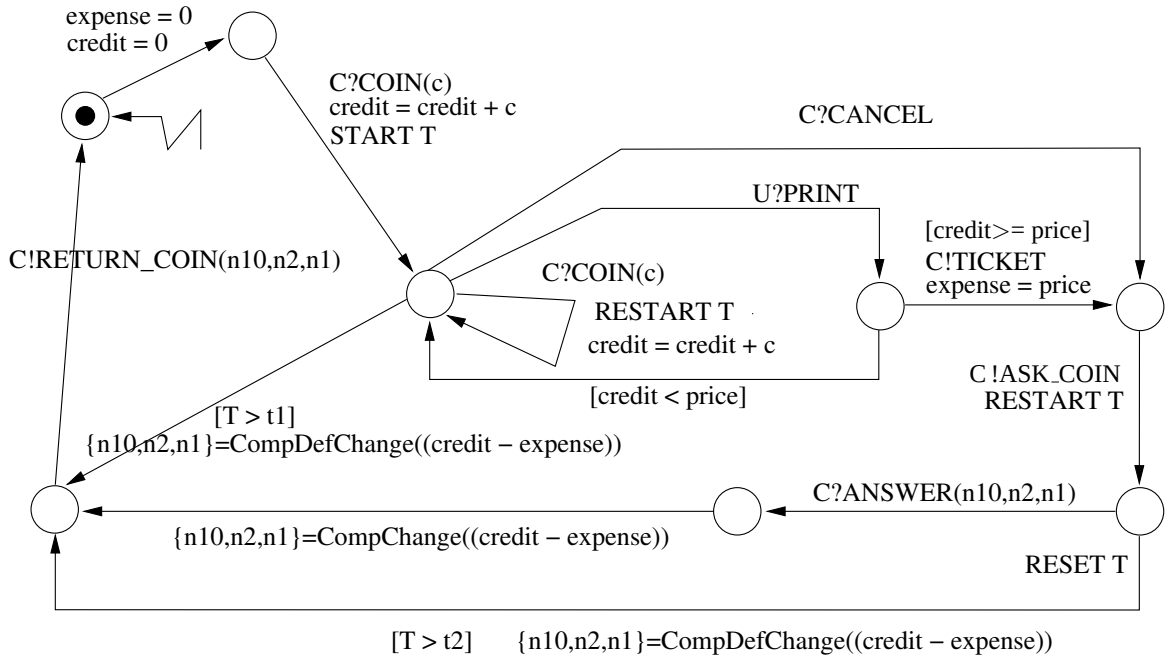

FIG. 8.8 – Représentation d'une implantation pour le contrôleur de la Machine Ticket.

Dans cette implantation l'action START T permet de déclencher un "timer" T. L'action RESTART réarme le "timer" T. Dans ce programme la fonction de calcul CompDefChange() est déclenchée après l'évaluation des gardes [T > t1] ou [T > t2] à vrai.

8.11 Une exécution d'un cas de test de robustesse sur une implantation Java

L'outil Spider permet de tester des implantations écrites en Java. Le testeur de Spider effectue les interactions entre une implantation à tester et le cas de test produit. Pour exécuter cette stimulation, les cas de test doivent dans leur construction contenir des actions de type "Call" et des actions de type "Return". Ces informations sont nécessaires pour le testeur afin de vérifier si une action est bien distribuée entre les composants communicants. À toute action de type "Call" un composant doit répondre par une action de type "Return". Pour faciliter la construction de ces cas de test, il est impératif d'écrire les actions "Call" et "Return" dans le programme IF (spécification). Pour effectuer cette intégration if faut préfixer les signaux par "IF_call" ou "IF_return". Les actions IF_call, sont des actions traduites par le testeur comme un envoi. Le testeur se met, grâce à l'action IF_return en attente d'une réponse "return".

Correspondance des actions IF - java .ima

Dans un programme IF les envois de messages sont uniquement des signaux asynchrones. Dans un programme Java nous ne disposons que des appels aux méthodes publiques. Il faut donc pour tester l'implantation donner une relation entre le format IF et le format Java. Le fichier de correspondance .ima permet la correspondance entre les noms des méthodes du programme Java et les signaux décrits dans la spécification IF. Ce fichier indique également la correspondance entre les classes Java et les processus IF.

NOTE :
Pour identifier les deux informations contenues dans le même fichier de correspondance ({méthodes - signaux}, {classes - processus}), le séparateur $\longleftrightarrow$.

Le fichier a la structure suivante :

```
{action IF}      ⟶   {méthodes java}
...
⟷                                    /* séparateur entre actions et processus */
{processus IF}   ⟶   {classe java}
...
```

Par exemple nous obtenons pour la Machine Ticket

IF_return_COIN	$\longrightarrow$	MachineTicketController.coin.return
IF_return_PRINT	$\longrightarrow$	MachineTicketController.printTicket.return
IF_call_COIN	$\longrightarrow$	MachineTicketController.coin.call
IF_call_PRINT	$\longrightarrow$	MachineTicketController.printTicket.call
...		
$\longleftrightarrow$		
{Counter}0	$\longrightarrow$	counter
{Controller}0	$\longrightarrow$	MT
...		

AGEDIS développe un profil XML pour représenter les cas de test abstrait (format ats). Ce profil est destiné à formaliser l'échange avec d'autres outils de génération et d'exécution de cas de test. Le fichier de cas de test au format ats se présent de la façon suivante :

Le fichier contient toujours un préambule de date :

```
< ?xml version="1.0" encoding="UTF-8" ?>
<testSuite>
<abstractTestSuite generator="TGV" dateTime="2004-07-29T11 :31 :50">
```

Les déclarations et les dépendances sont générées à partir du fichier IF et du fichier de correspondances .ima :

Cette partie du fichier contient la déclaration des classes du programme, des méthodes (member) qui leurs sont associées ainsi que les types non conventionnels utilisés (ici le type data est un entier compris entre 1 et 11). Toutes les méthodes Java sont déclarées ici ainsi que toutes les classes trouvées dans le fichier ima.

```
<model>
   <class name="MachineTicketController">
      <types>
         <type name="data">
            <range type="int">
               <interval from="1" to="11"/ >
            < /range>
         < /type>
      < /types>
   < /class>
   <class name="MachineTicketController">
      <members>
            <member signature="COIN(data)"/ >
            <member signature="sum() :int"/ >
      < /members>
```

```
   < /class>
   . . .
< /model>
```

Chacune des interactions du cas de test (contrôlable ou observable) est traduite dans le format ats.

```
<testCase id="tc1">
   <step id="T0" nextPass="T1">
      <interaction object="MachineTicketController" signature="COIN(data)" type="call">
         <value>1< /value>
      < /interaction>
   < /step>
   <step id="T1" nextPass="T2">
      <interaction object="MachineTicketController" signature="COIN(data)" type="return">
      < /interaction>
   < /step>
```

8.11.1 Les verdicts

Pour nous permettre d'obtenir les verdicts d'exécution, nous décrivons un proxy Java. Ce proxy a pour but de fixer les valeurs des compteurs Cpt_l et Cpt_u avant exécution, de les faire évoluer et de les rendre accessibles pendant l'exécution du cas de test. Les compteurs Cpt_l et Cpt_u sont incrémentés suivant les interactions entre le proxy et le testeur. Les compteurs sont augmentés à chaque fois qu'un état distingué $\in$ état_U ou état_L est atteint pendant l'exécution du cas de test. Nous passons par ces compteurs pour évaluer le nombre de fois que nous passons dans un état distingué. Nous rappelons que si $Cpt_l > c_l$ et que $Cpt_u < c_u$, alors l'observation conduit au verdict **Non Robuste** pour l'implantation testée. Dans tous les autres cas, si les valeurs des compteurs ne respectent pas et $Cpt_l > c_l$ et $Cpt_u < c_u$ alors, cela conduit à une observation **Robuste** pour l'implantation testée (selon le cas de test exécuté) (définition de la section 5.9).

Dans la pratique, le proxy Java est formé d'une classe "counter". Cette classe comporte les méthodes Inc_L et Inc_U qui permettent d'incrémenter les compteurs Cpt_l et Cpt_u. Le principe de Inc_L et Inc_U est qu'à chaque fois qu'un état_L ou état_U est visité, alors, il est ajouté +1 au compteur Cpt_l ou Cpt_u correspondant. La classe "counter" comporte également les méthodes Limit_L et Limit_U qui permettent de fixer les paramètres (seuils) des compteurs Cpt_l et Cpt_u par les valeurs c_l et c_u souhaitées (en relation avec la propriété donnée et donc du cas de test exécuté).

8.11.2 Les résultats d'exécution

Pour les cas de test produits, l'implantation élaborée "robuste" peut émettre le signal C !RETURN_COIN(n10,n2,n1). Il est émis, soit parceque T>t1 est vrai et donc l'action CompDefChange est possible, soit parceque T2>t2 est vrai et donc l'action CompDefChange est également possible. Dans ces cas, l'action CompDefChange permet de progresser au niveau de l'implantation

et donc ne permet pas de rester dans un état_L (prévu par les séquences des cas de test produits). L'observation des compteurs Cpt_l et Cpt_u permet de constater que l'implantation est **Robuste** vis-à-vis de la propriété donnée, car Cpt_l ne pourra pas dépasser un certain seuil c_l (par exemple c_l = 10).

