Le test de conformité: état de l'art

Thierry JÉRON

Table des matières

Table des figures

1 Le test de conformité

Le test de conformité est une technique de validation qui consiste à réaliser des expérimentations sur une implémentation d'un système réactif afin d'en déduire la correction vis à vis d'une spécification de référence. C'est un test de type fonctionnel car on ne s'intéresse qu'à la correction vis à vis des fonctions réalisées par la spécification. C'est un test dit boîte noire car le code de l'implémentation n'est pas connu. L'expérimentation consiste à faire interagir l'implémentation sous test (*IUT* pour *implementation under test*) avec son environnement. Cet environnement est simulé par un testeur qui exécute des tests élémentaires appelés *cas de test*. Les comportements de l'IUT sont observés à des interfaces appelées *PCO* pour *Points de Contrôle et d'Observation* (voir figure 1.

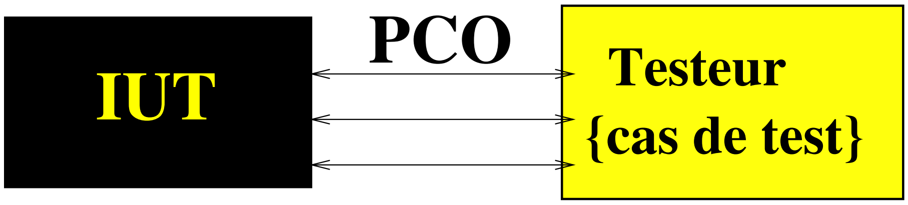

FIG. 1 – *Test de conformité*

Le test de conformité est utilisé dans divers domaines mais a été particulièrement étudié dans le domaine des protocoles de télécommunication. Dans ce cadre une norme a été décrite, ISO 9646 [**?**] qui décrits les ingrédients principaux du test de conformité. Nous décrivons ci-dessous quelques termes définis dans cette norme et utilisés dans la suite du document.

A faire:`Compléter`

Terminologie inspirée de ISO 9646 **Implantation sous test (IUT pour *Implementation Under Test*)**: L'implantation réelle du système à tester. C'est en général un code exécutable qui n'est pas connu du testeur. Cela peut être aussi du matériel.

Testeur: programme exécutant les tests sur l'IUT.

Architecture de test: description abstraite de la situation du testeur vis à vis de l'IUT, c'est à dire quelles sont les interfaces de l'IUT contrôlables ou observables (**points de contrôle et d'observation (PCO)**) par le testeur, comment s'effectue la communication entre testeur et IUT. L'architecture de test est choisie en fonction de l'architecture réelle de test. L'écriture des tests dépend donc étroitement de l'architecture choisie. Plusieurs architectures type sont normalisées: locale, remote, etc...

Suite de test: ensemble des tests élémentaires (**cas de test**) d'une IUT. En TTCN (*Tree and Tabular Combined Notation*), langage de description de tests issue de ISO 9646), une suite de test contient plusieurs parties. Le sommaire contient la **structure** c'est à dire comment la suite est organisée en groupes (par exemple, tests de capacités du protocole et paramètres) et sous-groupes (un sous-groupe par état, divisé pour chaque état en tests sur PDUs valides, invalides et inoportuns) et l'**index** qui décrit chaque **objectif de test**. La partie **déclarations** définit les types de données, i.e. les PDUs et ASPs, les timers, les PCO, les variables de la suite de test, des opérations de base. La partie **contraintes** définit les valeurs des différents champs des PDUs et ASPs utilisées dans les cas de test. La partie **comportementale** décrit le comportement du testeur par des **cas de tests**.

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

Cas de test: décrit un programme constitué d'interactions entre le testeur et l'implantation sous test, des opérations sur des temporisateurs de test, des verdicts. Il est généralement constitué d'un préambule, d'un corps de test, d'une séquence d'identification et d'un postambule. Il fait appel à des procédures qui sont des portions de cas de test communes à plusieurs cas de test. En particulier, les parties préambule, séquence d'identification et postambule sont décrites par des procédures.

Préambule: le préambule consiste en un sous-programme effectuant des intéractions avec l'IUT de façon à mener l'IUT dans un état particulier permettant d'appliquer le corps du test.

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

Identification: souvent, une partie du cas de test sert à identifier l'état dans lequel se trouve l'IUT (ou plutôt se trouvait l'IUT avant la phase d'identification)

Postambule: d'après ISO 9646, le postambule sert à revenir dans un état initial afin d'enchainer un autre cas de test. En fait, il s'agit bien souvent plutôt de l'état de contrôle initial. On ne se préoccupe pas forcément de l'état des variables du protocole. Ce qui peut poser des problèmes lors de l'exécution du cas de test suivant quand les variables n'ont pas été proprement réinitialisées.

2 Etat de l'art sur les techniques de génération

Les méthodes de génération formelle de test se décomposent en plusieurs familles. On ne parlera pas ici de certaines méthodes *ad hoc* qui n'ont pas de bases formelles même si elles utilisent parfois des techniques de description formelle comme SDL. Les deux familles auxquelles on s'intéresse ont comme base commune qu'elles partent de la donnée (au moins implicitement) des comportements d'une spécification formelle du protocole (on parle souvent de graphe d'état). Elles diffèrent sur le modèle utilisé pour représenter ce graphe d'état et par conséquent sur les techniques mises en œuvre. La première famille, la plus ancienne, est celle des méthodes fondées sur les automates. Elle est issue des méthodes de test de circuit et des problèmes de reconnaissance de machines [**?**, 11]. La deuxième famille est fondée sur les systèmes de transition et est issue des algèbres de processus et des problèmes d'équivalence de test.

Notons aussi qu'un groupe de chercheurs de ces deux communautés se sont réunis afin d'essayer d'unifier ces techniques de synthèse. Il en est résulté le document *Formal Methods in Conformance Testing* [14] qui décrit à un niveau méta les ingrédients essentiels à la description d'une technique de synthèse de test. En particulier ces ingrédients sont constitués des modèles de spécification, d'implémentation, de tests, de la définition de la conformité (modèle de faute ou relation de conformité), de la formalisation de l'exécution d'un test sur une implémentation, des verdicts associés, d'un algorithme de génération et des propriétés attendues des tests générés.

Par la suite, nous allons examiner les deux familles de méthodes citées précédemment. Nous n'allons pas faire une synthèse complète qui nous mènerait beaucoup trop loin, mais plutôt essayer de donner les ingrédients de base de ces méthodes permettant de comprendre leur philosophie. Nous décrivons ensuite la technique de génération à la volée telle qu'elle existe dans l'outil TGV.

2.1 Méthodes fondées sur les automates

La première famille de méthodes utilise comme modèle les machines de Mealy. Le lecteur désireux d'en savoir plus lira avec intérêt la synthèse de Lee et Yanakakis [22].

Modèle et problème

Les machines de Mealy sont des automates déterministes dont chaque transition est étiquetée par une entrée et une sortie. Formellement:

Définition 1 *Une machine de Mealy M est un quintuplet (S, I, O, δ, ω) tel que :*

- *S est une ensemble fini d'états,*
- *I est l'alphabet fini d'entrée,*
- *O est l'alphabet fini de sortie,*
- *$\delta : S \times I \to S$ est la fonction de transfert. On l'étend sur les séquences d'entrée en $\delta^* : S \times I^* \to S^*$,*
- *$\omega : S \times I \to O$ est la fonction de sortie. On l'étend également en $\omega^* : S \times I^* \to O^*$.*

La figure 2 décrit une machine de Mealy.

On suppose en général les machines déterministes (δ et ω sont des fonctions) et complètes (δ et ω sont partout définies). Comme pour les automates d'états finis, il existe une notion d'équivalence entre états: deux états s_i et s_j sont *équivalents* si pour toute suite d'entrée, ils produisent les mêmes suites de sorties: $\forall \sigma \in I^*, \omega^*(s_i, \sigma) = \omega^*(s_j, \sigma)$. On dit qu'une machine est *minimale* si elle n'a pas de paire d'états équivalents. Deux machines M et M' de mêmes alphabets sont dites *équivalentes* si pour tout état de M il existe un état équivalent dans M' et inversement. Pour toute machine M, on peut construire une machine minimale équivalente en calculant les classes d'équivalence d'états par raffinement de partition, de façon similaire à la minimisation des automates.

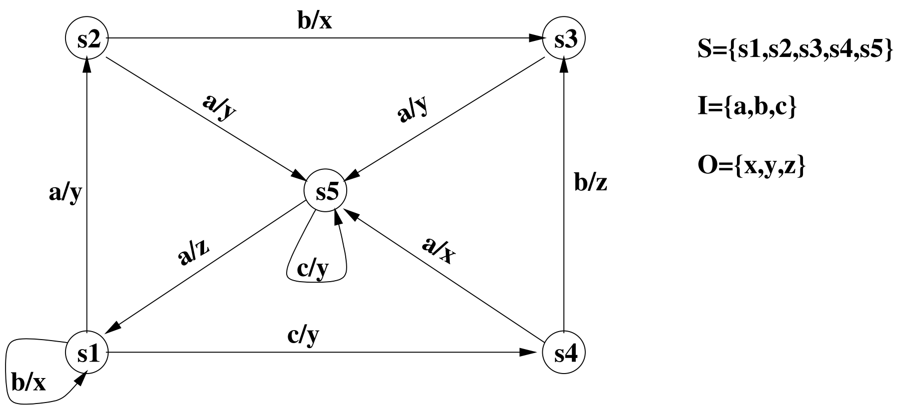

FIG. 2 – *Exemple de machine de Mealy*

Le problème du test de conformité basé sur ce modèle est donc le suivant: étant donnée une machine M connue (la spécification) et une machine I inconnue (l'implémentation) dont on peut seulement observer les entrées/sorties, déterminer par un test (une séquence d'entrées/sorties) si I et M sont équivalentes.

Pour que ce problème ait une solution, on suppose en général que la spécification est minimale, fortement connexe et que les deux machines ont le même nombre d'états (parfois que le nombre d'états de l'implémentation n'excède celui de la spécification que d'une constante).

Avec ces hypothèses, l'équivalence entre I et M est assurée si I n'a aucune faute des types suivants:

– les fautes de sortie: dans un état s, pour une entrée i, la sortie produite par I est o' au lieu de la sortie attendue o.
– les fautes de transfert: dans un état donné s, après la transition étiquetée i/o, l'état d'arrivée est s'' au lieu de s'.

Dans la littérature, on parle souvent de *modèle de fautes*.

La synthèse de test est fondée sur la recherche de la correction et de l'exhaustivité de la suite de test. C'est à dire que l'on cherche une suite de test qui ne rejette que des implémentations non conformes (correction) et qui puisse rejeter toute implémentation I non conforme (exhaustivité).

L'exhaustivité implique qu'il faut tester chacune des transitions de I en fonction de celles de M. Chaque test élémentaire correspondant à une transition $s \xrightarrow{i/o} s'$ de M a la forme suivante:

– aller en s
– appliquer i et vérifier la sortie o
– identifier l'état d'arrivée s' par une séquence.

En plus de la correction et de l'exhaustivité de la suite de test, on cherche à optimiser la taille de celle-ci en enchaînant les tests élémentaires en une séquence de taille minimale.

Séquences d'identification

Il existe plusieurs méthodes basées sur le même principe et qui diffèrent par la partie identification de l'état d'arrivée. Cette identification s'effectue par une séquence d'entrées/sorties qui doit permettre d'identifier certainement l'état d'arrivée.

La manière la plus facile d'identifier l'état d'arrivée est de supposer l'existence d'une entrée $status$ à laquelle le système répond par son état courant. C'est l'hypothèse faite pour la méthode dite du tour de transition (TT) [24].

La deuxième méthode appelée DS *Distinguishing Sequence* [**?**, **?**, **?**] suppose qu'il existe une séquence d'entrées DS capable de différencier toutes les paires d'états. Formellement, l'existence d'une séquence DS est caractérisée par la formule:

$$\exists DS \in I^*, \forall s_i \in S, \forall s_j \in S, \omega^*(s_i, DS) \neq \omega^*(s_j, DS) \tag{1}$$

La troisième méthode est appelée UIO (pour *Unique Input Output*) [21]. Elle suppose que pour chaque état s_i, il existe une séquence d'entrées UIO_i qui différencie s_i de tout autre état s_j. Formellement

$$\forall s_i \in S, \exists UIO_i \in I^*, \forall s_j \in S, \omega^*(s_i, UIO_i) \neq \omega^*(s_j, UIO_i) \tag{2}$$

Enfin la dernière méthode appelée W [6] suppose que toute paire d'états s_i, s_j peut être différenciée par une séquence d'entrée $c_{i,j}$.

Formellement

$$\forall s_i \in S, \forall s_j \in S, \exists c_{ij} \in I^*, \omega^*(s_i, c_{ij}) \neq \omega^*(s_j, c_{ij}) \tag{3}$$

Notons que les caractérisations des séquences d'identification DS, UIO_i et $c_{i,j}$ sont très proches. Elles diffèrent seulement par la place de la quantification existentielle sur la séquence d'identification. Evidemment ceci fait toute la différence.

En effet, il est clair que les séquences c_{ij} existent toujours aussitôt que la machine est minimale. En effet la formule 3 est précisément la définition de la minimalité pour les machines de Mealy.

En revanche l'existence des séquences UIO_i pour chaque état s_i n'est pas guarantie même pour des machines minimales. Cependant, il semble que les UIO existent assez souvent. De plus, bien que leur longueur soit au pire exponentielle dans la taille de la machine, elles sont souvent courtes en pratique.

Bien sûr, les séquences DS existent encore moins souvent que les UIO et en fait rarement. La technique de génération associée est donc en pratique inapplicable.

Exemple L'exemple de la machine de la figure 3, extrait de [22], permet de mieux se fixer les idées sur la différence entre les différentes séquences d'identification.

M est minimale comme le prouvent les séquences c_{ij} suivantes: $c_{12} = c_{13} = b$ qui distinguent s_1 de s_2 et s_3 et $c_{23} = a$ qui distingue s_2 de s_3.

On en déduit que s_1 possède un UIO, $UIO(s_1) = b$. On a $UIO(s_3) = a$ car $\omega(s_3, a) = 1$ et $\omega(s_1, a) = \omega(s_2, a) = 0$. Par contre, on peut montrer que s_2 n'a pas d'UIO. En effet:

– $UIO(s_2)$ ne peut débuter par a car $\omega(s_1, a) = \omega(s_2, a) = 0$ et $\delta(s_1, a) = \delta(s_2, a) = s_1$. Donc s_2 est indistinguable de s_1 après un a.

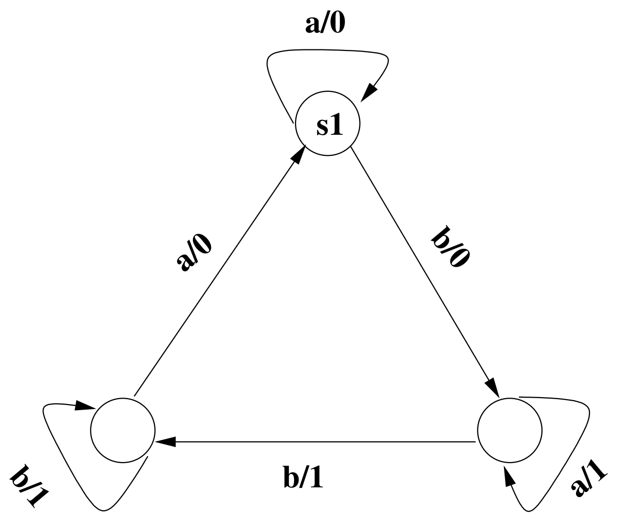

FIG. 3 – *Illustration des séquences d'identification*

– $UIO(s_2)$ ne peut débuter par b car $\omega(s_2, b) = \omega(s_3, b) = 1$ et $\delta(s_2, b) = \delta(s_3, b) = s_2$. Donc s_2 est indistinguable de s_3 après un b.

Il n'y a donc pas de solution.

Par conséquent cette machine n'a pas non plus de séquence distinguante (sinon celle-ci serait un UIO pour les 3 états).

Synthèse de test

Tour des transitions Expliquons d'abord la méthode la plus simple, celle du tour de transition. Puisque le problème de l'identification est réglé par l'entrée $status$, le test élémentaire d'une transition consiste en l'application de son entrée. Une suite de test correcte et exhaustive consiste donc à parcourir toutes les transitions. Optimiser la suite de tets consiste alors à trouver dans la machine une séquence de transitions de longueur minimale et passant au moins une fois par chaque transition. C'est un problème de type voyageur de commerce appelé Problème du Postier Chinois (voir [12] par exemple). Après transformation de la machine, ce problème se réduit à celui de la recherche d'un circuit Eulérien dans un graphe: un circuit est Eulérien s'il passe une et une seule fois par chaque transition. On utilise le théorème suivant:

Théorème 1 *Un graphe dirigé possède un circuit eulérien si et seulement si il est fortement connexe et symétrique.*

Un graphe est symétrique si chaque nœud a un degré entrant (nombre d'arcs entrants) égal à son degré sortant (nombre d'arcs sortants).

Sous les conditions de forte connexité et de symétrie, le calcul d'un circuit Eulérien se fait en temps linéaire par parcours et suppression des transitions empruntées jusqu'à épuisement des possibilités et recollement des circuits intermédiaires construits.

La machine de Mealy de départ est supposée fortement connexe mais pas nécéssairement symétrique. Elle n'a donc en général pas de circuit Eulérien. Par contre on peut la rendre symétrique: il suffit de dupliquer certaines transitions.

De plus on a le théorème:

Théorème 2 *Un circuit Eulérien d'une augmentation symétrique minimale (obtenue en dupliquant le moins possible de transitions) est un circuit de longueur minimale empruntant chaque transition au moins une fois.*

Le problème se ramène donc au calcul d'une augmentation symétrique minimale de la machine de départ qui peut se résoudre par programmation linéaire de la façon suivante. On associe une variable X_{ij} à chaque transition de s_i vers s_j représentant le nombre de duplications de cette transition dans l'augmentation symétrique. Pour chaque état s_i on pose alors l'équation $\Sigma_{s_j \to s_i}(X_{ji} + 1) = \Sigma_{s_i \to s_k}(X_{ik} + 1)$ qui caractérise la symétrie en l'état s_i. Il s'agit alors de résoudre ce système d'équation en optimisant $\Sigma_{i,j} X_{ij}$.

Méthode UIO La méthode UIO est un peu plus complexe et se réduit aussi au calcul de circuit Eulérien. Ici le test élémentaire de la transition $s_i \overset{i/o}{\to} s_j$ est composé de cette transition suivie de la séquence d'identification de l'état d'arrivée UIO_j. Le problème est de trouver la séquence de longueur minimale contenant au moins une fois chaque test élémentaire. Tout d'abord pour chaque test élémentaire on ajoute au graphe une transition de l'état s_i jusqu'à l'état s_k où s_k est l'état obtenu après application de UIO_j dans s_j. Il s'agit alors de trouver un circuit de taille minimale passant une fois par chacune de ces transitions supplémentaires et empruntant si nécéssaire les transitions originales. Ceci est à nouveau un problème connu de type voyageur de commerce appelé Problème du Postier Chinois Rural. On se ramène au problème de calcul de circuit Eulérien en construisant un graphe symétrique à partir des tests élémentaires et en ajoutant et dupliquant le minimum de transitions originales. Le problème d'augmentation "rurale" symétrique est plus complexe mais est à nouveau résolu par programmation linéaire ou calcul de flot. Cependant, pour que la méthode de test soit exhaustive (rejette bien toute implémentation non conforme) il est nécéssaire de vérifier que les UIO calculés sur la spécification sont aussi des UIO de l'implémentation.

Avantages et inconvénients

Le premier avantage de ce type de méthode est l'exhaustivité. Cependant les algorithmes sont relativement coûteux, ce qui cantonne leur applicabilité à des graphes d'états de taille très modeste (de l'ordre de la centaine d'états et de transitions). On voit parfois dans la littérature des applications de ces techniques sur de grosses spécification. En réalité elles ne s'applique qu'à un sous-graphe du graphe d'état obtenu en bornant la profondeur d'exploration lors de la construction de ce graphe. L'exhaustivité est donc souvent un leurre.

L'autre avantage est celui de l'identification des états d'arrivée. Cet aspect est intéressant car les suites de test manuelles possèdent souvent des séquences d'identification. Cependant, ces dernières ne cherchent pas à identifier un état global particulier mais seulement un état de contrôle particulier. Il y a donc un décalage entre ces méthodes et la pratique. Ce problème est cependant abordé dans les travaux de [25] où les auteurs cherchent à distinguer des ensembles d'états.

L'inconvénient le plus important est celui des hypothèses nécéssaires à l'application de ces méthodes portant d'une part sur la spécification mais surtout sur l'implémentation. On peut concevoir que les spécifications et implémentations soient déterministes quand elles se rapportent à une entité de protocole. C'est beaucoup moins réaliste si l'on considère un système composé dans le cas où la communication est asynchrone entre sous-systèmes. On est alors confronté à des situations où plusieurs sorties sont possibles après la même entrée (ω ne peut plus être une fonction). L'hypothèse sur l'égalité des nombres d'états est encore moins réaliste et extrèmement difficile à vérifier. Il existe cependant des travaux sur les machines non déterministes et des travaux où le nombre d'états de l'implémentation est borné par celui de la spécification plus une constante.

Un autre inconvénient est dû au modèle des machines de Mealy qui n'est pas relié aux langages comme SDL de façon évidente. En effet en SDL une transition est souvent constituée d'une entrée suivie de plusieurs sorties ou uniquement de sorties alors que pour les machines de Mealy, une transition comprend une entrée et une sortie.

Enfin les tests produits sont monolithiques (une seule longue séquence) alors qu'en pratique les tests manuels sont structurés en fonction d'objectifs précis.

Pour toutes ces raisons, nous avons choisi un autre type de technique et de théorie sous-jacente qui nous semble plus appropriée à la synthèse de test.

2.2 Méthodes fondées sur les systèmes de transition

Préliminaires

Dans ce paragraphe nous introduisons le modèle des systèmes de transitions étiquetés (LTS pour *Labelled Transition System*). et quelques notations standard.

Définition 2 *Un LTS est un quadruplet $M = (Q^M, A^M, \rightarrow_M, q_0^M)$ tel que:*
- *Q^M est un ensemble dénombrable non vide d'états*
- *$q_0^M \in Q^M$ est l'état initial*
- *A^M est l'alpabet des actions. A^M contient un sous-ensemble I^M d'actions internes*[1]
- *$\rightarrow_M \subseteq Q^M \times (A^M \cup \{\tau\}) \times Q^M$ est la relation de transition*

Nous utilisons $_M$ comme exposant (ou indice) pour identifier le LTS dont on parle car nous aurons souvent besoin de parler de plusieurs LTS à la fois. Ceci sera ommis quand il n'y a pas d'ambiguité.

On notera $q \xrightarrow{a}_M q'$ pour $(q, a, q') \in \rightarrow_M$. De plus $q \xrightarrow{a}_M$ sera parfois utilisé à la place de $\exists q' : q \xrightarrow{a}_M q'$. Nous ne distinguerons pas toujours entre un LTS et son état initial et nous noterons par exemple $M \rightarrow_M$ au lieu de $q_0^M \rightarrow_M$.

1. En général on restreint l'alphabet aux actions observables et on considère une action interne indifférenciée τ ne faisant pas partie de l'alphabet. On a modifié ceci ici car nous aurons besoin de raisonner sur les actions internes en les différenciant.

Introduisons quelques notations supplémentaires:

Définition 3 *Soit $M = (Q^M, A^M, \rightarrow_M, q_0^M)$ un LTS, $\mu_{(i)} \in A^M$ des actions, $a_{(i)} \in A^M \backslash I^M$ des actions observables , $\tau_{(i)} \in I^M$ des actions internes $\sigma \in (A^M \setminus I^M)^*$ une séquence d'actions observables, $q, q' \in Q^M$.*

- $q \stackrel{\mu_1 \cdots \mu_n}{\rightarrow}_M q' =_{def} \exists q_0, \ldots, q_n : q = q_0 \stackrel{\mu_1}{\rightarrow} q_1 \stackrel{\mu_2}{\rightarrow} \cdots \stackrel{\mu_n}{\rightarrow} q_n = q'$
- $q \stackrel{\varepsilon}{\Rightarrow} q' =_{def} q = q'$ *or* $q \stackrel{\tau_1 \cdot \tau_2 \cdots \tau_n{}^*}{\rightarrow} q'$
- $q \stackrel{a}{\Rightarrow} q' =_{def} \exists q_1, q_2 : q \stackrel{\varepsilon}{\Rightarrow} q_1 \stackrel{a}{\rightarrow} q_2 \stackrel{\varepsilon}{\Rightarrow} q'$
- $q \stackrel{a_1 \cdots a_n}{\Rightarrow} q' = def \exists q_0, \ldots, q_n : q = q_0 \stackrel{a_1}{\Rightarrow} q_1 \cdots \stackrel{a_n}{\Rightarrow} q_n = q'$
- $q \stackrel{\sigma}{\Rightarrow} =_{def} \exists\, q' : q \stackrel{\sigma}{\Rightarrow} q'$
- $\tau^*(q) =_{def} \{q' \in Q \mid q \stackrel{\varepsilon}{\Rightarrow} q'\}$
- q *after* $\sigma =_{def} \{q' \in Q \mid q \stackrel{\sigma}{\Rightarrow} q'\}$ *et* P *after* $\sigma =_{def} \bigcup_{q \in P} q$ *after* σ
- $Traces(q) =_{def} \{\sigma \in A^* \mid q \stackrel{\sigma}{\Rightarrow}\}$ *et* $Traces(M) =_{def} Traces(q_0^M)$.

Origines

Les méthodes de synthèse de test fondées sur les systèmes de transition (LTS) ont leur origine dans les travaux sur les équivalences comportementales et préordres, en particulier le préordre de test $\leq_{te}$ [7, 1]. Grossièrement, on a $I \leq_{te} S$ si pour tout autre LTS observateur T (le testeur), les observations (traces et blocages) que l'on peut faire par composition parallèle de T et I sont incluses dans les observations possibles de T sur S.

Brinksma [5] introduit ensuite la *relation de conformité* **conf**, qui limite l'observation aux traces de la spécification plutôt que de considérer toutes les observations possibles. Il en résulte la notion de *testeur canonique* pour **conf**. Un testeur canonique d'une spécification S est un LTS T qui permet de tester exhaustivement la relation **conf**: une implémentation passe le test si et seulement si elle est conforme à la spécification S.

Une relation plus forte que le préordre de test est le *préordre de refus* [**?**, **?**] qui permet au testeur, par l'introduction d'une nouvelle action Θ et l'adaptation de la composition parallèle d'observer l'absence d'action.

Tous ces travaux, bien que fondateurs, sont restés assez théoriques et n'ont jamais été appliqués sur un quelconque protocole réel. La raison principale est que le modèle de systèmes de transitions avec interaction symétrique n'est pas directement utilisable pour le test: il n'est fait aucune différence entre événements observés (sorties du système sous test) et événements contrôlés (entrées du système).. La différence est pourtant fondamentale dans la pratique du test où le testeur choisit une entrée et observe, de façon passive, les réaction de l'implémentation.

Systèmes de transition à entrée/sorties

Cette différence entre entrées et sorties nécessite un modèle plus riche. Plusieurs de ces modèles ont été utilisés pour le test dont les Input Output Automata (IOA) de Lynch [23] par Segala [**?**] les Input Output State Machines (IOSM) de Phalippou [26], et les Input Output Transition Systems (IOTS) de Tretmans [28]. Contrairement aux machines de Mealy dont les transitions portent à la fois une entrée et une sortie, les transitions de ces modèles sont soit des entrées, soit des sorties, soit des actions internes. Ces modèles sont tout à fait similaires ainsi que les travaux qui en ont découlé.

Dans la suite nous nous intéresserons principalement aux travaux de Tretmans sur les IOTS car nous avons pris un modèle similaire que nous avons appelé Input Output Labelled Transition System (IOLTS) et les travaux de l'Université de Twente nous ont beaucoup inspiré, au moins sur le plan théorique. Notons simplement que Tretmans considère que la spécification peut être décrite par un LTS alors que nous considérons la spécification aussi comme un IOLTS, mais ceci n'est pas fondamental.

Les systèmes de transitions à entrée/sortie (IOLTS) sont simplement des LTS où on distingue deux types d'actions observables: les entrées et les sorties.

Définition 4 *Un IOLTS est un LTS $M = (Q^M, A^M, \rightarrow_M, q_o^M)$ où l'alphabet d'actions A^M est partitionné en trois ensembles distincts $A^M = A_I^M \cup A_O^M \cup I^M$ avec A_I^M l'alphabet d'entrées, A_O^M l'alphabet de sorties, I^M l'alphabet des actions internes.*

En plus des notations standard sur les LTS, nous utiliserons les notations suivantes propres aux IOLTS:

- $Out_{\mathrm{M}}(q) =_{def} \{a \in A_{\mathrm{O}}^{\mathrm{M}} \mid q \overset{a}{\Rightarrow}_{\mathrm{M}}\}$ l'ensemble des sorties possibles en q.
- $Out_{\mathrm{M}}(P) \equiv \{Out_{\mathrm{M}}(q) \mid q \in P\}$ pour $P \subseteq Q^{\mathrm{M}}$ est l'ensemble des sorties possibles dans un ensemble d'états Q.

Les IOLTS et des variantes de ce modèle seront utilisés pour modéliser les objets intervenant dans la génération de test.

Spécification, implémentation

Le comportement de la *spécification* est modélisée par un IOLTS $S = (Q^{\mathrm{S}}, A^{\mathrm{S}}, \rightarrow_{\mathrm{S}}, q_0^{\mathrm{S}})$.

L'*implémentation* est une boîte noire. On suppose cependant (hypothèse de test habituelle) que son comportement est modélisable par un IOLTS $IUT = (Q^{\mathrm{IUT}}, A^{\mathrm{IUT}}, \rightarrow_{\mathrm{IUT}}, q_0^{\mathrm{IUT}})$ avec $A^{\mathrm{IUT}} = A_{\mathrm{I}}^{\mathrm{IUT}} \cup A_{\mathrm{O}}^{\mathrm{IUT}} \cup I^{\mathrm{IUT}}$ et $A_{\mathrm{I}}^{\mathrm{S}} \subseteq A_{\mathrm{I}}^{\mathrm{IUT}}$, $A_{\mathrm{O}}^{\mathrm{S}} \subseteq A_{\mathrm{O}}^{\mathrm{IUT}}$[2]. On suppose que l'implémentation est complète en entrée: dans chaque état elle ne peut refuser aucune entrée i.e. $\forall q \in Q^{\mathrm{IUT}}, \forall a \in A_{\mathrm{I}}^{\mathrm{IUT}}, q \overset{a}{\rightarrow}$. Pour ne pas allourdir les exemples, cette complétion sur les entrées sera supposée implicite.

Blocages

Le test permet d'observer le blocage d'un système par l'utilisation de temporisateurs.

Le blocage ou absence d'action *quiescence* peut être due à plusieurs causes: le blocage (ou *deadlock*: le système ne peut plus évoluer), le blocage de sortie (le système est bloqué en attente d'une entrée venant de l'environnement), ou le blocage vivant (ou *livelock*: le système diverge par une suite infinie d'action internes)[3]. Les différents

2. On suppose que l'alphabet d'entrée (resp. de sortie) de l'implémentation contient l'alphabet d'entrée (resp. de sortie) de la spécification.

3. Tretmans ne considère pas les livelocks car il se restreint aux processus non divergents. Cependant, il est intéressant de considérer les livelocks sur les systèmes finis qui correspondent simplement à des boucles d'actions internes

blocages sont illustrés dans la partie gauche de la figure 4.

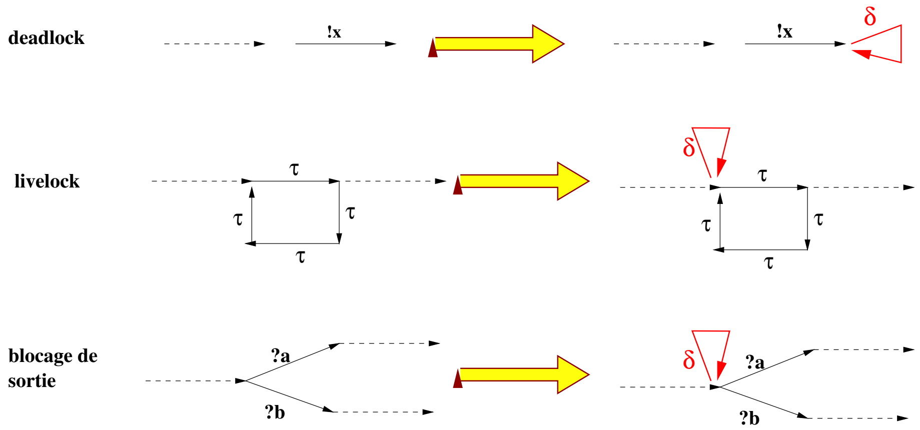

FIG. 4 – *Différents blocages*

Afin de considérer l'observation des blocages dans les tests, nous avons besoin de modéliser ceux-ci dans la spécification. En effet l'observation d'un blocage de l'IUT ne doit pas forcément produire un verdict *Fail* car il se peut que ce blocage existe dans la spécification.