Dans le cas où nous testons une implantation plus naïve (c'est à dire qu'elle représente exactement la spécification avant mutation) l'observation conclue qu'elle n'est pas **Robuste** pour la propriété donnée. En fait, pendant l'exécution des cas de test, la propriété n'est pas vérifiée. Ce constat est établi en fonction des valeurs des c_l et c_u. Il sera possible pour une exécution que $Cpt_l > c_l$ ou que $Cpt_u < c_u$ et donc de conclure que l'implantation testée est **Non Robuste** (pour les valeurs des paramètres c_l et c_u fixés).

8.12 Conclusion

Nous venons de présenter une chaîne d'outils permettant de générer et d'exécuter un cas de test de robustesse. Cet outil applique les étapes de construction décrites dans les parties précédentes. Cette chaîne nous donne la possibilité d'adapter certains outils existants dans le but de réaliser une exécution des cas de test produits. Le prototype permet en partant d'une spécification (même décrite dans un langage de haut niveau) et d'un modèle de faute, une exécution d'un cas de test pour vérifier la condition de robustesse d'une implantation (Java) vis-à-vis d'une propriété de robustesse donnée.

Conclusion

Contribution de notre travail :

Dans cette thèse, nous avons proposé d'étendre une approche formelle, basée sur les modèles du test de conformité, pour valider des propriétés de robustesse. Le test de conformité introduit reprend les principes de génération de cas de test de l'outil TGV [JJ02] avec la relation de conformité **ioco** [Tre96]. Les modèles sous-jacents utilisés pour décrire les spécifications sont les systèmes de transitions étiquetées à entrées-sorties (IOLTS). Et, les modèles proposés pour décrire les propriétés (de robustesse) sont les automates de Rabin paramétrés. Pour réaliser la méthode formelle de génération de cas de test de robustesse, nous avons d'abord proposé, une méthode formelle pour le test de propriétés basée sur les modèles.

Pour le test de propriétés, nous avons défini une relation de satisfiabilité entre les implantations et les propriétés, une méthode de construction d'un graphe de test et un algorithme de sélection de cas de test à partir du graphe de test construit.

Pour la méthode de test de robustesse, nous avons introduit la notion de fautes (aléas). Pour manipuler les fautes, nous avons défini la notion d'environnement dégradé en opposition avec la notion d'environnement nominal. Puis, pour intégrer les fautes aux modèles de la spécification, nous avons proposé une fonction de mutation.

Pour valider la méthode de génération de cas de test de robustesse un prototype a été conçu.

Une première contribution : Le test de propriétés.

L'idée du test de propriétés est de tester si une implantation satisfait (préserve) des propriétés données. Il s'agissait de mettre en place une méthode pour qu'un utilisateur puisse tester plusieurs propriétés pour une même implantation.

Dans notre contexte, le modèle de l'implantation n'est pas connu, et cela, nous interdit de tester directement les propriétés avec des techniques de model-checking. Nous proposons alors, (pour exécuter des tests) que les implantations à tester soient du domaine des systèmes réactifs, et plus particulièrement du domaine des systèmes de communications. Cette proposition, nous permet d'utiliser une méthode de test de type boîte noire, où les tests sont basés uniquement sur les interactions entre les entrées et les sorties des systèmes (interfaces visibles des systèmes testés).

Les ingrédients de base de la méthode de test de propriétés sont : une (des) propriété(s), une

spécification (modèle formel représentant des comportements de l'implantation), une architecture de test (ensemble des actions visibles, interfaces des implantations testées), et donc une implantation.

Pour tester une implantation, nous avons besoin d'exécuter une séquence d'actions par un testeur. Pour construire une telle séquence (un cas de test), nous effectuons un produit synchrone étendu entre les propriétés et la spécification puis, une sélection parmi le résultat du produit. Enfin, pour valider un test, nous appliquons une relation de satisfiabilité.

La relation de satisfiabilité : La préservation d'une propriété est traduite par la relation de satisfiabilité établi entre une propriété $\mathcal{P}$ et une implantation IUT : IUT $\models \mathcal{P}$. Si cette relation n'est pas préservée, alors, l'implantation IUT ne satisfait pas (par son comportement) la propriété P.

Dans notre méthode de génération de cas de test, nous représentons par un observateur $\mathcal{O}$bs (automate de Rabin paramétré) la négation de la propriété P. Dans ce contexte, l'observateur $\mathcal{O}$bs identifie exactement les traces de $\neg\mathcal{P}$.

La relation de satisfiabilité s'exprime alors par $\mathcal{L}(\Delta(IUT))\downarrow \mathcal{A}_{\mathcal{O}} \cap \mathcal{L}(\mathcal{O}\text{bs}) = \emptyset$. En d'autres termes, si IUT $\models$ P, alors aucune trace de l'IUT n'est reconnue par l'observateur $\mathcal{O}$bs. Ainsi, à la description du comportement d'une implantation s'ajoute la notion de correction. La correction est exprimée par une suite de propriétés que doit satisfaire l'implantation.

Les propriétés : Nous proposons de tester des propriétés de "sûreté" ou de "vivacité bornée". De par la nature même du test, nous ne pouvons pas tester des propriétés de "vivacité", car l'exécution d'un test doit être finie. Nous proposons dans la méthode de borner l'exécution d'une propriété de "vivacité". Pour cela, nous intégrons au modèle décrivant les propriétés des paramètres définissant ainsi les propriétés de "vivacité bornée". Il existe de nombreuses manières pour modéliser de telles propriétés, mais, quel que soit le formalisme utilisé, il doit permettre de décider si cette propriété est satisfaite pour les différents comportements d'exécution de l'implantation. Nous considérons enfin des propriétés linéaires, car la vérification peut se faire sur les traces d'exécution.

Dans la méthode du test de conformité basée sur les modèles, l'ensemble des comportements est représenté par un graphe d'état. Nous proposons de modéliser nos propriétés par un automate de Rabin paramétré décrivant un observateur = ($\mathcal{O}$bs, $\mathcal{T}_{\mathcal{O}\text{bs}}$, $\mathcal{C}_{\mathcal{O}\text{bs}}$), où $\mathcal{O}$bs est un IOLTS, $\mathcal{T}_{\mathcal{O}\text{bs}}$ est un ensemble contenant des états distingués U et L de $\mathcal{O}$bs et $\mathcal{C}_{\mathcal{O}\text{bs}}$ est l'ensemble des couples de paramètres liés aux états distingués. Nous précisons que l'observateur représente le comportement de la négation de la propriété et sert d'oracle pour le test. Le choix de fixer l'observateur comme la négation de la propriété est arbitraire, mais, il nous a permis de décrire les verdicts et d'analyser les résultats d'observations. En d'autres termes, le choix de fixer $\mathcal{O}$bs avec ¬P nous paraissait plus évident pour générer, sélectionner et l'analyser les tests.

Construction de cas de test de propriétés : Les implantations testées sont des systèmes réactifs, et comme le test de propriétés va s'effectuer sur les modèles, nous avons proposé de construire des cas de test étendus avec la notion de verdict.

Un cas de test est construit avec, une architecture de test, une spécification et un observateur. L'ar-

chitecture de test fournit au testeur les actions visibles à tester et permet de réduire les modèles initiaux (aux actions visibles). La spécification propose certains comportements de l'implantation. L'observateur représente la négation de la propriété.

Un cas de test est un sous-graphe extrait d'un graphe de test (automate de Rabin paramétré). Le graphe de test GT est construit avec la spécification (déterministe, minimale et contenant éventuellement des blocages explicités avec l'action $!\delta$) et l'observateur. Le produit utilisé est un produit synchrone étendu permettant de garder, dans le graphe de test, tous les comportements de la spécification augmentés des comportements de la propriété (si ceux-ci ne sont pas déjà contenus dans les comportements de la spécification). Le graphe de test possède un marquage désignant des états paramétrés hérités du modèle de l'observateur.

Pour extraire les cas de test CT ("sous-graphe contrôlable" de GT), nous effectuons une sélection à partir de GT. Un cas de test contient au moins une séquence de $\mathcal{L}$(GT), et par conséquent de $\mathcal{L}(\mathcal{O})$. Clairement, pour appartenir à $\mathcal{L}$(GT), une séquence d'exécution doit comporter une composante fortement connexe avec un état de l'ensemble distingué L^{GT}. Réciproquement, aucune séquence de GT ne menant pas à une composante connexe avec un état de L^{GT} ne peut appartenir à $\mathcal{L}$(GT). Par conséquent, l'algorithme de sélection proposé, définit un graphe de test GT en utilisant le prédicat L2L (pour *"leads to L"*). Le prédicat L2L dénote l'ensemble des états menant à une composante fortement connexe comportant au moins un état de L. Puis, nous définissons un sous-ensemble de relation T_{GT}, contrôlable, contenant au moins une séquence de $\mathcal{L}(\mathcal{O})$. Ce sous-ensemble contient, les transitions non contrôlables de T_{GT} (marqué par un élément de A_u), et tout au plus une transition contrôlable (aléatoirement choisie) de T_{GT} menant à un état de L2L si plusieurs transitions existent (pour atteindre un état de GT).