A partir de l'IOLTS M de la spécification, nous aurons donc besoin de considérer un IOLTS M^δ (appelé *automate suspendu*, en anglais*suspension automata* par Tretmans) obtenu par ajout des informations de blocage. Formellement, M^δ est défini comme suit:

Définition 5 *L'*automate suspendu *de* $M = (Q^M, A^M, \rightarrow_M, q_0^M)$ *est l'IOLTS* $M^\delta = (Q^M, A^{M^\delta}, \rightarrow_{M^\delta}, q_0^M)$ *avec*

- $A^{M^\delta} = A^M \cup \{\delta\}$ *avec* $\delta \in A_O^{M^\delta}$ *(i.e.* δ *est considéré comme une sortie)*
- $\rightarrow_{M^\delta}$ *est obtenu à partir de* $\rightarrow_M$ *par ajout de boucles* $q \xrightarrow{\delta} q$ *pour tous les états de blocage (i.e. deadlock, livelock et blocage de sortie).*

Le calcul de M^δ est représenté par la transformation de la figure 4. A noter que dans le cas d'un livelock, il suffit d'ajouter une transition δ dans un des états d'une composante fortement connexe de τ car elle est devient alors accessible de tous les états de la composante.

Relations de conformité

De la même façon que pour les LTS, les IOLTS permettent de définir des préordres de test en considérant tous les observateurs possibles parmi les IOLTS. En restreignant les observations aux traces de la spécification, on obtient des relations de conformité.

On peut définir beaucoup de préordres de test et de relations de conformité suivant les observations permises par le test. Cependant pour être utilisables, il faut que ces relations de conformité tiennent compte de la réalité du test. Il est commun de penser que le test permet d'observer des traces mais pas les branchements, l'observations des branchements nécéssitant un retour en arrière. Nous avons signalé aussi précédemment que le test permettait d'observer les blocages.

Deux relations de conformité introduites par Tretmans nous ont intéressées plus particulièrement:

- **ioconf** qui dit qu'une implémentation I est conforme à sa spécification S si pour toute trace σ de S, les sorties (et blocages) de I après σ sont inclus dans les sorties (et blocages) de S. Formellement:

$$IUT \textbf{ ioconf } S \iff \forall \sigma \in Traces(S), Out(Iut^{\delta} \text{ after}_{\text{IUT}^{\delta}}\ \sigma) \subseteq Out(S^{\delta} \text{ after}_{\text{S}^{\delta}}\ \sigma)$$

- **ioco** [30] qui étend **ioconf** en considérant non seulement les traces de S mais aussi les traces avec blocages de S i.e. les traces de l'automate S^{δ} obtenu à partir de S en ajoutant des boucles étiquetées par δ dans chaque état de blocage. Formellement:

$$IUT \textbf{ ioco } S \iff \forall \sigma \in Trace(S^{\delta}), Out(Iut^{\delta} \text{ after}_{\text{IUT}^{\delta}}\ \sigma) \subseteq Out(S^{\delta} \text{ after}_{\text{S}^{\delta}}\ \sigma)$$

La relation **ioco** est expliquée sur l'exemple de la figure 5. Les deux premières implémentations $IUT1$ et $IUT2$ sont conformes à la spécification S alors que les deux suivantes ne le sont pas:

- IUT_1 **ioco** S: on peut vérifier qu'en tout état, les sorties de IUT_1 sont incluses dans celles de S. On voit qu'on peut donc restreindre les sorties.
- IUT_2 **ioco** S: ici l'état initial de IUT_2 permet une entrée supplémentaire par rapport à S mais ceci est autorisé: seules les sorties comptent. On permet donc de faire des spécifications partielles.
- $\neg(IUT_3$ **ioco** $S)$: la sortie $!c$ après l'entrée $?x$ n'est pas autorisée dans la spécification.
- $\neg(IUT_4$ **ioco** $S)$: le blocage de IUT_4 après la séquence $?x.!a$ n'est pas autorisée dans la spécification.

Cas de tests et suite de test

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

Définition 6 *Un cas de test est un IOLTS $TC = (Q^{TC}, A^{TC}, \rightarrow_{TC}, q_0^{TC})$ tel que*

- *$A^{TC} = A_I^{TC} \cup A_O^{TC}$ avec $A_O^{TC} \subseteq A_I^{S}$ (TC n'émet que des entrées de la spécification) et $A_I^{TC} \subseteq A_O^{IUT}$ (TC doit s'attendre à recevoir toutes les sorties possibles de l'implémentation),*
- *les verdicts sont caractérisés par des ensembles d'états puits,* **Pass** $\subseteq Q^{TC}$, **Fail** $\subseteq Q^{TC}$ *et* **Inconc** $\subseteq Q^{TC}$.
- *les états dans* **Fail** *et* **Inconc** *peuvent uniquement être atteints par des réceptions:* $\forall (q, a, q') \in \rightarrow_{TC},\ (q' \in \textbf{Inconc} \cup \textbf{Fail} \Rightarrow a \in A_I^{TC})$,

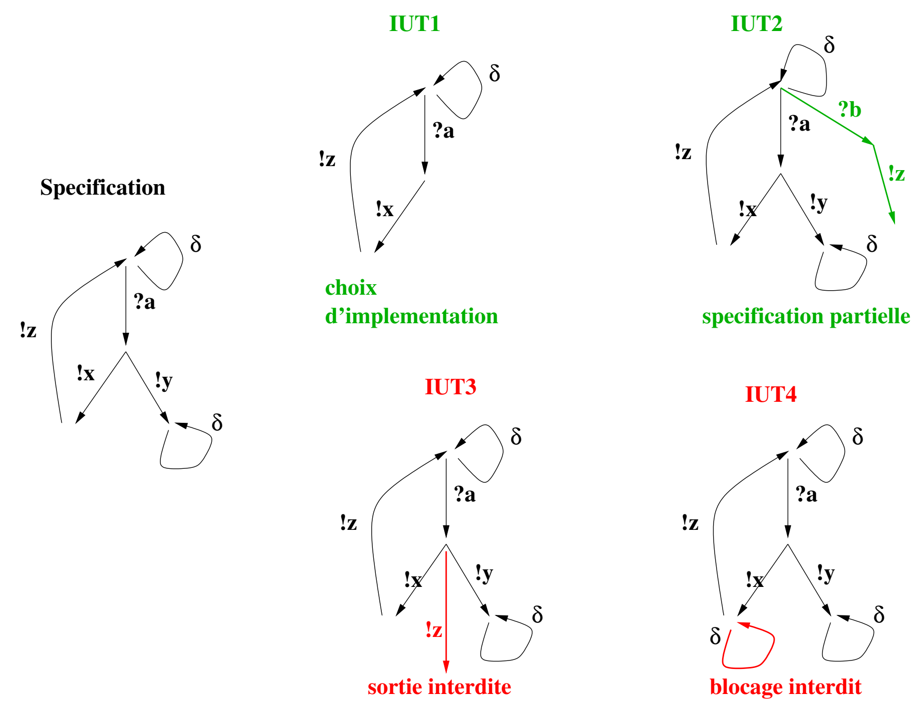

FIG. 5 – *Explication de* **ioco**

– *de tout état, un verdict est accessible:* $\forall q, \exists \sigma, q' \in \textbf{Pass} \cup \textbf{Inconc} \cup \textbf{Fail} q \xrightarrow{\sigma} q'$ [4]*,*

– *un test doit être contrôlable : il n'a jamais le choix entre deux sorties ou entre une entrée et une sortie, i.e.* $\forall q \in Q^{TC}, \forall a \in A_O^{TC}, q \xrightarrow{a}_{TC} \Rightarrow \forall b \neq a, q \not\xrightarrow{b}_{TC}$

– *il est complet sur les entrées dans tout état où une entrée est possible:* $\forall q \in Q^{TC},\ (\exists\ a \in A_I^{TC},\ q \xrightarrow{a}_{TC} \Rightarrow \forall\ b \in A_I^{TC},\ q \xrightarrow{a}_{TC})$.

Une suite de test *est un ensemble de cas de test.*

L'exécution d'un cas de test sur une implémentation est modélisée par la composition parallèle avec synchronisation sur les actions observables communes. De façon générale cette composition se définit de la façon suivante sur les LTS :

$$\frac{P \xrightarrow{a} P', Q \xrightarrow{a} Q'}{P||Q \xrightarrow{a} P'||Q'} \quad \frac{P \xrightarrow{\tau} P'}{P||Q \xrightarrow{\tau} P'||Q} \quad \frac{Q \xrightarrow{\tau} Q'}{P||Q \xrightarrow{\tau} P||Q'}$$

Le *verdict* associé à l'exécution d'un test sur une implémentation est déterminé par l'état final d'une exécution maximale.

*A faire:*définir proprement la notion de verdict et d'exécution maximale

Une suite de test et les éléments qui la compose, les cas de test, doivent permettre de tester la conformité d'une implémentation à une spécification. Une propriété minimale requise est la *correction* ou *non biais* des cas de test vis à vis de la spécification et de la relation de conformité. Elle dit en substance qu'un cas de test ne doit rejeter que des implémentations non conformes. Inversement on souhaiterait qu'une suite de test soit *exhaustive* i.e. qu'elle rejette toute implémentation non conforme. Formellement:

Définition 7 *Un cas de test TC est* non biaisé *pour S et* **ioco** *si et seulement si* $\forall IUT, IUT$ **ioco** $S \Rightarrow \neg(TC may reject IUT)$.

Une suite de test est exhaustive *pour S et* **ioco** *si et seulement si* $\forall IUT, not(IUT$ **ioco** $S) \Rightarrow TC may reject IUT$

Une suite de test est complète *si elle est non-biaisée (tous les cas de test sont non-baisés) et exhaustive.*

Il est possible d'obtenir la propriété de non-biais. Mais seule, elle ne suffit pas à avoir de "bons" tests. En effet des tests qui acceptent toute implémentation sont non biaisés. Cependant, en pratique il est impossible d'atteindre l'exhaustivité pour une suite de test finie aussitôt que la spécification contient des cycles. Il faudrait une infinité de cas de test et/ou des cas de tests ayant des comportement infinis. On cherchera donc à obtenir une propriété d'exhaustivité sur l'ensemble infini de tous les cas de tests potentiellement synthétisable. Cette propriété caractérise donc l'exhaustivité de la méthode de synthèse. Elle permettra de dire que si une implémentation est non conforme, alors il est toujours possible de produire un test qui peut rejeter l'implémentation. En effet, si on connait la cause de non-conformité, il est facile de construire un objectif qui produit un cas de test pouvant rejeter l'implémentation.

Génération de tests

Pour les relations de conformité **ioconf**et **ioco**, Tretmans [29, 30] a décrit des algorithmes de génération de test aléatoires. L'algorithme parcourt le système de transition

4. On n'exige pas l'inévitabilité d'un verdict car ceci empècherait d'avoir des tests possédant des boucles.

S (ou S^δ pour **ioco**) en construisant un test T. A chaque pas, soit on s'arrête et le verdit associé à l'état courant de T est *Pass*, soit on choisit une entrée i de S et on applique l'algorithme récusrsivement sur S after i, soit on considère toutes les sorties (et blocage) de l'alphabet. Dans ce cas, pour une sortie ou un blocage x de S, on applique récursivement sur S after o, pour une sortie ou un blocage non permis dans S on produit le verdict *Fail*. Notons que les deux relations de conformité et les algorithmes peuvent se généraliser en limitant l'exploration à un ensemble de traces $\mathcal{F}$ représentant un objectif de test. Notons également que les tests ne produisent que 2 types de verdicts: *Pass* et *Fail*. En effet le verdict *Inconc* est relatif aux objectifs de test or l'algorithme ci-dessus ne considère pas d'objectif de test.

L'algorithme peut produire un nombre *a priori* infini de tests. La suite de test (infinie) comprenant tous ces tests est prouvée correcte (ne rejette que des implémentations non conformes) et exhaustive (rejette toute implémentation non conforme)

Un algorithme découlant de celui de **ioco**est implémenté dans l'outil TorX [8]. Cet algorithme fonctionne pendant l'exécution du test au lieu de calculer les tests au préalable. A chaque étape, soit on s'arrête en produisant un verdict *Pass*, soit on choisit une entrée de la spécification qui est envoyée à l'implémentation, soit on recoit une sortie de l'implémentation ou on ne recoit rien (blocage) et on vérifie que la sortie reçue (ou le blocage) est autorisée dans la spécification.

Les travaux que nous avons réalisé sur la génération de test sont fondés sur un modèle similaire à celui de Tretmans et nous nous sommes beaucoup inspiré de sa théorie. Par contre les algorithmes de génération sont différents. Ceux-ci sont fondés sur la donnée d'un objectif de test représentant de façon abstraite un ensemble de comportements à tester. L'algorithme principal est très similaire à un algorithme de model-checking d'une propriété d'accessibilité. Enfin, les algorithmes fonctionnent à la volée. C'est à dire qu'ils construisent le graphe d'état de la spécification de façon paresseuse en contraignant l'exploration par l'objectif de test.

Avantages et inconvénients

Les modèles et techniques de générations fondées sur les systèmes de transition offrent une vision assez différente de celles fondées sur les machines de Mealy. Un premier avantage est que les modèles sont mieux adaptés car plus proches des modèles des langages de spécification. En particulier, ils permettent aisément de parler de non-déterminisme au sens des automates (des transitions de même étiquette résultant dans des états différents) et de non-déterminisme observable (plusieurs sorties possibles dans le même état). Les techniques de génération tiennent compte de ce non déterminisme au sens des automates par la déterminisation usuelle et tiennent compte du non déterminisme observable en produisant des tests ayanat des choix sur les sorties possibles de l'implémentation.

D'un point de vue algorithmique, ces techniques présentent une meilleure complexité (linéaire excepté la déterminisation qui est exponentielle) et des implémentations efficaces (notamment par construction des tests à la volée) qui permettent de traiter des spécifications de taille importante.

Bien sûr ceci a un prix. Un prix à payer est la non-exhaustivité. Même si les techniques sont exhaustives (la suite de test comprenant tous les tests potentiellement synthétisables est exhaustive), cette exhaustivité est impossible à atteindre en pratique car les suites de test doivent être finies. On perd aussi la notion d'identification d'état. En effet, l'identification d'état sert à prouver l'équivalence entre implémentation et spécification alors que les relations de conformité **ioconf**et **ioco**se limitent à rechercher une inclusion de sorties et blocages possibles.

3 Génération de tests à la volée

Nous présentons dans cette section les travaux que nous avons réalisé sur la génération de test. Une partie de ces travaux a été effectuée grâce à une collaboration étroite avec le laboratoire Verimag de Grenoble et a mené à un premier algorithme de génération [10] qui a été implémenté dans le prototype TGV (pour *Test Generation with Verification technology*) et expérimenté sur une étude de cas [**?**]. Ceci nous a permis de permis de démontrer la faisabilité de l'approche mais à cette époque le prototype TGV ne fonctionnait pas complètement à la volée. Durant la thèse de Pierre Morel que j'ai encadrée de 96 à début 2000, nous avons conçu et implémenté dans TGV les algorithmes permettant de calculer les tests à la volée. En particulier nous avons conçu un algorithme de réduction et déterminisation à la volée [17] et un nouvel algorithme principal fondé sur le model-checking [18]. TGV a été expérimenté sur différentes études de cas [4, 19, 20] et a fait l'objet d'un transfert industriel [13].

Dans la suite, nous allons présenter les principales idées de la génération à la volée en nous référant a nos travaux les plus récents. Nous présenterons également le prototype TGV et son expérimentation.

3.1 Motivation

L'idée maîtresse de notre technique est d'adapter des algorithmes de model-checking à la volée à la synthèse de test. Nous ne sommes pas les seuls à avoir pensé à utiliser le model-checking pour la synthèse de tests mais nous sommes certainement ceux qui avons poussé l'idée le plus loin.