Le sous-ensemble de T_{GT} est ensuite étendu avec toutes les actions (non contrôlables) de A_u qui ne sont pas explicitement dans T_{GT} (en s'assurant que l'exécution d'un cas de test de propriétés ne sera jamais arrêtée lors d'une réception d'un événement inattendu de l'implantation). Toutes ces constructions permettent d'obtenir un cas de test étendu avec la notion de verdicts.

Modèle d'exécution et verdicts de cas de test de propriétés : Les exécutions, des cas de test de propriétés, sont supposées délivrer un certain nombre de verdicts. Nous définissons, σ, une séquence d'exécution d'un cas de test (CT) sur une implantation à tester (IUT). La séquence σ est perçue comme une composition parallèle entre l'IUT et CT, avec des synchronisations sur les actions des ensembles A_c et A_u (ensembles données par l'architecture de test). Les verdicts sont formalisés par une fonction **Verdict** sur les séquences d'exécution de la composition parallèle CT $||$ Δ(IUT). Nous affectons, à chaque état $q \in$ Chemin(σ) et appartenant aux ensembles distingués U^{CT} ou L^{CT} de CT, une fonction λ comptant le nombre de fois qu'un état $q_i \in L^{\mathrm{CT}}$ ou qu'un état $q_i \in U^{\mathrm{CT}}$ est visité pendant une exécution du cas de test CT (avec i $\in \mathbb{N}$).

L'implantation est déclarée "non satisfaite" si nous obtenons : $\mathcal{L}(\Delta(IUT))\downarrow \mathcal{A}_{\mathcal{O}} \cap \mathcal{L}(\mathcal{O}\mathrm{bs}) \neq \emptyset$. Ou, selon les conditions de la fonction λ associée aux états, le nombre de fois qu'un état $q \in$ L est visité soit supérieur à son paramètre c_l et que le nombre de fois qu'un état $q \in$ U est visité soit inférieur ou égal à son paramètre c_u. Le cas de test est instancié pour garder une exécution de test finie (contennant les bornes c_u et c_l des états distingués). L'observation d'une exécution est donnée en relation des verdicts "*Ne Satisfait Pas*", ou "*Satisfait*" provenant des paramètres du cas de test. Une implantation est rejetée (resp. acceptée) par un cas de test produit, si il existe une (resp. toutes)

séquence d'exécution σ de l'implantation qui ne satisfait pas (resp. satisfait) la propriété et dont $\sigma \in$ (CT $||$ $\Delta(IUT)$).

Une deuxième contribution : Le test de robustesse

L'objectif de la méthode est de prendre en compte dans les modèles de construction les aléas extérieurs de l'implantation. Nous définissons les aléas comme des erreurs, des pannes ou des changements d'environnement. Classés par un modèle de faute, ces aléas sont ensuite intégrés dans les spécifications.

Le but du test de robustesse est de tester si une implantation garde toujours un comportement de fonctionnement acceptable, en dépit de fautes. Un comportement est acceptable si les propriétés de robustesse sont préservées par les implantations (c'est à dire qu'elles respectent la relation de satisfiabilité vue pour le test de propriétés). En complément d'autres techniques, comme l'injection de fautes, la contribution de cette méthode est de pouvoir guider des tests dans le but de vérifier la robustesse d'une implantation. La génération de cas de test s'effectue avec une spécification et une propriété de robustesse. L'exécution d'un cas de test est faite sur l'IUT à l'aide d'un testeur.

Les fautes et la mutation : Pour intégrer la notion de robustesse d'un système, il est nécessaire de parler de notion de faute et de définir l'environnement dégradé des systèmes. Le travail, contenu dans cette thèse, a été d'apporter une définition des fautes possibles (aléas internes ou externes au système), pouvant intervenir pendant l'exécution d'un système. Nous proposons de classer les fautes, de définir la notion d'environnement nominal et de définir la notion d'environnement dégradé (pour un IOLTS).

Les fautes ont tout d'abord été décrites selon les actions visibles au test, c'est à dire les actions d'entrée et de sortie des IOLTS (les actions internes ne pouvant pas être testées avec notre méthode). Nous avons défini, sous la forme de tableaux, des fautes, d'élargissements des données, de dycfonctionnements, de coupures et des pannes.

Pour intégrer une faute au modèle de la spécification IOLTS, une fonction de mutation *Apply* a donc été proposée. Définie sur une grammaire des modèles des systèmes de transitions, la fonction de mutation transforme les actions existantes d'un modèle IOLTS par de nouvelles actions données en fonction du tableau de faute. Pour effectuer chaque type de mutation, une fonction a été décrite.

En comparaison, avec d'autres techniques existantes pour tester la robustesse des implantations (comme par exemple celles basées sur l'injection de fautes), le principal avantage de notre approche est de mieux cibler les cas de test par rapport aux conditions de robustesse prévues. Les fautes particulières sont injectées par le testeur seulement quand elles sont nécessaires, car la génération (construction d'un graphe de test puis application de la fonction de sélection) des cas de test est guidée par les actions de la spécification. Un cas de test est généralement préfixé par les actions de la spécification puis contient les actions de la propriété à satisfaire. Cependant, cette approche reste efficace si il existe au moins une spécification nominale formelle (même partielle). La spécification est utile pour guider la génération des tests et doit décrire quels sont les scénarii d'exécution prévus par l'implantation à tester. Une spécification peut être exprimée en langage de haut niveau (UML, LOTOS, . . .) puis traduite ensuite en IOLTS.

Construction de cas de test de robustesse : Initialement nous disposons de la spécification, du modèle de faute, de l'observateur, d'une architecture de test et de l'implantation.

Nous construisons une spécification mutée par transformations syntaxiques en intégrant au modèle de la spécification initiale les aléas contenus dans le modèle de faute.

Le graphe de test est construit à partir de la spécification mutée (déterminisée, minimale en tenant compte des éventuels blocages) et de l'observateur. Le produit reprend les règles de construction du produit synchrone étendu vu pour le test de propriétés. Où, la spécification sert de guide de construction et permet d'intégrer dans le graphe de test les comportements accédant aux actions à tester et permet éventuellement à l'observateur d'introduire certaines actions manquantes (dans la spécification) pour respecter la propriété donnée. Les automates représentant les observateurs sont des automates de Rabin paramétrés complets en entrées et en sorties.

La phase de sélection pour extraire les cas de test du graphe reste celle décrite pour la méthode de test de propriétés.

Pour analyser les exécutions des cas de test sur les implantations, le testeur délivre deux verdicts possibles "Robuste" et "Non Robuste" en relation avec les paramètres proposés par l'observateur. Dans la pratique, pour obtenir les verdicts de robustesse, il faut que les compteurs λ associés aux états distingués respectent les valeurs c_l et c_u fixées initialement.

Les verdicts du test de robustesse :

1. **Robuste** : Il indique une violation d'une condition de robustesse $\Longrightarrow$ le compteur des état_U est inférieur à la valeur fixée et le compteur des état_L est supérieur à la valeur fixée.

2. **Non Robuste** : Il indique que pour le cas de test exécuté, la propriété est préservée par le comportement de l'implantation $\Longrightarrow$ le compteur des état_U est supérieur à la valeur fixée ou le compteur des état_L est inférieur à la valeur fixée.

Exécution des cas de test de robustesse : En complément de la génération, nous avons proposé une exécution des cas de test sur des implantations écrites en Java. Cette proposition utilise et adapte des outils existants. Pour ce processus, il faut transformer les formats du cas de test, puis ajouter à l'implantation les valeurs des compteurs (nommé proxy Java). Le testeur exécute le cas de test sur l'implantation, met à jour les compteurs et délivre un verdict à la fin de l'exécution. Cette technique a été appliquée pour proposer une chaîne d'outils complète (intégrant la génération de cas de test et la phase d'exécution de cas de test de robustesse).

Outil pour le test de robustesse : La chaîne d'outils proposée est conçue avec des outils existants (adaptés si besoin) et nos algorithmes. L'algorithme de construction du graphe de test est présenté comme un produit synchrone étendu. Où, il reprend les principes des produits de composition parallèle en gardant les actions synchrones entre la spécification de référence et l'observateur. Puis, il étend chaque comportement de la spécification avec ceux de la propriété donnée (si ils ne sont pas déjà inclus dans la spécification). L'algorithme de sélection s'appuie sur les marquages du

graphe de test (issues des marquages de l'observateur), pour extraire un cas de test. L'algorithme est composé d'une recherche de boucle d'état_L, puis d'une reconstitution des chemins élémentaires menant à cette boucle. Si nous disposons de plusieurs chemins menant à une boucle une étape de contrôlabilité peut être appliquée. Lorsque nous avons le choix entre une action observable et une action contrôlable, nous sélectionnons l'action contrôlable. Nous gardons les actions observables lorsqu'elles ne sont pas couplées avec une action contrôlable. Dans cette phase de sélection, si nous avons le choix entre plusieurs actions contrôlables cela nous donne plusieurs cas de test différents. Il y a autant de cas de test que d'actions contrôlables partant d'un même état.

Perspectives

Un certain nombre de perspectives peuvent être envisagées pour étendre le travail qui a été présenté dans ce document.

Tout d'abord, l'approche que nous avons proposée pour produire des tests de robustesse mériterait d'être évaluée sur davantage d'études de cas, et notamment sur des études de cas plus conséquentes. En effet, même si la correction de cette méthode a pu être établie sur le plan théorique (les suites de tests produites ne rejettent que des implantations non robustes), certains aspects pratiques peuvent encore être améliorés pour les rendre plus accessibles à un utilisateur final. Nous pensons notamment à la description du modèle de fautes (pour lequel nous pourrions proposer un langage de mutations de plus haut niveau), ainsi qu'à l'expression de la propriété de robustesse (écrire un observateur complet par rapport à l'architecture de test s'avère assez vite fastidieux, un langage de "patrons" de propriétés serait là-aussi souhaitable). De même, la fonction de sélection des cas de test que nous avons implémentée est encore assez naïve. Un travail d'expérimentation plus large pourrait sans doute permettre de déterminer un certain nombre d'heuristiques destinées à sélectionner en priorité des cas de test les plus "aggressifs" vis-à-vis de la propriété de robustesse considérée (en exploitant par exemple davantage le modèle de faute).

Dans le même ordre d'idée l'implémentation qui a été réalisée n'est encore qu'un prototype. Confronter cette implémentation à des études de cas de plus grande taille permettrait de déterminer les parties algorithmiques les plus coûteuses en pratique et sur lesquelles des améliorations seraient souhaitables pour faciliter le passage à l'échelle. Dans cet ordre d'idée, un point particulier concerne l'opération de mutation appliquée à la spécification pour prendre en compte le modèle de faute. Dans la solution actuelle, cette mutation est appliquée de manière globale, indépendamment de la propriété de robustesse considérée. Un risque potentiel est donc d'obtenir après mutation un modèle de spécification de très grande taille, sur lequel la construction du graphe de test s'avérera très coûteuse. Une perspective envisageable consisterait ici à *raffiner* au préalable ce modèle de fautes en fonction de cette propriété et de la spécification. Concrètement il s'agirait de déterminer statiquement (en appliquant des techniques classiques de *slicing* ou de calcul de *cône d'influence*) quels sont les impacts du modèle de faute sur les comportements de la spécification qui sont susceptibles d'influer sur la validité de la propriété de robustesse. L'intérêt est de permettre ainsi de ne muter que des parties *utiles* de la spécification, en limitant ainsi le phénomème d'explosion d'états du modèle. Cette approche (schématisée sur la figure C.9) s'inspire des techniques de réduction sta-

tiques de modèles utilisées en vérification par model-checking (la différence étant qu'ici la réduction porte sur un couple modèle de faute et spécification).

Une autre perspective envisageable consiste à appliquer cette technique de génération de test à d'autres types de propriétés de robustesse, plus spécifiques. Il peut s'agir par exemple de propriétés portant sur les temps de réponse de l'implantation, ou sur sa capacité à résister à une augmentation de la fréquence de requêtes envoyées en entrée à un élément du système (résistance au *stress*). L'utilisation de modèles temporisés (comme c'est le cas avec le langage IF) pour décrire la spécification permet en effet de prendre en compte cette classe de propriétés, et de modéliser cette forme de stress par un opérateur de mutation (en modifiant l'horloge qui rythme la fréquence nominale des entrées). Dans le cas d'une propriété temporisée (exprimée par un observateur temporisée) la technique de génération de test à partir de la spécification mutée devra alors reposer sur des techniques adaptées à ce type de test. De manière similaire, nous pouvons également envisager d'appliquer cette technique à du test de politiques (ou tout au moins de propriétés) de *sécurité*. Il s'agirait ici d'exprimer les propriétés de sécurité par un observateur, et d'intégrer au modèle de faute l'ensemble des *attaques* contre lesquelles nous souhaitons nous prémunir (modifications intempestives de données sensibles du système, comme l'en-tête d'un message de contrôle, ou le contenu d'une mémoire interne). Si ces scénarii d'attaque peuvent être exprimés par mutation syntaxique de la spécification alors la technique, que nous avons proposée, peut permettre de générer des tests susceptibles de mettre en défaut des implantations (une difficulté sera alors de disposer d'une architecture de test suffisante pour permettre leur exécution).

Enfin, l'approche de génération de test de robustesse développée dans ce document a été présentée comme une alternative aux techniques de test basées sur de l'injection de fautes. Il pourrait être intéressant d'envisager également comment une *combinaison* entre ces deux approches pourrait renforcer leur efficacité mutuelle, par exemple en *pilotant* un dispositif d'injection de fautes avec des tests produits à partir d'une spécification comportementale.

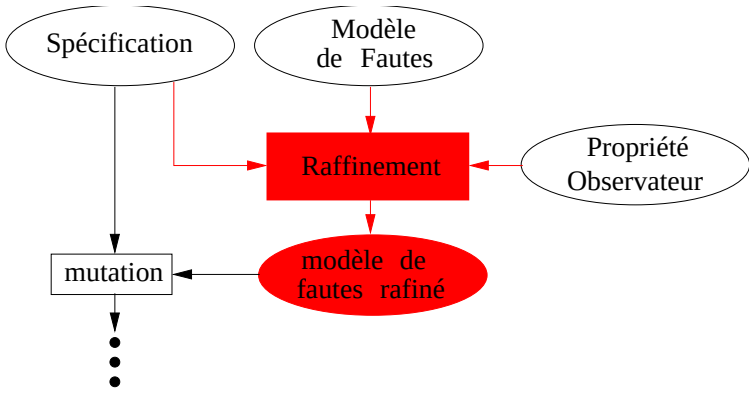

FIG. C.9 – Perspective de raffinement des modèles de faute

Bibliographie

[ABM97] L. Van Aertryck, M. Benveniste, and Metayer. CASTING : A formally based software test generation method. In *ICFEM'97, First IEEE International Conference on Formal Engineering Methods*, Hiroshima, Japon, November 1997. 2.3

[Abr87] S. Abramsky. Observational Equivalence as a Testing Equivalence. *Theoretical Computer Science*, 53(3), 1987. 2.2

[ABS94] T. M. Austin, S. E. Breach, and G. S. Sohi. Efficient Detection of All Pointer and Array Access Errors. In *SIGPLAN Conference on Programming Language Design and Implementation*, pages 290–301, 1994. 4

[Aer98] L. Van Aertryck. *Une méthode et un outil pour l'aide à la génération de jeux de tests de logiciels.* Thèse de doctorat, Université de Rennes I, January 1998. 2.3

[AFRS02] Jean Arlat, Jean-Charles Fabre, Manuel Rodríguez, and Frédéric Salles. Dependability of COTS microkernel-based systems. *IEEE Transactions on Computers*, 2002. 4, 6.1

[AML99] A. Arnould, B. Marre, and P. LeGall. Génération automatique de tests à partir de spécifications de structures de données bornées. *Technique et Science informatique*, 19(3) :297 – 321, 1999. 4

[ASU86] A. Aho, R. Sethi, and J. Ullman. *Compilers Principes, techniques and tools*. Addison-Wesley Publishing Company, 1986. ISBN 0-201-10088-6. 2.3

[BA85] T. A. Budd and G. S. Ajei. Program testing by specification mutation. *Computer Languages*, 10(1) :63–73, 1985. 6.5.1

[Bak97] S. Baker. *CORBA Distributed Objects - Using Orbix*. ACM Press, Adison Wesley, 1997. 2.4

[BB88] T. Bolognesi and E. Brinksma. Introduction to the ISO Specification Language LOTOS. *ISDN*, 14(1) :25–29, January 1988. I.1, 2.4

[BD88] S. Budkowski and P. Dembinski. An Introduction to ESTELLE : A Specification Language for Distributed Systems. *ISDN*, 14, January 1988. I.1

[BDD+91] G. Von Bochmann, A. Das R. Dssouli, M. Dubuc, A. Ghedamsi, and G. Luo. Fault models in testing. In *IFIP TC6/WG6.1 Fourth International Workshop on Protocol Test Systems IV table of contents*, pages 17 – 30. North-Holland, 1991. ISBN :0-444-89517-5. 4, 2