L'idée d'utiliser le model-checking est simple: soit P une propriété décrivant un objectif de test, i.e. un ensemble de séquences de comportements d'une spécification S. Imaginons le cas simple où S n'a que des actions observables et est déterministe. Typiquement en CTL, P serait de la forme $P = \exists \diamond (a \wedge \exists \diamond b)$: on cherche une séquence qui contient un a puis un b. Effectuer le model-checking de $\neg P$ sur S (ici $\neg P = \forall \Box (\neg a \vee \forall \Box \neg b)$). Si P décrit effectivement des comportements de S, $\neg P$ n'est pas satisfaite par S et un model-checker produira en général une séquence contre-exemple de la forme $\sigma = v.a.w.b$ pour des séquences v et w. On utilise ensuite σ comme séquence principale du test, les choix sur les sorties de S dans les états traversés par σ menant à un verdict *Inconc*. Noter que plutôt que de considérer le model-checking de la négation de la propriété et un contre-exemple, on peut considérer de façon équivalente le model-checking de la propriété et une séquence temoin, mais les model-checkers produisent rarement des témoins. Cette idée est utilisée par exemple dans [**?**] en utilisant le model-checker SPIN et la logique LTL.

Malheureusement, on voit qu'avec cette idée, on ne produit que des séquences de test. Or les tests écrits à la main ont souvent la forme d'arbre ou même de graphes généraux. De plus, il faut prendre en compte la notion de comportement visible (les spécification ont en général des actions internes), la différence entre observation et contrôle, les problèmes de non-déterminisme, les blocages, les timers, les verdicts, etc. Il est donc illusoire de se contenter d'utiliser un model-checker standard qui ne considère pas toutes ces notions. Il faut plutôt spécialiser les algorithmes de model-checking au problème de la synthèse de test. Le problème d'explosion combinatoire existant en vérification existe aussi en synthèse de test. La solution que nous avons exploré en vérification est le model-checking à la volée. Nous avons donc spécialisé les algorithmes de model-checking à la volée à la synthèse de test.

3.2 Modèles du test de conformité

Comme nous l'avons suggéré dans la section **??**, nous utilisons comme modèle principal les systèmes de transitions à entrées/sorties (IOLTS) qui sont dérivés des systèmes de transition etiquetés (LTS pour *Labelled Transition System*).

Les IOLTS et des variantes de ce modèle seront utilisés pour modéliser les objets intervenant dans la génération de test.

Le comportement de la *spécification* est modélisée par un IOLTS $S = (Q^{\text{S}}, A^{\text{S}}, \rightarrow_{\text{S}}, q_0^{\text{S}})$.

L'*implémentation* est modélisable par un IOLTS $IUT = (Q^{\text{IUT}}, A^{\text{IUT}}, \rightarrow_{\text{IUT}}, q_0^{\text{IUT}})$ avec $A^{\text{IUT}} = A_{\text{I}}^{\text{IUT}} \cup A_{\text{O}}^{\text{IUT}} \cup I^{\text{IUT}}$ et $A_{\text{I}}^{\text{S}} \subseteq A_{\text{I}}^{\text{IUT}}$, $A_{\text{O}}^{\text{S}} \subseteq A_{\text{O}}^{\text{IUT}}$. On suppose que IUT est complète en entrée.

Un des ingrédients principaux de notre technique de génération de test est l'utilisation d'objectifs de test.

Un *objectif de test* est en pratique (dans une suite de test TTCN par exemple) une description abstraite de comportements à tester, en général une séquence d'actions. Pour la synthèse de test, nous l'utiliserons comme un critère de sélection de test. On aurait pu prendre une logique comme CTL ou LTL pour décrire les objectifs de test. Nous avons choisi des automates d'état finis pour plusieurs raisons:

- c'est un formalisme simple que tout ingénieur comprend de façon intuitive.
- un test doit avoir un comportement fini alors que CTL et LTL permettent de raisonner sur les comportements infinis.
- on peut aisément étendre les automates avec des variables comme dans le langage GOAL d'ObjectGéode pour permettre une sélection plus fine.
- les types d'objectifs que nous avons rencontré dans des suites de test réelles se codent bien par des automates et mèneraient à considérer des sous-classes de CTL ou LTL correspondant aux automates.

Pour rester dans le même formalisme, nous allons étendre les IOLTS afin de leur donner le rôle d'automate reconnaisseur. Pour permettre une sélection fine, on considèrera deux ensembles distincts d'états marqués servant soit à accepter soit rejeter des séquences d'actions de la spécification. Formellement:

Définition 8 *Un* objectif de test *est un IOLTS déterministe et complet*[5] $TP = (Q^{TP}, A^{TP}, \rightarrow_{TP}, q_0^{TP})$ *équipé de deux ensembles d'états puits* $Accept^{TP}$ *et* $Reject^{TP}$[6] *de même alphabet que la spécification i.e.* $A^{TP} = A^S$.

Remarque 1 L'objectif de test sert à sélectionner des séquences de S. Ce n'est pas un test aussi il n'y a pas de raison de limiter son alphabet à des actions observables. De plus, utiliser des actions internes dans les objectifs s'est avéré très utile dans le cas du test dans un contexte: l'objet à tester n'est accessible qu'à travers un contexte (voir figure 6). Dans ce cas la spécification doit décrire à la fois l'objet à tester et le contexte. Les actions observables sont celles à l'interface du contexte et de l'environnement. Or ce n'est pas le contexte qu'on veut tester (on suppose en général qu'il est testé par ailleurs). On sélectionnera donc les tests en décrivant des objectifs décrivant des comportements de l'objet à tester, pas du contexte. Bien sûr, le test résultant lui n'a que des actions observables.

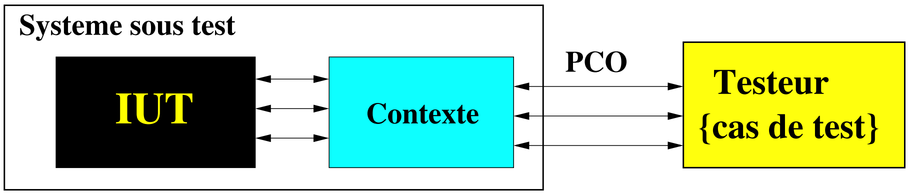

FIG. 6 – *Test dans un contexte*

Remarque 2: choisir des automates complets peut sembler laisser peu de place à l'abstraction dans la description des objectifs. En pratique dans TGV on utilise une étiquette spéciale notée "*" qui si elle étiquette une transition $q \xrightarrow{*} q'$ signifie que pour tout a n'étiquetant pas explicitement une transition d'origine q, on a $q \xrightarrow{a} q'$. Cette notation permet de mentionner le complémentaire d'un ensemble de transition sans l'énumérer. De plus on décrira des objectifs incomplets mais qui seront complétés implicitement par une boucle dans le même état étiquetée par "*". Enfin, nous permettons de décrire des ensembles d'étiquettes de transition par des expressions régulières.

Exemple: voir figure 7

3.3 Algorithmes de synthèse à la volée

Nous avons maintenant défini les entrées de la synthèse de test (la spécification et l'objectif de test), et la sortie (le cas de test) ainsi que les propriétés attendues de

5. Complet: dans tout état, toute action est tirable i.e. $\forall q \in Q^{\mathrm{TP}}, \forall \in A^{\mathrm{TP}}, q \xrightarrow{a}_{\mathrm{TP}}$.

6. Etat puits: q est un état puits si $\forall a \in A^{\mathrm{TP}}, q \xrightarrow{a}_{\mathrm{TP}} q$.

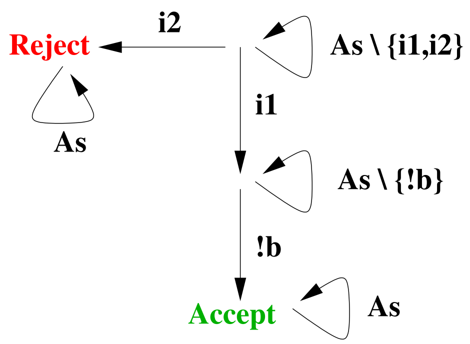

FIG. 7 – *Exemple d'objectif de test*

cette sortie: le non-biais et l'exhaustivité (de la méthode). Reste à décrire comment construire les cas de test. Cette construction se décrit fonctionellement en plusieurs étapes: un produit synchrone entre la spécification et l'objectif de test, la suppression des actions internes et la déterminisation, la synthèse d'un sous-graphe et la résolution des conflits de contrôlabilité. Néanmoins, ces algorithmes fonctionnent à la volée ce qui signifie qu'une exécution de l'algorithme principal va activer l'exécution de chacune des étapes intermédiaires. Le terme "à la volée"signifie également que pour produire le résultat, il ne sera pas nécéssaire de construire les résultats complets de chaque étape, ni même les entrées complètes, en particulier le graphe d'état (l'IOLTS) de la spécification.

Il serait très difficile d'expliquer (et de comprendre surtout) le comportement global de la composition des algorithmes sans cette découpe fonctionnelle. Nous allons donc décrire chaque étape et l'algorithme correspondant comme si la construction était complète puis nous reviendrons plus tard sur l'aspect à la volée. Nous ne décrivons pas ici tous les détails des algorithmes (on pourra en trouver une description complète dans [**?**]) mais nous essayons d'en donner les idées principales. Mais d'abord, nous présentons quelques notions préliminaires qui serviront ensuite dans plusieurs algorithmes.

Préliminaires

Les LTS (et les IOLTS) peuvent être vus comme des graphes dirigés. Nous aurons donc besoin d'utiliser des concepts et algorithmes de graphes. Un graphe G est défini par un ensemble de nœuds V (pour *vertices*) et un ensemble d'arcs E (pour *edges*). Une des notions principales que nous utilisons dans les algorithmes de génération de test est celle de composante fortement connexe d'un graphe dirigé.

Définition 9 *Un graphe dirigé est* fortement connexe *si pour tout nœud v et tout nœud w il existe un chemin de v à w.*

Soit maintenant la relation d'équivalence suivante définie sur un graphe $G = (V, E)$:

$$v\mathcal{R}w \iff v = w \text{ ou il existe un chemin de } v \text{ à } w \text{ et un chemin de } w \text{ à } v.$$

Les classes d'équivalence de cette relation forment une partition. Chaque classe d'équivalence V_i induit un sous-graphes G_i dont l'esemble de nœuds est V_i et l'ensemble des arcs est le sous-ensemble d'arcs de V reliant les nœuds de V_i entre eux. Un tel sous-graphe $G_i = (V_i, E_i)$ est appelé *composantes fortement connexes* ou CFC. Une composante *non triviale* est une composante qui n'est pas réduite à un seul nœud.

Le quotient du graphe par la relation d'équivalence $G/\mathcal{R}$ définit aussi un graphe appelé *graphe réduit* dont les nœuds sont les CFC et il existe un arc d'une CFC cfc_i vers une CFC cfc_j s'il existe des nœuds $v_i \in cfc_i$ et $v_j \in cfc_j$ et un arc de v_i vers v_j. Ce graphe est acyclique mais pas nécéssairement connexe. C'est donc une fôret d'arbres réduits de CFC.

FIG. 8 – *Composantes fortement connexes*

Les CFCs ont la propriété d'être stables pour les propriétés d'accessibilité i.e. deux nœuds u et v sont dans la même CFC si et seulement si pour tout autre nœud w, il existe un chemin de u à w si et seulement il existe un chemin de v à w:

$$u\mathcal{R}v \iff (\forall w, u \rightarrow^* w \iff v \rightarrow^* w$$

Nous utiliserons plusieurs fois cette propriété dans les algorithmes de synthèse de test pour la simple raison que plusieurs problèmes rencontrés s'exprimeront en terme d'accessibilité.

Algorithme de calcul de CFC L'algorithme le plus efficace pour calculer les CFCs est dû à Tarjan [27]. Nous n'allons pas ici expliquer en détail le fonctionnement de l'algorithme mais juste en donner une version itérative (la version originale de Tarjan est récursive) et expliquer quelques unes de ses caractéristiques. Dans [18], nous donnons une version itérative "à trous". Les "trous"de cet algorithme sont ensuite instanciés pour les algorithmes de chaque problème traité.

L'algorithme StrongConnect (voir figure 9) est un parcours en profondeur (DFS pour *Depth First Search*) itératif. Il utilise donc une pile Dfs_Stack. Une DFS numérote les nœuds dans l'ordre de parcours (attribut *number*) et partitionne les arcs en quatre ensembles qui dépendent de l'ordre de parcours:

- les *tree-arcs* qui vont d'un nœud vers un nœud non encore visité, ces arcs formant une forêt couvrante du graphe (ensemble d'arbres couvrant tous les nœuds du graphe,
- les *fronds* qui vont d'un nœud vers un ancêtre dans la forêt couvrante,
- les *cross-links* qui relient deux nœuds de deux sous arbres distincts,
- les *short-cuts* qui relient un nœud à un descendant dans la forêt.

Cette partition est illustrée par l'exemple de la figure 10

On appelle *racine* (*root*) d'une CFC le premier nœud de la CFC rencontré dans une DFS. Il a donc le plus petit numéro de DFS parmi les nœuds de la même composante. Le principe de l'algorithme StronConnect est de calculer ce nœud pour chaque CFC

FIG. 9 – *StrongConnect itératif*

$Adjacency\_Set(q)$ donne l'ensemble des transitions tirables en q,
$Adjacency\_Set(q) := \{(a, q') \mid q \xrightarrow{a} q'\}$
Procedure StrongConnect (q_{start}:state);
var
q_{source}, q_{target}, q_{pred}, q : state;
Adj_{source}, Adj_{target}, Adj_{pred}: adjacency_set;
BEGIN
Creation of Dfs_Stack; Creation of Scc_Stack;
$q_{\text{start}}.number := i$; $q_{\text{start}}.lowlink := i$; $i := i + 1$; $q_{\text{start}}.act := \epsilon$;
Push $(q_{\text{start}}, Adjacency\_Set(q_{\text{start}}))$ in Dfs_Stack; Push q_{start} in Scc_Stack;
while not empty Dfs_Stack **do begin**
$(q_{\text{source}}, Adj_{\text{source}}) := Top(Dfs\_Stack)$;
if $Adj_{\text{source}} \neq \emptyset$ **then begin**
ChooseAndRemove (m, q_{target}) from Adj_{source};
if q_{target} is not yet numbered **then begin** (*$(q_{\text{source}}, m, q_{\text{target}})$ is a tree-arc*)
$q_{\text{target}}.number := i$; $q_{\text{target}}.lowlink := i$; $i := i + 1$; $q_{\text{target}}.act := m$;
Push $(q_{\text{target}}, Adjacency\_Set(q_{\text{target}}))$ in Dfs_Stack; Push q_{target} in Scc_Stack;
else (* $(q_{\text{source}}, m, q_{\text{target}})$ is not a tree-arc *)
if $q_{\text{target}}.number < q_{\text{source}}.number$ **and** q_{target} in Scc_Stack **then**
(* $(q_{\text{source}}, m, q_{\text{target}})$ is a frond or a cross-link in the same SCC) *)
$q_{\text{source}}.lowlink := min(q_{\text{source}}.lowlink, q_{\text{target}}.number)$;
else begin (* $Adj_{\text{source}} = \emptyset$ *)
Pop (q_{source}) from Dfs_Stack;
if $q_{\text{source}}.lowlink = q_{\text{source}}.number$ **then** (*q_{source} is root of SCC*)
while $q := top(Scc\_Stack)$ satisfies $q.number \geq q_{\text{source}}.number$ **do**
(* creation of a new component *)
Pop (q) from Scc_Stack and put q in current component;
if not empty Dfs_Stack **then begin** (* backtracking *)
$(q_{\text{pred}}, Adj_{\text{pred}}) := top(Dfs\_Stack)$;
$q_{\text{pred}}.lowlink := min(q_{\text{pred}}.lowlink, q_{\text{source}}.lowlink)$;
end
end (* end of else Adj_{source} is empty *)
end (* end of while not empty Dfs_Stack*)
END;

Procedure MAIN (**Graph:** G);
BEGIN
var i : integer **init** 0;
for all vertex q_{start} in Q^{G} **if** q_{start} not yet numbered **then** StrongConnect (q_{start});
END;

FIG. 10 – *Partition de l'ensemble des arcs par une DFS.*

en utilisant un attribut entier *lowlink*. Pour un nœud v, $v.lowlink$ est le numéro le plus petit d'un nœud u accessible par un nombre quelconque de *tree-arcs* suivi d'au plus un *frond* ou un *cross-link*. Cette synthèse se fait par synthèse lors de la DFS le long des *tree-arcs*, *fronds* et *cross-links*. On utilise alors la propriété que r est racine de CFC $\iff r.number = r.lowlink$. D'autre part l'algorithme utilise une pile Scc_Stack contenant les nœuds dont on n'a pas encore fini le calcul de CFC. Les nœuds sont empilés en même temps dans Dfs_Stack et Scc_Stack. Quand on dépile une racine r de CFC de Dfs_Stack, tous les nœuds de Scc_Stack au dessus de r sont dans la même CFC et toute la CFC est dépilée.