[BFdV+99] A. Belinfante, J. Feenstra, R. G. de Vries, J. Tretmans, Nicolae Goga, Loe M. G. Feijs, Sjouke Mauw, and Lex Heerink. Formal Test Automation : A Simple Experiment. In *IWTCS*, pages 179–196, 1999. 3, 3.7

[BFG99a] M. Bozga, J-C. Fernandez, and L. Ghirvu. State Space Reduction based on Live Varibles Analysis. In *SAS*, volume 1694, Venezia, 1999. LNCS. 1, 8.3

[BFG+99b] 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 *R. Dssouli, G. Bochmann, and Y. Lahav, editors, Proceedings of SDL FORUM'99.(Montreal, Canada),Elsevier*, pages 423–440, June 1999. 8.3

[BFG00] M. Bozga, J-C. Fernandez, and L. Ghirvu. Using Static Analysis to Improve Automatic Test Generation. In *Tools and Algorithms for Construction and Analysis of Systems*, pages 235–250, 2000. 1, 2, 8.3

[BGM91] G. Bernot, M. C. Gaudel, and B. Marre. Software testing based on formal specifications : a theory and a tool. *Software Engineering Journal*, 6(6) :387–405, November 1991. 3.5

[BGM02] M. Bozga, S. Graf, and L. Mounier. If-2.0 : A validation environment for component-based real-time systems. In K.G. Larsen Ed Brinksma, editor, *Proceedings of CAV'02 (Copenhagen, Denmark)*, volume 2404 of *LNCS*, pages 343–348. Springer-Verlag, July 2002. 2a, 7.2.1

[BGMO04] M. Bozga, S. Graf, L. Mounier, and I. Ober. IF tutorial. Presented at the 9th SPIN'04 Workshop on Model-Checking of Software, Barcelona, Spain, April 2004. 8.3

[BGMS98] M. Bozga, S. Graf, L. Mounier, and J. Sifakis. The Intermediate Representation IF. (Internal document, don't distribute), 1998. 2.4, 1, 8.3

[BGOS04] M. Bozga, S. Graf, I. Ober, and J. Sifakis. Tools and applications : the if toolset. In M. Bernanrdo and F. Corradini, editors, *Proceedings of the 4th International School on Formal Methods for the Design of Computer, Communication and Software Systems : Real Time, SFM-04 :RT*, volume 3185 of *LNCS*. Springer, 2004. 8.3

[BME98] A. Basu, G. Morrisett, and T. Eicken. Promela++ : A language for Constructing Correct and Efficient Protocols, jan 1998. 2.4

[Boz99] M. Bozga. *Vérification symbolique pour les protocoles de communication.* PhD thesis, Verimag Grenoble - France, December 1999. 2.4, 1, 8.3

[BP94] G. V. Bochmann and A. Petrenko. Protocol testing : Review of methods and relevance for software testing. In Thomas Ostrand, editor, *Proceedings of the 1994 International Symposium on Software Testing and Analysis (ISSTA)*, pages 109–124, 1994. 1

[Bri88] E. Brinksma. A theory for the derivation of tests. In S. Aggarval and K. Sabnani, editors, *Protocol Specification, Testing and Verification VIII, IFIP*, pages 63–74. Elsevier Science Publishers, B.V., North-Holland, 1988. 2.2

[BT97] D.M. Blough and T. Torii. Fault-Injection-Based Testing of Fault-Tolerant Algorithms for Message-Passing Parallel Computers. In *Digest of the 27th IEEE International Symposium on Fault-Tolerant Computing (FTCS-27), Seattle, WA, USA, IEEE CS Press*, 1997. 6.1

[BT01] Ed. Brinksma and J. Tretmans. Testing transition systems : Annotated bibliography. In *F Cassez, C Jard, B Rozoy, M D Ryan (Eds) Modeling and Verification of Parallel Processes*, volume 2067 of *LNCS*, pages 196–205, 2001. 2

[Bud80] T. A. Budd. *Mutation analysis of program test data*. PhD thesis, Yale University, New Haven , CT , USA, 1980. 6.5.1

[CFP93] A. R. Cavalli, J. P. Favreau, and M. Phalippou. Formal Methods for Conformance Testing : Results and Perspectives. In *Protocol Test Systems*, pages 3–17, 1993. 2.2

[Cho78] T. S. Chow. Testing software design modeled by finite-state machines. *IEEE Trans. Software Eng*, 4(3) :178–187, 1978. 1

[CJRZ01] D. Clarke, T. Jéron, V. Rusu, and E. Zinovieva. Stg : A Tool for Generating Symbolic test programs and oracles from operational specifications. In *Joint 8th European Software Engineering Conference (ESEC) and 9th ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE-9)*, 2001. 2.4

[CJRZ02] D. Clarke, T. Jéron, V. Rusu, and E. Zinovieva. STG : A symbolic Test Generation Tool. *Lecture Notes in Computer Science*, 2280 :470–482, 2002. 2.4

[COR] CORBA. Conformance test suite, tip - the competence centre for testing, interoperability and performance. 2.4

[C.P03] C.Pachon. Une approche pour la génération automatique de tests de robustesse. In *1ere MAnifestation des JEunes Chercheurs du domaine des STIC, MAJECSTIC'03, Marseille*, Octobre 2003. 6.1

[C.P04] C.Pachon. Une approche pour la génération automatique de tests de robustesse. *ISDM : InfoComm Sciences for Decision Making : Permanent online journal of Information and Communication Technologies*, 13(119) :167–175, Janvier 2004. 6.1

[CS00] C. Csallner and Y. Smaragdakis. JCrasher : an automatic robustness tester for Java. *Software - Practice and Experiece*, 1(7), 2000. 4

[CVWY92] C. Courcoubetis, M. Y. Vardi, P. Wolper, and M. Yannakakis. Memory-efficient algorithms for the verification of temporal properties. *Formal Methods in System Design*, 1(2/3) :275–288, 1992. (document), 5.3, 7.2.3, 9, 7.2.3, 7.2.3

[dBORZ99] L. du Bousquet, F. Ouabdesselam, J-L. Richier, and N. Zuanon. Lutess : A specification-driven testing environment for synchronous software. In *International Conference on Software Engineering*, pages 267–276, 1999. 2.3

[DGK+88] R. A. DeMillo, D. S. Guindi, K. N. King, W. M. McCracken, and A. J. Offutt. An extended overview of the Mothra software testing environment. In *Proceedings of the Second Workshop on Testing, Analysis, and Verification*, pages 142–151. IEEE Computer Society Press, 1988. 6.5.1

[DJMT96] Scott Dawson, Farnam Jahanian, Todd Mitton, and Teck-Lee Tung. Testing of Fault-Tolerant and Real-Time Distributed Systems via Protocol Fault injection. In *Symposium on Fault-Tolerant Computing*, pages 404–414, 1996. 6.1

[DKPR95] R. Dssouli, K.Karoui, A. Petrenko, and O. Rafiq. Towards Testable Communication Software. In *IWPTS'95,Paris*, pages 239–244, 1995. 2.5

[DMM96] M. Delamaro, J. Maldonado, and A. Mathur. Integration testing using interface mutations. In New York IEEE Computer Society Press, White Plains, editor, *Proceedings of the Seventh International Symposium on Software Reliability Engineering*, pages 112–121, 1996. 6.5.1

[DT02] P. Deussen and S. Tobies. Formal test purposes and the validity of test cases. In *22nd International Conference on Formal Techniques for Networked and Distributed Systems*, 2002. 3.4.3

[EFLR99] A. Evans, R. France, K. Lano, and B. Rumpe. The UML as a Formal Modeling Notation. In Jean Bézivin and Pierre-Alain Muller, editors, *The Unified Modeling Language, UML'98,Beyond the Notation*, volume 1618, pages 336–348. Springer, 1999. 2.4

[Fer88] J-C. Fernandez. *ALDEBARAN : Un systéme de Vérification par Réduction de Processus Communicants*. PhD thesis, Université Joseph Fourier de Grenoble, May 1988. 2.4, 7.2.1, 7.2.1, 4

[FHP02] E. Farchi, A. Hartman, and S. S. Pinter. Using a model-based test generator to test for standard conformance. *IBM Systems Journal*, 41(1) :89–110, 2002. 2.4

[FJJV96] J-C. Fernandez, C. Jard, T. Jeron, and Cesar Viho. Using On-The-Fly Verification Techniques for the Generation of Test Suites. In *Computer Aided Verification*, pages 348–359, 1996. 2.3, 2, 3, 3.4, 3.4.3

[FJJV97] J-C. Fernandez, C. Jard, T. Jeron, and C. Viho. An Experiment in Automatic Generation of Conformance Test Suites for Protocols with Verification Vechnology. *Science of Computer Programming*, 29 :123–146, 1997. 2.5

[FK96] R. Ferguson and B. Korel. The chaining approach for software test data generation. *ACM Transactions on Software Engineering and Methodology ISSN 1049-331X*, 5(1) :63–86, January 1996. 2.3