Produit synchrone

Nous avons en entrée de la génération l'IOLTS de la spécification S et celui de l'objectif de test TP. Le produit synchrone consiste à définir les comportements de la spécification qui nous intéresse pour l'objectif donné. Il ne va pas pour l'instant s'agir d'extraire ces comportements car nous aurons besoin aux étapes suivantes de comportements de S aux frontières de ceux acceptés par TP. De plus nous verrons que le calcul à la volée fera en sorte de ne construire effectivement qu'une partie des comportements du produit, au mieux, juste ce qu'il faut. Le produit va donc simplement étiqueter les comportements de S par les états $Accept$ et $Reject$ de l'objectif. Ceci aura éventuellement pour effet de déplier les comportements de S car un même état de S peut être accessible par plusieurs chemins préfixes ou non d'états marqués. Notons aussi que TP est complet. Ceci implique que tous les comportements de S seront présents dans le produit.

Définition 10 *Soient S un IOLTS et TP un IOLTS complet et déterministe. Le produit synchrone $PS = S \times TP$ est un IOLTS $PS = (Q^{PS}, A^{PS}, \rightarrow_{PS}, q_0^{PS})$ muni de deux ensembles disjoints d'états $Accept^{PS}$ et $Reject^{PS}$ tels que :*

- *$q_0^{PS} = (q_0^S, q_0^{TP})$, l'état initial est formé des états initiaux de chaque opérande,*
- *$Q^{PS} = Q^S \times Q^{TP}$: les états du produit sont formés par les couples d'états de chaque opérande. On restreindra Q^{PS} aux états accessibles depuis l'état initial par la relation de transition $\rightarrow_{PS}$*
- *$\rightarrow_{PS}$ est définie comme suit: $(q^S, q^{TP}) \xrightarrow{a}_{PS} (q'^S, q'^{TP})$ ssi $q^S \xrightarrow{a}_S q'^S$ et $q^{TP} \xrightarrow{a}_{TP} q'^{TP}$,*
- *$A^{PS} = A^S$, l'alphabet d'actions est le même que celui de S (et de TP)*
- *$Accept^{PS} = \{(q^S, q^{TP}) \in Q^{PS} | q^{TP} \in Accept^{TP}\}$,*
- *$Reject^{PS} = \{(q^S, q^{TP}) \in Q^{PS} | q^{TP} \in Reject^{TP}\}$*

L'algorithme de calcul de la partie accessible de $PS = S \times TP$ est donc trivial. Il suffit de construire (en largeur ou en profondeur) le produit à partir de l'état initial. C'est en fait l'étape suivante qui effectue ce parcours en profondeur.

τ-fermeture et déterminisation

L'étape suivante consiste à extraire le comportement observable de la spécification S (qui est aussi celui de PS) . Par comportement observable, on entend les actions observables mais aussi les blocages. Nous avons déjà défini le *suspension automata* qui ajoute des boucles étiquetées δ à chaque état de blocage. Nous verrons ensuite comment on le calcule pendant la τ-fermeture. Nous allons donc définir l'IOLTS PS_{VIS} résultat de la τ-fermeture et de la déterminisation à partir de l'IOLTS PS^{δ}.

Définition 11 *Le produit synchrone τ-réduit et déterminisé est un* IOLTS $PS_{\textit{VIS}} = (Q^{\textit{VIS}}, A^{\textit{VIS}}, \rightarrow_{\textit{VIS}}, q_0^{\textit{VIS}})$ *muni de deux ensembles d'états $Accept^{\textit{VIS}}$ et $Reject^{\textit{VIS}}$ tels que :*

- $Q^{\textit{VIS}} \subseteq 2^{Q^{PS^{\delta}}}$*, l'ensemble des parties de $Q^{PS^{\delta}}$. On appelera* meta-état *un état de $Q^{\textit{VIS}}$.*
- $A^{\textit{VIS}} = A_I^{\textit{VIS}} \cup A_O^{\textit{VIS}}$ *avec* $A_O^{\textit{VIS}} = A_O^{PS^{\delta}} = A_O^{PS} \cup \{\delta\}$ *et* $A_I^{\textit{VIS}} = A_O^{PS^{\delta}} = A_I^{PS}$.
- $q_0^{\textit{VIS}} = q_0^{PS}$ *after*$_{PS^{\delta}}$ ϵ*, le meta-état initial de $PS_{\textit{VIS}}$ est l'ensemble des états accessibles par des actions internes depuis l'état initial de PS^{δ},*
- $\forall a \in A^{\textit{VIS}}, \forall P, P' \in Q^{\textit{VIS}}, P \xrightarrow{a}_{\textit{VIS}} P' \iff P' = P$ *after*$_{PS^{\delta}}$ a*. Comme d'habitude, on se restreint à la partie accessibles depuis le meta-état initial.*
- $Reject^{\textit{VIS}} = \{s \in Q^{\textit{VIS}} \mid s \cap Reject^{PS} \neq \emptyset\}$*, un meta-état de rejet est un ensemble d'états contenant au moins un état de rejet de PS (ou PS^{δ})*
- $Accept^{\textit{VIS}} = \{s \in Q^{\textit{VIS}} \mid s \cap Accept^{PS} \neq \emptyset\} \setminus Reject^{\textit{VIS}}$ *un meta-état accepteur est un ensemble d'états contenant au moins un état accepteur de PS (ou PS^{δ}) mais aucun état de rejet.*

En plus du comportement observable, PS_{VIS} conserve la notion de meta-état accepteur et de rejet qui permettront de sélectionner un cas de test. Pour définir ces notions dans PS_{VIS}, nous avons choisi de rejeter un comportement observable aussitôt qu'il correspond à au moins un comportement rejeté dans PS. C'est le choix qui semble le plus naturel.

Algorithme Le calcul de PS_{VIS} à partir de PS peut se décomposer en plusieurs opérations différentes:

1. le calcul de PS^{δ} par ajout de boucles étiquetées δ sur les états de blocage de PS,
2. le calcul du meta-état initial $q_0^{\text{VIS}} = q_0^{\text{PS}^{\delta}}$ after ϵ. et des meta-états P after$_{\text{PS}^{\delta}}$ a pour une action a et pour un meta-état $P \subseteq 2^{PS}$ accessible depuis q_0^{VIS}
3. le calcul des ensembles $Accept^{\text{VIS}}$ et $Reject^{\text{VIS}}$: il suffit de marquer les état dans $Reject^{\text{VIS}}$ aussitôt qu'un des états de PS^{δ} le constituant est dans $Reject^{\text{PS}}$ et de marquer un état dans $Accept^{\text{VIS}}$ s'il n'est pas dans $Reject^{\text{VIS}}$ et s'il contient un état de $Accept^{\text{PS}}$.

La première opération ne pose aucun problème pour les deadlocks et blocages de sortie puisqu'il suffit d'examiner l'ensemble des transitions tirables. Pour les livelocks, il faut identifier les boucles d'actions internes, autrement dit les CFC non triviale d'actions internes. Puisqu'on s'intéresse ensuite seulement aux traces de PS^{δ}, il est inutile d'ajouter une boucle δ sur chaque état d'une CFC non triviale de τ, il suffit d'en ajouter une sur la racine de CFC par exemple. Ceci découle de la propriété de stabilité des CFCs pour les propriétés d'accessibilité.

Le deuxième problème est en fait identique au problème de la déterminisation d'un automate d'état fini [**?**]. Il se décompose en deux opérations:

- le calcul de la τ-fermeture (ou ϵ-clôture) d'un état q, $\tau^*(q)$

– la déterminisation.

La τ-fermeture d'un ensemble d'états P se calcule généralement en parcourant le sous-graphe constitué des arcs étiquetés par τ à partir des états de P. L'automate déterministe se calcule par construction de sous-ensemble en utilisant la τ-fermeture: on calcule l'état initial $\tau^*(q_0)$ qu'on ajoute à un ensemble D sans le marquer. Ensuite, tant que D contient des états non marqués, on choisit un état T non marqué. Pour chaque action a, on calcule l'ensemble des successeurs d'états de T par a dans T' et on en fait la τ-fermeture $\tau^*(T')$. Si $\tau^*(T')$ n'est pas dans D, on l'ajoute sans le marquer.

Optimisation L'algorithme précédent pose un problème: chaque application de $\tau^*(T)$ est de complexité linéaire dans la taille du sous-graphe composé de τ accessible depuis T. Mais la τ-fermeture ne tient pas compte du fait qu'elle est utilisée par la déterminisation. Ainsi, si on considère la complexité de tous les appels à la τ-fermeture dans le calcul d'un automate déterministe, la complexité n'est plus linéaire. En effet, un appel à $\tau^*(T)$ pour un ensemble d'états T peut reparcourir une partie déjà calculée. En fait, l'ensemble des calculs de τ-fermeture est de complexité $O(n^2)$ en espace et $O(m_o.n^3 + m_o.n(m_i + m_o))$ en temps où n est le nombre d'états, m_o et m_i le nombre de transitions observables et internes.

Nous avons donc proposé dans [17] un algorithme de τ-fermeture et déterminisation qui se différencie de l'algorithme de [**?**] par la τ-fermeture. L'algorithme évite toute redondance de calcul au prix d'un stockage d'information supplémentaire et donc d'une complexité en espace plus importante. L'idée est de représenter un meta-état non pas comme un ensemble mais de le structurer en sa forêt d'arbre réduit de τ-CFC et de synthétiser sur chaque τ-CFC cfc_i l'ensemble des couples (q, a) où q est un état accessible après des τ et une action observable a. Le calcul de τ-CFC se justifie par la propriété de stabilité des CFCs pour l'accessibilité, ici des transitions observables. Ainsi, quand on accède de nouveau à un état q dans un nouvel appel à τ^*, il suffit de retrouver sa τ-CFC et on a toute l'information utile pour la déterminisation. Un meta-état est donc représenté de manière unique par l'ensemble des CFC racines de chaque arbre réduit, ce qui facilite la comparaison d'états. La τ-fermeture a alors une complexité $O(n + n.m_o)$ en espace et $O(n + m_i + m_o + 2n.m_o)$ en temps. La déterminisation est comme toujours de complexité exponentielle. *A faire:* $n^2 < n + n.m_o$?

On a vu que le calcul des livelocks et la τ-fermeture font appel au calcul de τ-CFCs. Ceci n'est bien sûr fait qu'une fois, les δ étant calculés pendant la tau-fermeture.

Un mot sur la minimisation: L'IOLTS PS^{VIS} construit n'est pas minimal. En effet pour pouvoir minimiser cet IOLTS, il faudrait le construire entièrement car les algorithmes de minimisation fonctionnent en arrière par raffinement de partition. Ceci n'est donc pas possible à la volée puisque nous construisons le graphe (en fait seulement une partie) en avant par une DFS. Cependant, nous appliquons une minimisation partielle limitée à la profondeur 1: deux meta-états de PS^{VIS} sont équivalents à profondeur 1 s'ils ont les mêmes transitions vers les mêmes états de PS^{δ}. Il suffit pour celà de regarder l'ensemble des couples (état,transition) synthétisés sur les CFC racines des meta-états.

Synthèse du graphe de test

PS_{VIS} représente les comportements observables de S et définit parmi ceux-ci, grâce à ses ensembles d'états $Accept^{\text{VIS}}$ et $Reject^{\text{VIS}}$, les comportements acceptés par l'objectif de test. L'étape suivante va consister à extraire de PS_{VIS} un IOLTS composé essentiellement de ses comportements acceptés. Afin de le faire correspondre à la définition d'un cas de test, il faudrait en faire un peu plus:

1. effectuer l'image mirroir afin de changer entrée en sortie et inversement,
2. le rendre complet en entrée dans tous les états où une réception est possible,
3. le rendre contrôlable,
4. définir les verdicts par les ensembles **Pass**, **Inconc** et **Fail**,

Dans un premier temps, nous n'allons pas le rendre contrôlable, ce qui va produire un IOLTS appelé *graphe de test complet* et noté CTG (pour *Complete Test Graph*). Nous séparons le problème de contrôlabilité du problème de calcul de graphe de test complet pour plusieurs raisons:

– le graphe de test complet s'avère être un IOLTS intéressant puisqu'il représente tous les tests correspondant à l'objectif,
– la résolution complète des conflits de contrôlabilité se fait par un algorithme différent de la sélection et ne fonctionne pas à la volée. On verra cependant qu'on peut en résoudre une partie pendant la sélection dans le cas où on ne souhaite pas calculer CTG.

Définition 12 *Le* graphe de test complet *CTG pour une spécification S et un objectif de test TP est l'IOLTS $CTG = (Q^{CTG}, A^{CTG}, \longrightarrow_{CTG}, q_0^{CTG})$, avec trois ensembles d'états marqués* **Pass** *et* **Inconc** *et* **Fail** *tels que :*

– $A^{CTG} = A_O^{CTG} \cup A_I^{CTG}$ *avec* $A_O^{CTG} \subseteq A_I^{VIS}$ *et* $A_I^{CTG} = A_O^{VIS}$ *(image miroir),*
– $Q^{CTG} = L2A \cup \textbf{Inconc} \cup \textbf{Fail}$*, avec*
  – $L2A = \{q \in Q^{VIS} \mid v \xrightarrow{*}_{VIS} Accept^{VIS}\}$*, i.e. l'ensemble d'états desquels $Accept^{VIS}$ est accessible,*
  – $Inconc = \{v \in Q^{VIS} \mid \exists u \in L2A, v \notin L2A,\ a \in A_O^{VIS},\ u \xrightarrow{a}_{VIS} v\}$*, les états de l'ensemble $Inconclusif$ sont les états n'appartenant pas à l'ensemble $L2A$ mais qui sont successeurs immédiats d'un état de $L2A$ par une émission dans PS_{VIS},*
  – $Fail = \{Fail\}$ *où* $Fail \notin Q^{VIS}$

  Comme d'habitude on se restreint à la partie accessible depuis l'état initial.
– $q_0^{CTG} = q_0^{VIS}$ *si* $q_0^{VIS} \in L2A$*, sinon CTG est vide,*
– $\rightarrow_{CTG} = \rightarrow_{L2A} \cup \rightarrow_{\textbf{Inconc}} \cup \rightarrow_{Fail}$ *où:*
  – $\rightarrow_{L2A} = \{(v, a, w) \in \rightarrow_{VIS} \mid\ v \in L2A \wedge w \in L2A\}$,
  – $\rightarrow_{\textbf{Inconc}} = \{(v, a, w) \in \rightarrow_{VIS} \mid\ v \in L2A \wedge w \in \textbf{Inconc} \wedge a \in A_o\}$
  – $\rightarrow_{\textbf{Fail}} = \{(v, a, Fail) \mid v \in L2A \wedge a \in A_O^{VIS} \wedge v \not\xrightarrow{a}_{VIS}\}$,
– $\textbf{Pass}^{CTG} = Accept^{VIS}$*, les états* **Pass** *sont les états de* $Accept^{VIS}$

Algorithme Le calcul de CTG est un problème d'accessibilité. En effet l'ensemble des états $L2A$ est l'ensemble des états desquels $Accept^{\text{VIS}}$ est accessible, c'est à dire l'ensemble des états qui satisfont la propriété de CTL $l2a = \exists DiamondAccept^{\text{VIS}}$. Comme les CFCs de PS^{VIS} sont stables pour les propriété d'accessibilité, il suffit de calculer le graphe réduit des CFC dont la racine (et donc tous les états) satisfait $l2a$ (plus précisément le sous-graphe composé des états de ces CFCs et des transitions dans et entre ces CFCs) . On adapte donc l'algorithme StrongConnect afin qu'il synthétise la propriété $l2a$ tout en construisant les transitions de $\rightarrow_{\text{L2A}}$ pendant le backtracking. Le calcul de l'ensemble **Inconc** et $\rightarrow_{\textbf{Inconc}}$ se fait aussi au backtracking de transitions de sorties vers des états dans $L2A$. L'état *Fail* et les transitions de $\rightarrow_{\textbf{Fail}}$ sont implicites psuique $\rightarrow_{fail}$ est défini par le complémentaire des sorties tirables. On peut bien sûr voir TGVloop comme une adaptation de l'algorithme de model-checking de $l2a = \exists \diamond Accept^{\text{VIS}}$ produisant l'ensemble des témoins de $l2a$ pour l'état initial (le sous-graphe de CTG restreint aux états de $L2A$). L'algorithme appelé TGVloop est illustré par la figure **??**. Il est de complexité linéaire en temps et espace comme StrongConnect.