[FMP04] J-C. Fernandez, L. Mounier, and C. Pachon. Property oriented test case generation. In Alexandre Petrenko and Andreas Ulrich, editors, *Formal Approaches to Software Testing : Third International Workshop on Formal Approaches to Testing of Software : FATES 2003 : Montréal, Québec, Canada, October 6th, 2003 : Revised Papers*, volume 2931 of *Lecture Notes in Computer Science*, pages 266 – 278, New York, NY, USA, 2004. Springer-Verlag Inc. 5, 5.1

[FMP05] J-C. Fernandez, L. Mounier, and C. Pachon. A model-based approach for robustness testing. In Rachida Dssouli Ferhat Khendek, editor, *Testing of Communicating Systems : 17th IFIP TC6/WG 6.1 International Conference, TestCom 2005, Montreal, Canada, May 31*, volume 3502, pages 333 – 345. Springer-Verlag GmbH, June 2005. 6.1

[Gau95] M-C. Gaudel. Testing can be formal, too. In *TAPSOFT*, pages 82–96, 1995. 2.3

[GG75] J. B. Goodenough and S. L. Gerhart. Toward a theory of test data selection. *IEEE Trans. Software Eng.*, 1(2) :156–173, 1975. 3.4.3

[Ghi02] L. Ghirvu. *Génération automatique de tests de conformité pour les protocoles de télécommunications*. PhD thesis, Université Joseph Fourier, Grenoble, July 2002. 2

[GHNS95] J. Grabowski, D. Hogrefe, I. Nussbaumer, and A. Spichiger. Test Case Specifications Based on mscs and ASN.1. In *Proceeding of 7th. SDL Forum A. S. R. Braek (eds), SDL'95 with MSC in CASE, Noth-Holland*, 1995. 2.4

[GHT95] J. Grabowski, D. Hogrefe, and D. Toggweiler. Partial Order Simulation of SDL Specifications. In *O. Braek and A. Sarma (eds), SDL'95 with MSC in CASE, Noth-Holland*, 1995. 2.4

[GHTS96] J. Grabowski, D. Hogrefe, D. Toggweiler, and R. Scheurer. Dealing with the Complexity of State Space Exploration Algorithms. In *Proceedings of the 6th. GI/ITG technical meeting on 'Formal Description Techniques for Distributed Systems' University of Erlangen*, 1996. 2.4

[Gil62] A. Gill. *Introduction to the Theory of Finite State Machine*. New-York McGraw-Hill, 1962. 1

[Gou97] V. Gouranton. *Dérivation d'analyseurs dynamiques et statiques à partir de spécifications opérationnelles*. PhD thesis, Université de Rennes, 1997. 8.3

[Gra94] J. Grabowski. *Test Case Generation and Test Case Specification with Message Sequence Charts*. PhD thesis, Université of Berne, Institue for Informatics and Applies Mathematics, 1994. 2.4

[GSS98] A. Ghosh, M. Schmid, and V. Shah. Testing the robustness of Windows NT software. In *In Pandroceedings of the Ninth International Symposium on Software Reliability Engineering (ISSRE '98), Los Alamitos, CA, IEEE Computer Society.*, pages 231–235, November 1998. 4

[Hal98] N. Halbwachs. Synchronous programming of reactive systems, a tutorial and commented bibliography. In *Tenth International Conference on Computer-Aided Verification, CAV'98*, Vancouver (B.C.), jun 1998. LNCS 1427, Springer Verlag. 2.3

[HCRP91] N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud. The synchronous dataflow programming language LUSTRE. In *Proc. of IEEE*, volume 79, pages 1305 – 1320, September 1991. 2.3

[HEG98] A. Helmy, D. Estrin, and S. K. S. Gupta. Fault-oriented test generation for multicast routing protocol design. In *Proceedings of the FIP TC6 WG6.1 Joint International Conference on Formal Description Techniques for Distributed Systems and Communication Protocols (FORTE XI) and Protocol Specification, Testing and Verification (PSTV XVIII)*, pages 93–109. Kluwer, B.V., 1998. 4

[Hel97] A. A-G Helmy. *Systematic Testing of Multicast Protocol Robustness*. PhD thesis, University of Southern California, Los Angeles, December 1997. 4

[Hol91] G. J. Holzmann. *Design and Validation of Computer Protocols*. Prentice Hall, Englewood Cliffs, NJ, 1991. 2.4

[HT99] Jt He and Kenneth J. Turner. Protocol-Insptred Hardware Testing. In Sarolta Dibuz Gyula Csopaki and Katalin Tarnay, editors, *Proc. Testing Communicating System XII*, pages pages 131 – 147. Kluwer Academic Publishers, London, UK, September 1999. 2.4

[HU79] J. E. Hopcroft and J. D. Ullman. *Introduction to Automata Theory, Languages, and Computation*. Addison-Wesley, Menlo Park, 1979.
The classic text book on the formal theory of computation. 3.4.1, 2b, 2c, 7.2.1

[IEE90] IEEE. *Standard Glossary of Software Engineering Terminology, IEEE Std 610.12.1990*, 1990. 6.5.3

[IR76] James C. King (IBM) and Thomas J. Watson (Research Center, Yorktown Heights, NY). Symbolic execution and program testing. *ACM Press New York, USA*, 19(7) :385 – 394, July 1976. 2.3

[ISO87] ISO. Lotos — A Formal Description Technique Based on the Temporal Ordering of Observational Behaviour. Draft International Standard 8807, International Organization for Standardization — Information Processing Systems — Open Systems Interconnection, Genève, July 1987. I.1

[ISO89] ISO/IEC. Estelle — A Formal Description Technique based on an Extended State Transition Model. Technical Report 9074, International Organization for Standardization — Information Processing Systems — Open Systems Interconnection, Genève, 1989. I.1

[ISO92] ISO/IEC. - Open Systems Interconnection, Information Technology - Open Systems Interconnection Conformance Testing Methodology and Framework - Part 1 : General Concept - part 2 : Abstract Test Suite Specification - part 3 : The Tree and Tabular Combined Notation (TTCN). *International Standard ISO/IEC 9646-1/2/3*, 1992. 2.5

[ISO94] ISO/IEC. - Information Technology - Open Systems Interconnection - Conformance Testing Methodology and Framework. *International Standard ISO/IEC 9646*, 1994. 2.4

[ISO97] ISO/IEC. - Open Systems Interconnection, Information Technology - Open Systems Interconnection Conformance Testing Methodology and Framework - part 3 : The Tree and Tabular Combined Notation (TTCN). *International Standard ISO/IEC 9646-3*, December 1997. 2.4

[IT92] ITU-T. ITU-T : Message Sequence Chart (MSC), ITU-T Recommendation Z.120, International Telecommunication Standards Sector SG 10, Geneva, 1992. 2.4

[IT95] ITU-T.Rec. X. 901/iso/iec 10746, Open Distributed Processing - Reference Model Genova, Swiss, 1995. 2.4

[IT99a] ITU-T. Recommendation Z.100. Specification and Description Language (SDL). Technical Report Z-100, International Telecommunication Union – Standardization Sector, Genève, November 1999. I.1, 2.4

[IT99b] ITU-TZ.130. Object Definition Language ITU-ODL, March 1999. 2.4

[Jah04] E. Jahier. The lurette v2 user guide. Technical Report TR-2004-5, Verimag, Centre Équation, 38610 Gières, Jun 2004. 2.3

[J.H99] A.Pataricza J.Hlavicka, E.Maehle. MAFALDA : Microkernel Assessment by Fault Injection and Design Aid. In Eds. Springer Lecture Notes in Computer Science 1667, editor, *3rd European Dependable Computing Conference (EDDC-3), Prague (République Tchèque) 15-17*, pages 143–160, Septembre 1999. 4

[JJ02] C. Jard and T. Jéron. TGV : theory, principles and algorithms. In *The Sixth World Conference on Integrated Design & Process Technology (IDPT'02)*, Pasadena, California, USA, June 2002. I.2, 2, 3, 3.1, 8.12

[JM99] T. Jeron and P. Morel. Test Generation Derived from Model-Checking. In *Computer Aided Verification'99, Trento, Italy, N. Halbwachs, D. Peled (eds.)*, volume 1633 of *Springer-Verlag, LNCS*, pages 108–122, Juillet 1999. 2, 2.5, 3, 3.4

[JSD+97] P. J. Koopman Jr., J. Sung, C. P. Dingman, D. P. Siewiorek, and T. Marz. Comparing operating systems using robustness benchmarks. In *Symposium on Reliable Distributed Systems*, pages 72–79, 1997. 4

[Jér91] T. Jéron. *Contribution à la validation des protocoles : test d'infinitude et vérification à la volée*. PhD thesis, Université de Rennes I, May 1991. 7.2.3, 7.2.3

[Jér02] T. Jéron. TGV : théorie, principes et algorithmes. *Techniques et Sciences Informatiques, numéro spécial Test de Logiciels*, 21, 2002. 3, 3.7

[Jér04] T. Jéron. *Contribution à la génération automatique de tests pour les systèmes réactifs*. Habilitation à diriger les recherches, Université de Rennes 1, March 2004. 3.2, 3.2, 3.4.5, 3.5