Suppression des conflits de contrôlabilité

Si on revient à la définition d'un cas de test (Définition 6), on constate que CTG possède toutes les propriétés d'un cas de test, excepté la contrôlabilité: il se peut que dans un état q de CTG on ait le choix entre plusieurs sorties ou entre sorties et entrées (voir figure 11).

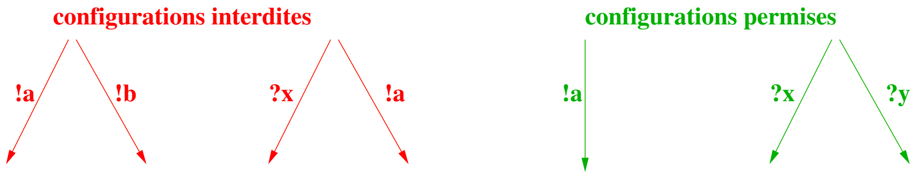

FIG. 11 – *Conflits de contrôlabilité*

Résoudre ces conflits de contrôlabilité consiste à extraire de CTG un sous-graphe contrôlable sans remettre en cause les propriétés d'un cas de test acquises dans CTG: les états dans **Fail** et **Inconc** peuvent uniquement être atteints par des réceptions, accessibilité à un verdict depuis tout état, complétion en entrée pour les états où une entrée est possible.

Dans un état en conflit de contrôlabilité, il suffit d'élaguer des transitions de l'une des façons suivantes:

– soit garder une seule sortie en élaguant les autres sorties et entrées éventuelles,

– soit garder seulement toutes les entrées en élaguant toutes les sorties.

Bien sûr appliquer ces règles peut rendre des états inaccessibles qui seront supprimés. Ce qu'il faut éviter c'est que des états pouvant mener à un état **Pass** (les états de $L2A$) n'y mènent plus (ils mènent alors à **Inconc** et **Fail**). Pour éviter celà on parcourt CTG en remontant des états **Pass** vers l'état initial, c'est à dire en utilisant la relation de transition $\rightarrow_{\text{CTG}}^{-1}$. On note CTG^{-1} le graphe de CTG avec la relation de transition $\rightarrow_{\text{CTG}}^{-1}$.

Stratégie en largeur: La première stratégies envisagée est un parcours en largeur de CTG^{-1} à partir des états de l'ensemble **Pass**. Quannd un état de conflit q est atteint, il l'est par la plus courte séquence menant à l'ensemble **Pass**. Si la dernière transition de cette séquence est une sortie, on supprime les autres transitions de $\rightarrow_{\text{CTG}}$ issues de q, si c'est une entrée, on supprime toutes les sorties. Puisque q est accessible depuis q_0^{CTG} et qu'on n'élague que les transitions sortantes, on finira par atteindre q_0^{CTG}.

Stratégie par calcul de CFC: On peut remarquer que l'élagage modifie l'accessibilité depuis l'état initial dans CTG, c'est à dire l'accessibilité à l'état initial dans CTG^{-1}. Notons $elag(CTG^{-1})$ le graphe CTG^{-1} modifié par élagage. Le problème peut alors être reformulé en le calcul du sous-graphe de $elag(CTG^{-1})$ défini par les états desquels l'état initial est accessible i.e. possédant la propriété de CTL $l2init = \exists \diamond q_0^{\text{CTG}}$. Poser le problème en termes d'accessibilité nous amène une fois de plus au calcul de composantes fortement connexes et à un autre algorithme adapté de StrongConnect. L'avantage de cet algorithme est qu'il permet d'effectuer de la récupération de mémoire (*garbage collection*) pendant le parcours: les CFC n'ayant pas la propriété sont éliminées. Remarquons que le sous-graphe $elag(CTG^{-1})$ n'est pas donné *a priori* mais qu'il est calculé pendant le parcours et qu'il dépend bien sûr de l'ordre de parcours. La propriété $l2init$ est bien entendu vraie au départ dans tous les états de CTG^{-1} mais est modifiée par l'élagage dans $elag(CTG^{-1})$.

Stratégie à la volée: Nous avons présenté deux algorithmes de résolution de conflits basés sur l'élagage de CTG par parcours en arrière. Ces deux algorithmes nécéssitent de construire CTG entièrement. Or pour calculer CTG entièrement, il faut aussi construire entièrement PS^{VIS}, PS et S, sauf si des états $Reject$ permettent de couper l'exploration. L'intérêt de la synthèse de test à la volée s'en trouve donc limité. Or on n'a pas forcément utilité de produire CTG. Dans ce cas, une autre solution consiste à effectuer l'élagage à la volée pendant l'algorithme TGVloop. Supposons que TGVloop dépile de DFS_stack un état q' de PS^{VIS} menant à $Accept^{\text{VIS}}$ (i.e. $q' \in L2A$), soit q son prédécesseur (le nouveau sommet de pile), et a l'étiquette de la transition de q à q'.

1. si a est une entrée de PS^{VIS} (donc une sortie du testeur), on peut élaguer toutes les autres transitions issues de q (déjà explorées ou non).
2. si a est une sortie de PS^{VIS} (donc une entrée du testeur), on peut élaguer toutes les entrées.

Dans le premier cas, les transitions dejà explorées ne mènent pas à $Accept^{\text{VIS}}$, sinon on aurait déjà appliqué l'une des deux règles et supprimé l'entrée a. Dans le second cas, les entrées déjà explorées ne mènent pas à $Accept^{\text{VIS}}$, sinon on aurait déjà appliqué la première règle et supprimé l'entrée a.

L'application de ces deux règles d'élagage pendant TGVloop peut permettre de régler certains conflits de contrôlabilité au plus tôt et éviter ainsi de construire certaines parties de CTG. Mais tous les conflits ne peuvent pas être résolus ainsi. En effet, quand on dépile q' de DFS_Stack, on ne sait pas toujours s'il est dans $L2A$. Au pire on ne le sait de façon définitive que lorsqu'on a calculé sa CFC, donc lorsqu'on le dépile de CFC_Stack. On ne peut donc pas appliquer l'une des deux règles précédentes. La figure 12 représente quelques cas de ce type.

FIG. 12 – *Quelques cas de conflits de contrôlabilité problématiques*

Remarque: dans le cas où il n'y a pas de boucles (toutes les CFC sont triviales), les nœuds sont dépilés des deux piles en même temps, donc tous les conflits peuvent être résolus.

Le graphe synthétisé par TGVloop+contrôlabilité n'est donc pas toujours contrôlable. Pour règler les conflits résiduels, il faut appliquer l'un des deux algorithmes précédents *a posteriori*. On pourrait introduire un de ces algorithmes dans TGVloop afin de règler tous les conflits résiduels au moment du dépilement de CFC.

3.4 Vérification de tests

Nous avons jusqu'ici présenté les algorithmes de synthèse de test dans une démarche automatique à partir d'une spécification formelle. Nous sommes conscients cependant que des tests continueront d'être écrits à la main. Cependant, à partir de l'instant où une spécification formelle existe, qu'on a défini une relation de conformité, et qu'on peut modéliser les comportements des cas de test, il est possible de vérifier la correction des cas de test. Nous définirons plus tard ce qu'on entend par correction des cas de test. Il est même possible de corriger certaines erreurs. Nous avons exploré ceci et expérimenté l'idée sur une étude de cas de taille significative (publié dans [15]). Ceci nous a permis de découvrir et corriger un certain nombre d'erreurs dans une suite de test manuelle.

Un autre usage de la vérification des cas de test à aussi été exploré pendant le codage de notre prototype TGV. Il s'agissait alors, dans une démarche de validation de l'outil, de vérifier la correction des cas de test produits par TGV. En effet, la validation d'un tel outil est difficile. Un test consiste en une spécification, un objectif et le cas de test à obtenir. Ceci est utile pour quelques exemples jouets et quelques problèmes typiques à vérifier mais s'avère insuffisant aussitôt qu'on considère des spécifications réalistes et que les tests ne sont pas uniques pour un objectif donné. Il faudrait alors comparer un test obtenu avec tous les tests corrects possibles, ce qui peut s'avérer fastidieux, voire impossible. Notre démarche de validation a donc consisté à vérifier cahque test produit en fonction de propriété de correction définies.

L'idée de la vérification des cas de test n'est pas nouvelle [**?**]. Cependant, à notre connaissance aucun de ces travaux ne prennent en compte le non-déterminisme. Pour nous, ceci n'a pas posé problème puisque nous réutilisons les mêmes notions et algorithmes de déterminisation en vérification de test qu'en synthèse de test.

Propriétés de correction des cas de test

On peut classifier les propriétés de correction des tests en plusieurs catégories:

1. les propriétés qui ne concernent que le cas de test lui-même. Celles-ci sont définies par la définition 6. En particulier on peut vérifier l'accessibilité à un verdict, la contrôlabilité, la complétion en entrée dans les états de réception. Elles peuvent être vérifiées par un parcours du cas de test. On supposera qu'elles ont été vérifiées et sont satisfaites.
2. les propriétés qui concernent le cas de test et la spécification. On s'intéressera à deux propriétés du deuxième type: le non-biais (un cas de test ne rejette pas d'implémentation conforme) et le non laxisme (le cas de test doit rejeter les implémentations non-conformes sur les traces qu'il teste). Bien qu'elles soient définies en fonction des implémentations, ces propriétés peuvent être vérifiées par un parcours d'un produit entre spécification et test.
3. les propriétés qui concernent le cas de test, la spécification et l'objectif de test. Elles caractérisent la correction des verdicts en fonction de l'objectif, en particulier que les verdicts *Pass*et **Inconc**sont correctement assignés. Ces propriétés nécessitent un parcours d'un produit à trois: spécification, objectif de test et cas de test. Nous ne nous y intéresserons pas ici car nous les avons définies dans le cadre plus large du test symbolique.

Rappelons la propriété de non-biais (définition 7). Cette propriété est définie par une quantification sur toutes les implémentations possibles:
TC est *non biaisé* pour S et **ioco** si et seulement si
$\forall IUT, IUT$ **ioco** $S \rightarrow \neg(TC mayreject IUT)$.

On explicite **ioco**, et $mayreject$, et on constate qu'on peut éliminer la quantification sur l'implémentation puisqu'elle est redondante avec la quantification sur les traces de S^δ et simplement définir le non-biais de TC en fonction des traces de S:

TC non-biaisé par rapport à Set**ioco** $\Leftrightarrow \forall \sigma \in Traces(S^\delta), \forall a \in A^s_O, a \in Out(S^\delta$ after$_\sigma$ $) \rightarrow$ TC after$_\sigma$ $.a \cap$ **Fail** $= \emptyset$.

Un cas de test est non laxiste s'il ne possède pas de comportement qui pourrait rejeter une IUT non conforme sans le faire. Formellement: TC est non laxiste pour S et **ioco** $\Longleftrightarrow$ $\forall IUT, \forall \sigma \in Traces(S^\delta) \cap Traces(TC), \forall a \in A^s_O, (a \in Out(IUT$ after$_\sigma$ $) \wedge a \notin Out(S^\delta$ after$_\sigma$ $)) \rightarrow (TC$ after$_\sigma$ $.a) \subseteq Fail$.

On peut également éliminer la quantification sur l'implémentation et exprimer le non-laxisme comme suit:
$\forall \sigma \in Traces(S^\delta) \cap Traces(TC), \forall a, a \notin Out(S^\delta$ after$_\sigma$ $) \rightarrow (TC$ after$_\sigma$ $.a) \subseteq Fail$.

Pour les deux propriétés, on peut faire les observations suivantes. TC étant déterministe, TC after$_\sigma$ $.a$ est un singleton. Donc $(TC$ after$_\sigma$ $.a) \subseteq Fail$ est équivalent à $(TC$ after$_\sigma$ $.a) \cap Fail \neq \emptyset$ et peut lui être substitué dans la propriété de laxisme. Quand $\sigma \notin Traces(TC)$ on a TC after$_\sigma$ $= \emptyset$. Dans la propriété de non-biais, il suffit de vérifier TC after$_\sigma$ $.a \cap$ **Fail** $= \emptyset$ pour $\sigma.a \in Traces(TC) \cap Traces(S^\delta)$ Enfin on note $in_{\neg Fail}(TC$ after$_\sigma$ $) =_{def} \{a \in A^S_O \cup \{\delta\} \mid (TC$ after$_\sigma$ $.a) \cap Fail = \emptyset\}$.

Finalement, en combinant les deux propriétés, on obtient qu'un cas de test est non-biaisé et non laxiste si et seulement si
$\forall \sigma \in Traces(S^\delta) \cap Traces(TC), \forall a \in A^S_O \cup \{\delta\}, Out(S^\delta$ after$_\sigma$ $) = in_{\neg Fail}(TC$ after$_\sigma$ $)$

Vérification et correction des cas de test

Il en découle que l'algorithme de vérification de non-biais et non-laxisme est basé sur le calcul de la τ-fermeture et déterminisation de S et le produit avec TC. Ceci peut s'effectuer à la volée en guidant la construction par TC et en ne calculant que la partie de S^δ (et donc de S) correspondant aux traces de TC. Pour chaque état du produit, il suffit de comparer l'ensemble des sorties dans S^δ avec l'ensemble des entrées de TC qui ne produisent pas *Fail*. Si l'algorithme constate une erreur, il peut aussi la corriger. Dans le cas du biais, la correction consiste à remplacer tout verdict *Fail* incorrect par un verdict *Inconc*. Dans le cas du laxisme, la correction consiste à produire un verdict *Fail* et à couper l'état atteint.

3.5 Synthèse et vérification à la volée: les outils TGV et VTS

Les algorithmes de synthèse de cas de test et de vérification de cas de test présentés ci-dessus ont été implémentés dans les outils TGV et VTS. Nous allons rapidement décrire leur architecture fonctionnelle et leur implémentation. Puis nous parlerons d'études de cas et de transfert industriel.

Architecture fonctionnelle

L'architecture fonctionnelle de TGV et VTS (voir figures **??** et **??**) a été conçue avec à l'esprit deux priorités:

– la construction/vérification à la volée.
– l'indépendance vis à vis des langages de spécification et la factorisation des codes

Il découle de ces considérations une architecture basée sur des interfaces de programmation d'application (API pour *Application Programming Interface*). Mise à part l'API fournie par l'outil CADP, toutes les API fournissent un IOLTS sous forme de trois fonctions: **init** qui rend l'état initial, **enabled** qui prend en entrée un état et calcule l'ensemble des transitions tirables et **succ** qui pour un état et une transition tirable, calcule l'état successeur.

Pour TGV, les entrées sont une spécification et un objectif de test. l'API fournissant l'IOLTS de la spécification S est offerte par les fonctions $init_{\mathrm{S}}$, $enabled_{\mathrm{S}}$, $succ_{\mathrm{S}}$.

L'API founissant l'objectif de test TP est du même type: $init_{\mathrm{TP}}$, $enabled_{\mathrm{TP}}$, $succ_{\mathrm{TP}}$ et définit également les fonction caractéristiques $accept_{\mathrm{TP}}$ et $reject_{\mathrm{TP}}$ des ensembles $Accept_{\mathrm{TP}}$ et $Reject_{\mathrm{TP}}$.

A partir de l'API de S et de l'API de TP, une première couche produit l'API du produit $PS = S \times TP$ sous forme de cinq fonctions permettant son parcours: $init_{\mathrm{PS}}$, $enabled_{\mathrm{PS}}$, $succ_{\mathrm{PS}}$, $accept_{\mathrm{PS}}$ et $reject_{\mathrm{PS}}$. Celle-ci est ensuite utilisée par la couche Fermdet qui réalise l'ajout des δ, la τ-fermeture et la déterminisation en fournissant les cinq fonctions permettant la construction de PS^{VIS} i.e. $init_{\mathrm{VIS}}$, $enabled_{\mathrm{VIS}}$, $succ_{\mathrm{VIS}}$, $accept_{\mathrm{VIS}}$ et $reject_{\mathrm{VIS}}$.