[KD99] P. Koopman and J. DeVale. Comparing the robustness of posix operating systems. In *FTCS*, pages 30–37, 1999. 4

[KJS98] N. P. Kropp, P. J. Koopman Jr., and D. P. Siewiorek. Automated Robustness Testing of Off-the-Shelf Software Components. In *Symposium on Fault-Tolerant Computing*, pages 230–239, 1998. 4

[Koh79] Z. Kohavi. *Switching and Finite Automata Theory*. McGraw-Hill, New York, 1979. ISBN 0-07-035310-7. 1

[Kor90] B. Korel. Automated Software Test Data Generation. *IEEE Transactions on Software Engineering ISSN 0098-5589*, 16(8) :870–879, August 1990. 2.3

[KPY99] I. Koufareva, A. Petrenko, and N. Yevtushenko. Test generation driven by user-defined fault models, 1999. 6.5.1

[LAC+96] J-C. Laprie, J. Arlat, A. Costes, A. Costes, Y Crouzet, Y. Deswarte, J-C. Fabre, H. Guillermain, M Kaäniche, K. Kanoun, C. Mazet, D. Powella, C. Rabéjac, and P. Thévenod. *Guide de la sûreté de fonctionnement*. Sous la direction de J-C. Laprie, Cepadues editions, ISBN 2.85428.382.1, 1996. 4, 2

[liH] IBM labs in HAIFA. *Test Execution Engine*. http ://www.haifa.il.ibm.com/projects/verification/mdt/papers/SPIDERPresentation.pdf. 2.4

[LL95] R. Lai and W. Leung. Industrial and Academic Protocol Testing : the Gap and the Means of Convergence. *Computer Networks and ISDN Systems*, 27 :537–547, 1995. 3.4

[LY94] D. Lee and M. Yannakakis. Testing Finite State Machines : State Identification and Verification. *IEEE Trans Computer*, 43(3) :306–320, 1994. 2.2, 1

[Lyn88] N. A. Lynch. I/O automata : A Model for Discrete Event Systems. In *Proc. of 22nd Conf. on Information Sciences and Systems*, pages 29–38, Princeton, NJ, USA, March 1988. 1.1.2, 2.2

[MA00] B. Marre and A. Arnould. Test sequences generation from lustre descriptions : Gatel. In *Fifteenth IEEE Int. Conf. on Automated Software Engineering (ASE 2000), Grenoble*, pages 229–237. IEEE Computer Society Press, septembre 2000. Démonstration de l'outil GATeL. 2.3, 5.1

[MB00] L. Ghirvu M. Bozga, J-C. Fernandez. C. Jard, T. Jéron, A. Kerbrat, P. Morel and L. Mounier. Verification and test generation for the SSCOP protocol. *Science of Computer Programming special issue on Formal Methods in Industry*, 36(1) :27–52, Janvier 2000. 2.5

[Meu98] C. Meudec. *Automatic Generation of Software Test Cases From Formal Specifications*. PhD thesis, Queen's University of Belfast, 1998. 4

[MFS90] B. P. Miller, L. Fredriksen, and B. So. An empirical study of the reliability of UNIX utilities. *Communications of the Association for Computing Machinery*, 33(12) :pages 32–44, 1990. 4

[MKL+95] B. Miller, D. Koski, C. Pheow Lee, V. Maganty, R. Murthy, A.r Natarajan, and Jeff Steidl. Fuzz Revisited : A re-examination of the Reliability of UNIX Utilities and Services. Technical report, citeseer.ist.psu.edu/miller95fuzz.html, 1995. 4

[Moo56] E. F. Moore. Gedanken-experiments on sequential machines. *Automata Studies*, pages 129–153, 1956. 1

[Mor00] P. Morel. *Une algorithmique efficace pour la génération automatique de tests de conformité*. PhD thesis, UFR IFSIC/ laboratoire IRISA, février 2000. 2.3, 3, 2b, 7.2.1

[MR01] T. Murnane and K. Reed. On the effectiveness of mutation analysis as a black box testing technique. In *13th Australian Software Engineering Conference (ASWEC 2001), Canberra, Australia*, pages 12–20. IEEE Computer Society, August 2001. 6.5.1

[MSG99] A. P. Mathur, M. L. Soffa, and N. Gupta. UNA Based Iterative Test Data Generation and its Evaluation. In *Proc of 14th IEEE International Conferanse on Automated Software Engineering (ASE'99), Florida, USA*, October 1999. 2.3

[MSG00] A. P. Mathur, M. L. Soffa, and N. Gupta. Generating Test Data For Branch Coverage. In *Proc of 15th IEEE International Conferanse on Automated Software Engineering (ASE'00), Grenoble, France*, September 2000. 2.3

[MV94] J. McManis and P. Varaiya. Suspension automata : A decidable class of hybrid automata. In *Proc. 6th International Computer Aided Verification Conference*, pages 105–117, 1994. 3.4.2

[MV99] G. McGraw and J. Viega. Why COTS software increases security risks. In *In Proceedings of the First International Workshop on Testing Distribu ted ComponentBased Systems*, May 1999. 4

[Nah94] R. Nahm. *Confornace Testing Based on Formal Description Techniques and Message Sequence Charts*. PhD thesis, Université of Berne, Institue for Informatics and Applies Mathematics, 1994. 2.4

[NH84] R. De Nicola and M. Henessy. Testing Equivalences for Processes. *Theoretical Computer Science*, 34 :83–133, 1984. 2.2

[NT81] S. Naito and M. Tsunoyama. Fault Detection for Sequential Machines by Transition Tours. In *Proceedings of the 11th IEEE Fault Tolerant Computing symposium*, pages 238–243, 1981. 2.2, 1

[Nta98] S. Ntafos. On random and partition testing. In *International Symposium on Software Testing and Analysis archive Proceedings of the 1998 ACM SIGSOFT international symposium on Software testing and analysis table of contents Clearwater Beach, Florida, United States*, pages 42 – 48. ACM Press New York, NY, USA, 1998. University of Texas at Dallas, Computer Science Program, Richardson, ACM Special Interest Group on Software Engineering ISBN : 0-89791-971-8. 2.3

[Org93] I. Organization. Open Systems Interconnection - OSI Conformance Testing Methodology and Framework, International Standards Organization. Open Systems Interconnection - OSI Conformance Testing Methodology and Framework, part 3 : TTCN Extensions,ISO/IEC JTC 1 DAM-1, 1993. 2.2, 2.5

[PC93] R. A. DeMillo (Purdue Univ. and Lafayette, IN) and A. J. Offutt (Clemson Univ., Clemson, SC). Experimental results from an automatic test case generator. *ACM Press New York, USA*, 2(2) :109 – 127, April 1993. ACM Transactions on Software Engineering and Methodology (TOSEM) archive. 2.3

[Pet01] A. Petrenko. Fault Model-Driven Test Derivation from Finite State Models :Annotated Bibliography. In *F Cassez, C Jard, B Rozoy, M D Ryan (Eds) Modeling and Verification of Parallel Processes*, volume 2067 of *LNCS*, pages 196–205, 2001. 2.2, 1, 6.5.1

[Pha94] M. Phalippou. *Relations d'Implantation et Hypothèses de Test sur les Automates à Entrées et Sorties*. PhD thesis, Université de Bordeaux, 1994. 2.2, 3.2

[Phi87] I.C.C. Phillips. Refusal Testing. *Theoretical Computer Science*, 50(3) :241–284, 1987. 2.2

[PL00a] A. Pretschner and H. Lötzbeyer. Autofocus on constraint logic programming. In *Proc of (Constraint) Logic Programming and software Engineering (LPSE'2000), London*, August 07 2000. 2.3

[PL00b] A. Pretschner and H. Lötzbeyer. Testing concurrent reactive systems with constraint logic programming. In *Proc of 2nd workshop on Rule-Base Constraint Reasoning and Programming, Singapore*, 2000. 2.3

[PL01] A. Pretschner and H. Lötzbeyer. Model based testing with constraint logic programming : First results and challenges. In *Proc. 2nd ICSE Intl. Workshop on Automated Program Analysis, Testing and Verification (WAPATV'01), Toronto*, May 2001. 2.3

[PRO01] PROTOS. *The Protos Project 1999 - 2001 - PROTOS - Security Testing of Protocol Implementations*. http ://www.ee.oulu.fi/research/ouspg/protos/, 2001. 4

[PV01] I. Parissis and J. Vassy. Test des proprietes de sûreté. In *In Actes du colloque Modelisation de Systemes Reactifs (MSR'01)*, pages 563–578, Hermes, 2001. 5.1

[PY92] A. Petrenko and N. Yevtushenko. Test suite generation for a given type of implementation errors. In *Proceedings of IFIP XII*, pages 229–243. Int'l Conf. Protocol Specification, Testing, and Verification, 1992. 6.5.1