TGVloop utilise cette API de construction de PS^{VIS} pour construire soit un graphe de test complet CTG, soit un graphe de test TG (si on essaie de résoudre des conflits

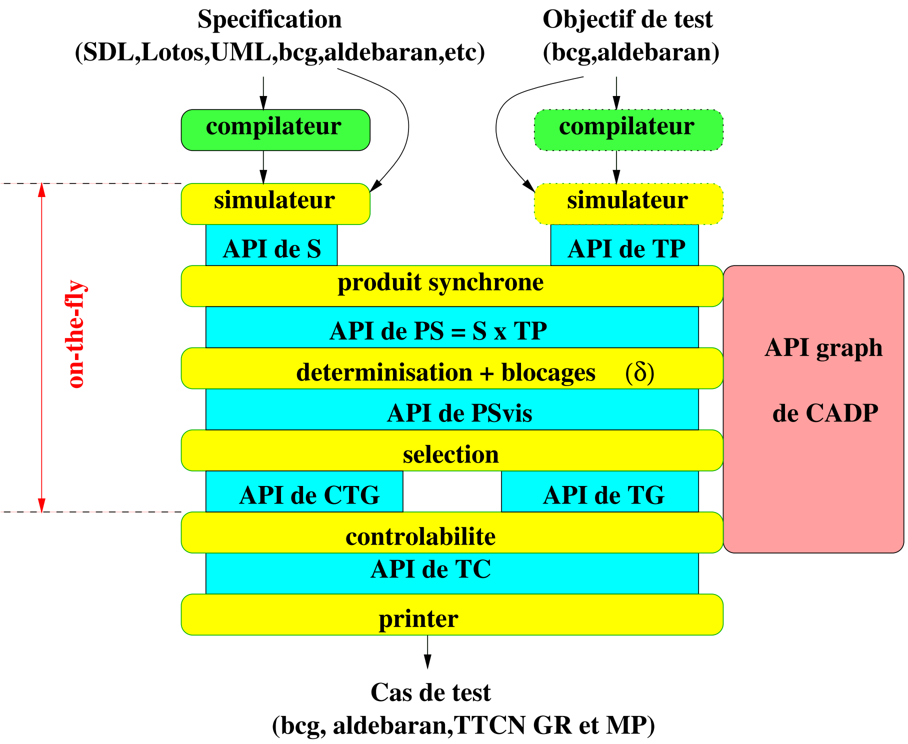

FIG. 13 – *Architectures de TGV*

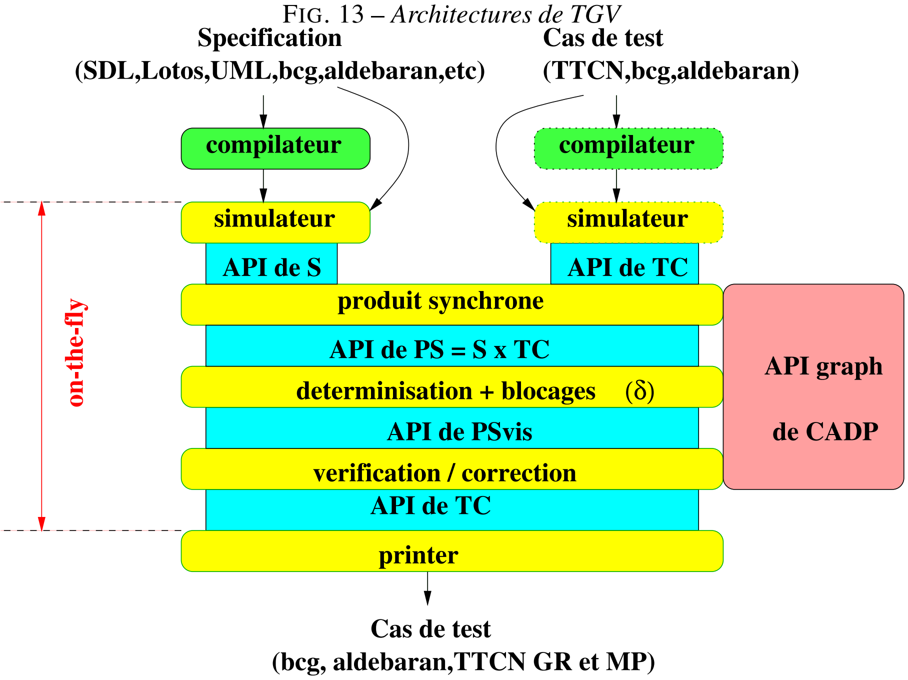

FIG. 14 – *Architecture de VTS*

de contrôlabilité). L'API produite est alors (dans le cas de CTG) $init_{\text{CTG}}$, $enabled_{\text{CTG}}$, $succ_{\text{CTG}}$, $pass_{\text{CTG}}$ et $inconc_{\text{CTG}}$.

Si on veut extraire un cas de test contrôlable de CTG (ou TG) on a aussi besoin de fonctions de parcours supplémentaires nécéssaires au parcours de CTG^{-1} i.e. $incoming_{\text{CTG}}$ et $pred_{\text{CTG}}$. La résolution des conflits (résiduels) produit l'API d'un cas de test TC par les fonctions $init_{\text{TC}}$, $enabled_{\text{TC}}$, $succ_{\text{TC}}$, $pass_{\text{TC}}$ et $inconc_{\text{TC}}$.

Enfin, on utilise également une API fournie par la boîte à outils CADP [9] qui permet le stockage des états et transitions des parties des IOLTS calculés.

L'architecure de VTS est plus simple mais très similaire à celle de TGV.

Implémentation

L'implémentation réelle de TGV (et VTS) respecte quasiment l'architecture fonctionnelle. L'IOLTS de la spécification est fourni soit de manière explicite par un graphe dans un format donné(BCG ou Aldébaran) soit de manière implicite par un compilateur d'un langage de spécification (le générateur de simulateur d'ObjectGéode pour SDL, Open/Caesar pour Lotos, Umlaut pour UML)[7]. Dans les deux cas l'IOLTS peut être parcouru (ou construit) par les fonctions de l'API du type de celles décrites plus haut. Ceci donne une très grande indépendance à TGV vis à vis des langages de spécification et a permis de factoriser tout le code en dessous de cette API. L'objectif est en réalité fournit explicitement au format BCG ou Aldébaran mais pourrait également être fournit dans un langage de plus haut niveau par compilation vers du code de simulation. Le calcul du produit $PS = S \times TP$ est intégré au parcours de PS dans Fermdet. Le graphe de test complet CTG ou le cas de test TC sont produits aux formats BCG ou Aldébaran. Optionellement on y intègre deux temporisateurs TAC et TNOAC. TAC est chargé de détecter l'absence de réponse sur une entrée attendue et TNOAC le blocage attendu correspondant à l'action δ. Les cas de test peuvent ensuite être traduits au format TTCN.

L'implémentation de TGV et VTS a été réalisée en C. Grâce à la factorisation de code, l'ensemble du code source (Fermdet, TGVloop, VTS et API diverses vers langages de spécification) est relativement petit (environ 10 000 lignes de code) .

3.6 Comparaison et complémentarité avec d'autres techniques

Travaux de Twente

A quelques détails près, l'outil TorX développé à Twente est basé sur la même théorie du test que TGV: modèles similaires, même relation de conformité. L'algorithme de génération est différent puisqu'au lieu d'utiliser des objectifs de test pour la sélection des cas de test, TorX fait un tirage aléatoire sur les entrées de la spécification ou demande à l'utilisateur d'en choisir une. D'autre part cette génération est simultanée à l'exécution des tests sur une implémentation. Cette exécution simultanée a pour conséquence que TorX ne parcourt qu'une séquence d'actions observables de la spécification sans avoir besoin de mémoriser les états. D'autre part pour chaque protocole testé, il

7. Les prototypes des fonctions fournies par le compilateur SDL n'étant pas complètement identiques à celles de Lotos et UML, il existe une couche d'adaptation spécifique à SDL non représentée ici.

est nécéssaire de développer une couche d'adaptation spécifique pour passer du niveau tests abstraits aux tests exécutables. La τ-fermeture et déterminisation utilise l'algorithme classique et ne se préoccupe pas des CFC de τ puisqu'il est supposé qu'il n'en existe pas (pas de livelock). Une version de TorX a été développé dans SPIN pour des spécifications Promela. Une autre est interfacée avec CADP pour des spécifications Lotos.

Nous avons préféré découplé le problème de la synthèse de test de celui de l'exécution (traité par ailleurs dans notre équipe) car celà nous permet de comparer les tests produits avec des tests écrits à la main. Nous avons de plus le sentiment qu'il est parfois préférable de découpler les problèmes pour des raisons de contraintes temporelles. En effet le calcul des tests pendant l'exécution peut ralentir celle-ci et provoquer des expirations de temporisateurs de protocoles inopinées. Cependant un outil comme TorX est utile pour détecter rapidement des erreurs dans l'implémentation alors quún outil comme TGV est plus utile dans la synthèse d'une suite de test. Un outil similaire à TorX serait cependant réalisable à partir de TGV. Il suffirait de modifier l'algorithme principal pour permettre le choix aléatoire ou manuel d'entrées, d'appliquer Fermdet sur S comme dans VTS et d'utiliser une couche d'adaptation similaire à celle de TorX.

Test et théorie des jeux

Solofo Ramangalahy a étudié dans sa thèse l'apport de la théorie des jeux dans la modélisation du test de conformité: un test est un jeu à deux joueurs (l'implémentation et le testeur) où le testeur essaie de détecter la non-conformité de l'implémentation ou la satisfaction d'un objectif de test. Une stratégie gagnante du testeur à ce jeu consiste à maximiser le contrôle qu'il peut avoir sur l'implémentation pour l'amener à la satisfaction de l'objectif. La stratégie calculée sur un produit entre la spécification (un IOLTS) et un objectif de test (un automate) est définie récursivement par un coût initialement nul des états correspondant à la satisfaction de l'objectif et prenant en compte la longueur des séquences d'actions d'entrées de la spécification (contrôlables) et les possibles pertes de contrôle dus à des choix de sorties. Le testeur a une stratégie gagnante car la spécification (et l'implémentation) est supposée fortement connexe (on peut donc toujours revenir à l'état initial) et l'implémentation joue de façon équitable (elle montre tous ses comportements en cas de non déterminisme) alors que ce n'est pas le cas du testeur. Vue la forme de la fonction de coût, la stratégie est calculée en arrière.

Nous avons collaboré avec S. Ramangalahy sur l'éventualité d'utiliser cette technique dans l'outil TGV. Une première possibilité est d'utiliser le calcul de test le plus contrôlable sur le graphe de test complet CTG. Mais on perd ainsi en grande partie l'avantage de la synthèse à la volée. Dans notre approche, la spécification et CTG ne sont pas supposés fortement connexes mais on peut raisonnablement faire cette hypothèse. L'autre possibilité est d'intégrer la fonction de coût pendant le backtracking de l'algorithme de TGV. On peut trouver des conditions permettant d'éviter l'exploration de certaines branches. On peut aussi, comme dans les programmes de jeux comme les échecs, utiliser des heuristiques au prix de ne pas calculer le test le plus contrôlable.

Autres techniques fondées sur l'utilisation d'objectifs de test

D'autres techniques utilisent des objectifs de test pour la sélection des tests. C'est le cas en particulier de l'outil Samstag [**?**] qui est à la base de TTCNLink (Telelogic) où les objectifs sont fournis sous formes de MSC (*Message Sequence Charts*) et la spécification est donnée en SDL. Il est assez difficile d'avoir une opinion scientifique sur cet outil car aucune théorie sous jacente n'est décrite. On ne sait donc pas quelle relation de conformité est testée par les tests générés. Il est clair cependant que la question du non-déterminisme est éludée car si une trace d'un MSC correspond à plusieurs comportements de la spécification, le calcul du test est impossible. Les objectifs de test sont beaucoup moins abstraits que ceux de TGV puisqu'un MSC doit décrire une suite d'actions consécutives de la spécification. Le calcul des tests est donc très simplifié puisqu'il suffit de calculer les traces communes à la spécification et de l'objectif. Du coup, les tests produits sont aussi très simples puisqu'ils consistent en une séquence principale menant à *Pass*et des actions menant à *Inconc*en cas de choix dans les sorties de la spécification.

Comparaison avec les techniques fondées sur les automates

Dans ces techniques, le modèle n'admet en général pas de non-déterminisme, ni même de non-détermisme observable (choix entre plusieurs sorties). De plus, les machines n'ont que des actions observables. Une phase préalable de τ-réduction est donc aussi nécéssaire.

4 Asynchronisme et distribution de tests

4.1 Prise en compte de l'asynchronisme dans la génération de tests

Il arrive parfois que l'architecture de test soit telle que la communication avec le système que l'on veut tester ne soit pas directe car celui-ci est enfoui dans un système englobant. On parle alors de test dans un contexte. Intuitivement, on voit bien alors que les comportements de l'implémentation sont déformés par le contexte et que par conséquent le testeur perd du contrôle et de l'observation sur le système à tester par rapport à une communication directe. Dans le test de conformité de protocole, ce type de problème arrive en particulier lorsque l'on fait du test à distance (*remote testing*). La communication est alors asynchrone à travers un médium de communication que l'on suppose de discipline fifo alors que nous l'avons supposée jusqu'ici synchrone. Verhard, Tretmans et Brinksma /citeVerhard-Tretmans-Brinksma-IWPTS93 ont montré que les tests "synchrones" ne sont pas corrects en environnement asynchrone: les verdicts produits ne sont pas corrects. En termes de propriétés de cas de test, on peut montrer que des tests non-laxistes et non-biaisés en environnement synchrone peuvent être biaisés et/ou laxistes en environnement asynchrone.

Il faut donc définir une relation de conformité qui prenne en compte le contexte:
I **ioco**$_{\mathcal{A}}$ $S =_{def} \mathcal{A}(I)$ **ioco** $\mathcal{A}(S)$
où pour un IOLTS M, $\mathcal{A}(M)$ est l'IOLTS obtenu en transformant M par le contexte asynchrone, c'est à dire en prenant en compte les comportements des files fifo entre M

et son environnement. On considère un nombre fini de files fifo non bornées[8]. $\mathcal{A}(M)$ se définit formellement en quelques règles. Les actions observables de M deviennent des actions internes agissant sur les contenus de file. Les actions observables sont les émissions de messages depuis les files vers l'environnement et les blocages (δ), les réceptions dans les files de messages provenant de l'environnement.

Un problème de cette définition est que **ioco** et **ioco**$_{\mathcal{A}}$ ne sont pas comparables. En effet I**ioco**$S \not\to I$**ioco**$_{\mathcal{A}}S$ et I**ioco**$_{\mathcal{A}}S \not\to I$**ioco**$S$ comme le montre l'exemple de la figure 14.

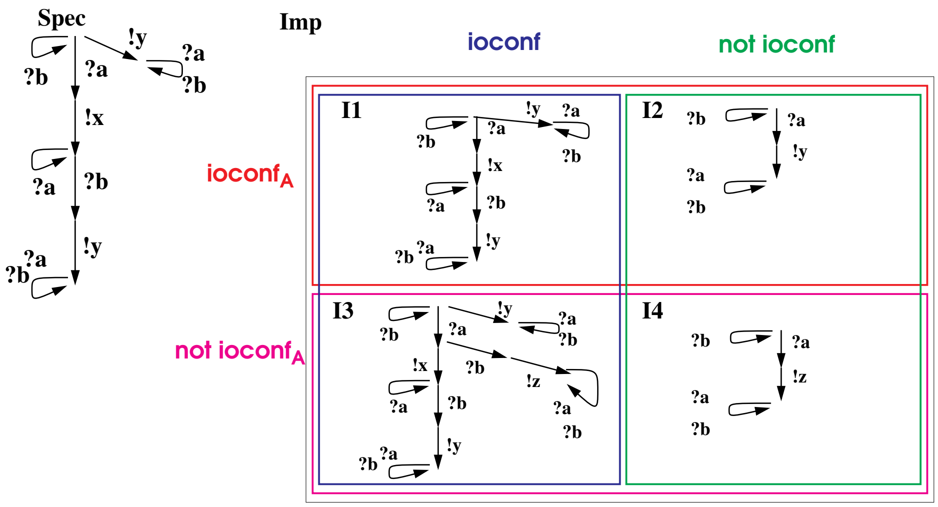

FIG. 15 – *Difference between synchronous and asynchronous testing*

On voudrait toutefois obtenir I**ioco**$S \to I$**ioco**$_{\mathcal{A}}S$ et de façon plus générale I**ioco**$S \to \mathcal{C}(I)$**ioco**$\mathcal{C}(S)$. Pour celà on peut se restreindre à des transformation monotone $\mathcal{C}$ i.e. conservant l'inclusion des traces et blocages: $Traces(M^{\delta}) \subseteq Traces(M'^{\delta}) \Rightarrow Traces(\mathcal{C}(M)^{\delta}) \subseteq Traces(\mathcal{C}(M')^{\delta})$). On suppose aussi que S et I sont complètes en entrée. Cette hypothèse n'est pas vraiment une restriction bien qu'on ait jusque là supposé que I seulement était complète en entrée. En effet, si S n'est pas complet en entrée i.e. $\exists \sigma, \exists a \in A_I^{\mathrm{M}}, S$ after$_\sigma$ $\stackrel{a}{\not\Rightarrow}_S$, on peut le transformer de la façon suivante: on ajoute un état puits (bouclant sur A_{M}) et pour tous les σ et les a en question, dans tous les états de S after$_\sigma$, on ajoute une transition vers l'état puits (ceci peut se faire durant la déterminisation). On peut alors montrer que $Comp(S)$ et S ont les mêmes implémentations conformes. On a donc I**ioco**$S \iff I$**ioco**$Comp(S) \to \mathcal{C}(I)$**ioco**$\mathcal{C}(Comp(S))$.