[PYH03] A. Petrenko, N. Yevtushenko, and J. L. Huo. Testing Transition Systems with Input and Output Testers. In *Proceedings of the IFIP TC6/WG6.1 XV International Conference on Testing of Communicating Systems (TestCom 2003) Sophia Antipolis, France*, May 26-29 2003. 2

[Rab69] M.O. Rabin. Decidability of second order theories and automata on infinite trees. *Trans. Amer. Math. Soc.*, 141, 1969. 5.3

[Rab72] M.O. Rabin. Automata on infinite objects and church's problem. In *Proc. Regional AMS Conf. Series in Mathemamtics*, 1972. 5.3

[Rei99] Stuart Reid. Software fault injection : Inoculating programs against errors, by jeffrey voas and gary mcgraw, wiley, 1998 (book review). *Software Testing, Verification & Reliability (STVR)*, 1999. 6.1

[RJA99] R.Groz, T. Jéron, and A.Kerbrat. Automated Test Generation from SDL specifications. In R. Dssouli, G. von Bochmann, and Y. Lahav, editors, *SDL'99 The Next Millenium, 9th SDL Forum, Montréal, Québec*, pages 135–152, Elsevier, June 1999. 2.4, 3, 3.4

[RMJ05] Vlad Rusu, Hervé Marchand, and Thierry Jéron. Automatic verification and conformance testing for validating safety properties of reactive systems. In John Fitzgerald, Andrzej Tarlecki, and Ian Hayes, editors, *Formal Methods 2005 (FM05)*, LNCS. Springer, July 2005. 5.10

[RMT+04] V. Rusu, H. Marchand, V. Tschaen, T. Jéron, and B. Jeannet. From safety verification to safety testing. In *The 16th IFIP International Conference on Testing of Communicating Systems (TestCom04). Volume 2978 of LNCS*, Oxford, UK, March 2004. Springer-Verlag. 5.10

[RWNH98] P. Raymond, D. Weber, X. Nicollin, and N. Halbwachs. Automatic testing of reactive systems. In *19th IEEE Real-Time Systems Symposium*, Madrid, Spain, dec 1998. 2.3, 5.1

[SH01] S. Salva and F. Hacène. La qualité du test de conformité des systèmes temps-réel. In *3ième Conférence Francophone de MOdélisation et SImulation - Conception, Analyse et Gestion des Systèmes industriels MOSIM'01, Troyes, France*, Avril 2001. 2.2

[SL] L. Schieferdecker and M. Li. Conformance testing of TINA — in response to TINA - CAT RFP on TINA conformance testing framework. 2.4

[SLH98] L. Schieferdecker, M. Li, and A. Hoffmann. Conformance testing of TINA service components — the TTCN/CORBA gateway. *Lecture Notes in Computer Science, In Proccedings 5th International Conference Int. Service in Netwok*, 1430 :393–404, 1998. 2.4

[Spi88] J. M. Spivey. The fuzz Manual. Technical report, Oxford, 1988. 4

[ST87] R. Saracco and P.A.J. Tilanus. CCITT SDL : Overview of the Language and its Applications. *Computer Networks and ISDN Systems*, 1987. I.1

[Tar72] R. Tarjan. Depth-first search and linear graph algorithms. *SIAM J. Computation*, 2(1), june 1972. 2b, 7.2.1

[TB99] J. Tretmans and A. Belinfante. Automatic testing with formal methods. In *In Proceedings of the 7th European International Conference on Software Testing Analysis and Review EuroSTAR'99 (Barcelona, Spain)*, November 1999. 2.4

[TB02] J. Tretmans and E. Brinksma. Côte de Resyste Automated Model Based, 2002. 2, 2.4, 3

[TF98] P. Thevenod-Fosse. Unit and integration testing of lustre programs : a case study from the nuclear industry. In *9th European Workshop on Dependable Computing (EWDC-9)*, pages 121–124, Gdansk, Pologne, 14-16 Mai 1998. 5.1

[THZ+99] Timothy K. Tsai, Mei-Chen Hsueh, Hong Zhao, Zbigniew Kalbarczyk, and Ravishankar K. Iyer. Stress-Based and Path-Based Fault Injection. *IEEE Trans. Computers*, 48(11) :1183–1201, 1999. 4

[Tip95] F. Tip. A survey of program slicing techniques. *Journal of programming languages*, 3, pages 121–189, 1995. 8.3

[Tre92] J. Tretmans. *A Formal Approach to Conformance Testing*. PhD thesis, University of Twente, Enschede, Th Netherlands, 1992. 2.2, 3.2

[Tre96] J. Tretmans. Test Generation with Inputs, Outputs, and Repetitive Quiescence. In T. Margaria and B. Steffen, editors, *Second Int. Workshop on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'96)*, volume 1055 of *Lecture Notes in Computer Science*, pages 127–146. Springer-Verlag, 1996. I.2, 1.1.2, 2.4, 3.2, 3.4.2, 3.4.2, 3.4.2, 3.7, 8.12

[VT00] R. De Vries and J. Tretmans. On-the-Fly Conformance Testing Using. *International Journal on Software Tools for Technology Transfer (STTT), Springer-Verlag Heidelberg*, Volume 2(4) :pages 382 – 393, March 2000. 2.4

[Wei79] M. Weiser. *Program slices : formal, psychological, and practical investigations of an automatic program abstraction method*. PhD thesis, University of Michigan, Ann Arbor, 1979. 2, 8.3

[Wei95] C. Weissman. Security Penetration Testing Guidelines - chapitre 10. In *Handbook for the Computer Security Certification of Trusted Systems*, January 1995. 4, 6.1

[Wey79] E. J. Weyuker. Translatability and decidability questions for restricted classes of program schemas. *SIAM Journal on Computing ISSN 0097-5397 (print), 1095-7111 (electronic)*, 8(4) :587–598, 1979. 2.3

[WG796] ITU-T SG 10/Q.8 ISO/IEC JTC1/SC21 WG7. Information Retrievial, Transfer and Management for OSI ; Framework : Formal Methods in Conformance Testing. Commitee Draft CD 13245-1, IUT-T proposed recommendation Z 500. ISO - IUT-T, 1996. 2.2, 2.5

[ZHM97] H. Zhu, P. A. V. Hall, and J. H. R. May. Software unit text coverage and adequacy. *ACM Computing Surveys*, 29(4) :366–427, December 1997. 2.3

Résumé : Les manières de créer et de développer des systèmes informatiques ne cessent d'évoluer. La complexité croissante des logiciels informatiques (répartition du code, utilisation de composants externes, limitation des ressources, etc.) nécessite des méthodes de conception et de validation rigoureuses. Dans ce contexte la phase de test s'avère particulièrement importante car elle contribue à garantir un bon fonctionnement de l'implantation du logiciel, dans son environnement réel d'exécution. Cette thèse définit une méthode automatique de génération de tests destinés à évaluer la robustesse d'une implantation, c'est-à-dire sa capacité à respecter certaines propriétés comportementales malgré un environnement d'exécution dégradé (susceptible de fournir des entrées incorrectes, ou d'inclure des composants externes incapables de rendre le service attendu). L'approche que nous proposons est inspirée des techniques de génération de test utilisées en test de conformité des protocoles de communications dans lesquelles les suites de test sont générées à partir d'un modèle comportementale d'une spécification du logiciel. L'originalité de ce travail consiste à étendre cette technique pour prendre en compte un modèle de fautes (exprimant le comportement dégradé de l'environnement sous forme de mutations syntaxiques de la spécification) et un observateur (exprimant l'ensemble des comportements incorrects du point de vue de la robustesse). Les séquences de test produites sont alors correctes dans le sens où elles ne rejettent que des implantations non robustes vis-à-vis de cet observateur. Un prototype a été réalisé dans le cadre de la boîte à outils IF et évaluée sur des exemples d'implantation Java.

Mots Clés : Mutation, Spécification mutée, Modèle de faute, Test de propriétés, Test de robustesse, Vérification et validation de modèles.

Abstract : The ways to create and develop computer systems don't stop evolving. The increasing complexity of computer software (distribution of the code, reuse of external components, limitation of the resources, etc.), requires conception and strict validations methods. In this context, the test phase is particularly important to guarantee a correct functioning of the software, in its real environment of the execution. This thesis defines an automatic method of test generations to evaluate the robustness of a program, which is its capacity to respect behavioral property despite an degraded execution environment (likely to furnish incorrect entries, or to include external components incapable to return the service awaited). The approach that we propose is inspired from the generation techniques used in conformance testing of communication's protocols in which the test sequences are generated from a behavioral model of a software's specification. The originality of this work consists in extending this technique to take into account a fault model (expressing the degraded behavior of the environment in the form the point of vue of a syntaxique mutation of the specification) and an observer (expressing the body of the incorrect behaviors from robustness). The test sequences are correct if they reject only non robust programs with respect to this observer. A prototype was realized in the IF tools and evaluated on examples (Java programs).

Keywords : Mutation, Mutated Specificaiton, Fault model, Property testing, Robustness testing, Verification and validation of models.