Le problème reste de trouver des transformations monotones. Supposons par exemple que $\mathcal{C}$ soit la composition parallèle avec un processus C. L'inclusion des traces est bien conservée mais pas l'inclusion des blocages: la composition parallèle peut ajouter des blocages dans le plus petit IOLTS qui n'existent pas dans le plus grand. En clair, l'inclusion des traces de l'automate suspendu n'est pas une pré-congruence pour l'opération

8. Pour que $\mathcal{A}(M)$ soit toujours fini, il faudrait supposer que les files sont bornées, mais ceci rajouterai potentiellement des blocages, ce qu'on voudra éviter

de composition parallèle avec un IOLTS fixé. Il faudrait donc recommander, avant de synthétiser des tests à partir de $S||C$ de vérifier que C ne rajoute pas plus de blocage à une implémentation conforme pour **ioco** qu'à la spécification.

La transformation $\mathcal{A}$ est par contre monotone. Pour les traces c'est évident. Pour les blocages on peut voir que la transformation n'introduit pas de blocage de sortie ni de deadlock (si les files sont non bornées). Des livelocks peuvent apparaître dues à des boucles d'émissions mais à la fois dans le plus petit et le plus gros IOLTS.

4.2 Test synchrone en environnement asynchrone

Nous avons vu ci-dessus qu'un contexte asynchrone préservait la conformité pour des spécifications complètes (ou complétées). Nous nous attaquons maintenant au problème inverse: le contexte asynchrone peut-il préserver la non-conformité ? En d'autres termes, si une implémentation I est non conforme à sa spécification S pour **ioco**, l'est elle encore en environnement asynchrone? La réponse est non comme le montre l'exemple de la figure 14. On peut toutefois remarquer que le problème vient du désordre provoqué par l'insertion des files entre l'environnement (donc le testeur) et le système: détecter la non conformité pour **ioco** nécéssite de connaître l'ordre d'occurrence des événements sur l'implémentation alors que l'on ne peut observer que l'ordre d'occurrence de leur cause (pour les entrées) ou conséquence (pour les sorties) à l'interface avec l'environnement.

Nous avons essayé dans [16] de résoudre ce problème pour une relation de conformité plus faible que **ioco** et **ioconf** puisqu'elle ne considère pas les blocages. Pour des IOLTS input complets, cette relation est l'inclusion de trace. Nous avons d'abord résolu le problème dans le cas d'une seule file fifo. La première possibilité investiguée est d'utiliser une fonction de codage st intercalée entre le système et la file fifo qui code l'ordre d'occurrence des événements observables: elle compte et estampille les sorties avant de les émettre vers la file et compte les entrées avant de les délivrer au système.

On peut alors montrer que I est conforme à S si et seulement si $\mathcal{A}(st(I))$ est conforme à $\mathcal{A}(st(S))$ (ou $\mathcal{A}(st(Comp(S)))$ si S n'est pas input complète). Il en résulte que pour tester I en environnement asynchrone avec la même puissance de test qu'en environnement synchrone, il suffit de calculer $\mathcal{A}(st(Comp(S)))$ et synthétiser des tests pour $\mathcal{A}(st(Comp(S)))$. A noter que ceci peut se faire à la volée, ce qui permet d'éviter la construction complète de $\mathcal{A}(st(Comp(S)))$ et de se focaliser sur des objectifs de test pertinents.

Plutôt que de calculer des tests sur $\mathcal{A}(st(Comp(S)))$ afin d'anticiper la déformation $\mathcal{A}$ dans les tests caculés, une autre possibilité est d'utiliser les tests synchrones synthétisés à partir de S et d'utiliser une mécanisme de décodage de st noté $\bar(st)$ implémenté sur le testeur (voir 15). Pour un cas de test TC de S, $\bar(st)$ calcule la prochaine émission à envoyer à I et décode la séquence de réceptions de façon à émettre un verdict en fonction du comportement de I et non pas celui de $\mathcal{A}(st(I))$. Le contrôle du testeur sur l'implémentation étant restreint par la communication asynchrone, on ne peut plus se limiter à des tests contrôlables comme dans le cas synchrone. Il faut permettre le choix entre une émission et des réceptions, ce que nous appelons *semi-contrôlable*.

On peut envisager d'étendre ces résultats pour une relation de conformité plus forte comme **ioco** qui permet d'observer les blocages par des temporisateurs. Il est clair

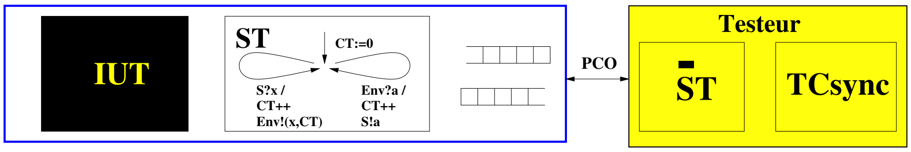

FIG. 16 – *Test synchrone en environnement asynchrone*

qu'il faut alors considérer les échéances de temporisateurs comme des sorties de l'implémentation et par conséquent gérer ces temporisateurs sur l'implémentation.

On peut également étendre ces résultats à plusieurs files fifo. L'information nécéssaire au testeur pour émettre un verdict sur réception d'un message est a priori la séquence des événements précédant son émission par l'implémentation. Il est possible en théorie d'associer cette séquence à cet événement mais le coût en communication est alors élevé. Dans le cas d'une seule file, cette information est codée par un seul entier. En effet les deux sous-ordres restreints respectivement aux émissions et aux réceptions sont conservés par une file fifo donc si on reçoit un message, on a déjà reçu tous ceux émis auparavant par l'implémentation et l'entier suffit à savoir quelles émissions du testeur ont été reçues par l'implémentation. Ceci n'est plus le cas pour plusieurs files car globalement la politique n'est plus fifo. Associer à une émission de l'implémentation toute la séquence d'événements la précédant permettrait de décoder l'ordre au fur et à mesure mais au prix d'une redondance d'information importante. On peut cependant réduire l'information à véhiculer si on accepte de différer l'émission d'un verdict. Par exemple en codant un événement par un entier et la sous-séquence d'événements depuis la dernière réception de l'implémentation.

L'intérêt pratique de ce travail reste toutefois encore à démontrer. En effet, on suppose qu'un mécanisme de codage instrumente l'implémentation. On sort donc du cadre strict du test boîte noire pour rendre plus précis le test en environnement asynchrone. Cette instrumentation a un coût qu'il faut pouvoir justifier.

4.3 Synthèse de tests répartis

Il est assez courant, pour des raisons intrinsèques à une application, que l'architecture de test soit décentralisée Dans ce cas, les cas de tests doivent être répartis. L'écriture de tels tests relève donc de la conception d'algorithmes répartis et est par conséquent très difficile. La difficulté provient à la fois de la communication entre testeurs et la communication entre testeurs et implémentation. Une technique automatique serait donc profitable.

Une première approche possible est de synthétiser d'abord un test centralisé à partir de la spécification et de le distribuer ensuite en un ensemble de tests répartis. Suivant cette approche, nous avons proposé un shéma de distribution d'un test centralisé en un ensemble de tests répartis et montré qu'il permettait permettait de conserver la même puissance de test [**?**].

On suppose que la communication entre testeurs et implémentation est synchrone. Si ce n'est pas le cas, il suffit de partir d'un test centralisé obtenu à partir de $\mathcal{A}(Comp(S))$.

La difficulté principale de la distribution provient des choix non locaux: quand le test centralisé a le choix entre plusieurs réceptions possibles sur des PCOs différents, comment effectuer ce même choix de manière répartie? La solution passe par l'utilisation d'un service de consensus: quand un testeur reçoit un message de l'implémentation, il vote pour cette action, le service de consensus élit une action et renvoie le résultat à tous les testeurs. On suppose une interaction synchrone entre testeurs et service de consensus, ce dernier ayant des solutions distribuées dans le cas d'absence de panne.

L'inconvénient de cette approche est qu'en partant d'une spécification globale sous forme d'IOLTS et en produisant un test, l'information de concurrence de la spécification est réduite à de l'entrelacement. Il en résulte des synchronisations inutiles dans les tests répartis. Une deuxième approche consisterait à conserver la concurrence de la spécification dans des modèles de vrai parallélisme. Il en résulte une refonte complète de la théorie du test de conformité dans ces modèles.

5 Test symbolique

Une des limitations majeures de notre travail sur la synthèse de tests est le fait que les variables soient manipulées de façon énumérative. De plus nous ne savons pas traiter de systèmes paramétrés, les paramètres devant être fixés avant la synthèse. En conséquence les tests sont complètement instanciés alors qu'ils sont souvent plus abstraits quand ils sont écrits manuellement. De plus l'énumération des valeurs possibles des variables est une des causes principales de l'explosion combinatoire donc de l'impossibilité de traiter complètement des systèmes de grande taille.

Bien sûr des travaux tentent de surmonter ces problèmes. La génération à la volée est une des solutions mais n'est que partielle. Elle permet d'éviter la construction complète du graphe d'état de la spécification mais ne permet pas de traiter de systèmes paramétrés.

Le problème de la synthèse de tests utilisant des techniques symboliques nous préoccupe déjà depuis un ou deux ans. Nous avons déjà commencé à y travailler mais beaucoup de problèmes restent encore à résoudre. Nous puisons notre inspiration dans les travaux sur le model-checking symbolique et dans notre connaissance de la synthèse de tests à la volée. Notre approche consiste à essayer de combiner les techniques de synthèse de test qui s'attachent seulement au contrôle et les techniques symboliques qui manipulent les données. Il n'est pas raisonnable en effet d'espérer résoudre les problèmes de synthèse de test en séparant les deux aspects puisque le contrôle est souvent dépendant des donnée et inversement. Il n'est pas question ici de donner des solutions définitives au problème puisque nos travaux en sont encore à leurs débuts. On peut cependant déjà identifier certains problèmes auxquels nous nous sommes confrontés et citer les outils théoriques et logiciels dont on dispose pour y faire face.

Modèle: Tout d'abord il est nécéssaire de définir un modèle adpaté à la manipulation symbolique et à la description des objets intervenant dans le test. Le modèle, appelons le IOSM pour InputOutput Symbolic Machine doit traiter des variables de type quelconque, des paramètres (constantes non définies), des variables de communication. Comme dans le cas des IOLTS, nous avons trois types d'actions: les émissions, les réceptions et les actions internes. Chaque action est muni d'une garde et d'un ensemble d'affectations. La garde porte sur la valeur courante des variables, et dans le cas d'une réception, sur la valeur des variables de communication. Chaque action peut avoir un effet sur la valeur des variables à l'état suivant par un ensemble d'affectations. Une IOSM a un ensemble fini d'états de contrôle. Les IOSMs de base peuvent être composés par un opérateur parallèle.

La sémantique d'une IOSM est définie en termes d'IOLTS pour une IOSMs instanciée (paramètres fixés). Les états globaux sont composés des états de contrôle de chaque IOSM et des valeurs des variables.

Les spécifications, implémentations, objectifs de test et cas de test seront exprimés par des IOSM avec quelques petites variations. En particulier les objectifs sont des automates étendus servant à sélectionner des comportements de la spécification.

Compatibilité avec les IOLTS Puisque la sémantique des IOSM s'exprime en termes d'IOLTS, nous utiliserons les mêmes notions de traces, relations de conformité, correction des tests que dans le cas des IOLTS. Bien sûr il faut quand même effectuer la synthèse sur les modèles d'IOSM. Mais il faut que cette synthèse soit compatible avec le modèle d'IOLTS. En particulier, les tests doivent être corrects pour la même notion de correction. Ceci signifie qu'un IOSM est un cas de test correct pour une spécification et une relation de conformité si pour toute instanciation des paramètres (du test compatible avec ceux de la spécification) le test instancié est correct vis à vis de la spécification instanciée et de la relation de conformité.

Synthèse Comme pour les IOLTS, la synthèse des tests fait apparaître plusieurs opérations de base.

Tout d'abord il faut calculer un IOSM produit synchrone entre un IOSM décrivant une spécification et un IOSM définissant un objectif de test. Cette opération syntaxique ne pose a priori pas de problème important.

Le produit synchrone muni de ses états accepteurs et de rejet sert à sélectionner la partie de l'IOSM menant aux états accepteurs. Ceci va déteminer une sous-IOSM duquel il faudra extraire un cas de test. Ceci peut s'effectuer sur la structure de contrôle de l'IOSM en utilisant un algorithme similaire à celui de TGV. Le calcul étant effectué sur la structure de contrôle seulement, il est possible que certains chemins dans l'IOSM résultant ne soient pas exécutables pour certaines valeurs des paramètres, voire pour toute valeur de ceux-ci. Du coup certains états de contrôle peuvent être en réalité inaccessibles. Une analyse de l'évolution des données est nécéssaire pour répondre à cette interrogation. Il est malheureusement indécidable en général de répondre à cette question. On peut envisager plusieurs solutions:

- restreindre les types de variables afin de rendre le problème décidable donc automatisable. Par exemple on peut se limiter à l'arithmétique de Presburger
- une autre possibilité est de considérer a priori tout type de variables et essayer avec l'aide de prouveurs de prouver l'accessibilité ou la non accessibilité. Il s'agit alors d'une approche assistée dont certaines parties peuvent être automatiques.

– on peut également utiliser l'abstraction. Celle-ci peut être fournie par l'utilisateur mais il est nécéssaire de prouver sa correction. Dans certains cas l'abstraction peut être calculée automatiquement [2].

Bref, il faut se préparer à utiliser toutes les techniques disponibles de type analyse statique, propagation et résolution de contraintes, abstraction, preuve.

Reste encore à dériver un test depuis cette IOSM. Divers problèmes sont encore à résoudre. D'abord les tests doivent être déterministes. Ils n'est pas exclu qu'ils aient des actions internes mais le choix de celles-ci de même que le choix entre actions observables doit être déterministe. Ce qui signifie que dans tout état de contrôle les gardes des transitions doivent être exclusives. Répondre à cette question est aussi indécidable en général. Il nécéssite a priori de connaître les valeurs possibles de toutes les variables dans tout état de contrôle. Ceci résulte donc encore du même type d'analyse que précédemment qui peuvent être combinées. Si l'IOSM n'est pas déterministe, il faut le déterminiser ce qui en général n'est pas toujours possible.

Notons que l'analyse peut être facilitée si on procède à un dépliage de la structure de l'IOSM. Certains calculs de point fixes peuvent en être facilités. Cependant, il n'est pas possible en général de déterminer le dépliage le plus pertinent. On peut proposer des heuristiques comme par exemple basées sur un algorithme de type Bourdoncle [3]. Le dépliage peut être vu comme une récriture de graphe par les règles d'une grammaire de graphes.

Il faut noter aussi que les tests sont souvent plus abstraits que les spécifications. Ceci signifie que les variables manipulées par les tests doivent être abstraites et simplifiées. Ceci résulte encore du même type de techniques que précédemment.

Références

[1] S. Abramsky. Observational Equivalence as a Testing Equivalence. *Theoretical Computer Science*, 53(3), 1987.

[2] S. Bensalem, Y. Lakhnech, and S. Owre. InVeSt: A tool for the verification of invariants. *Lecture Notes in Computer Science*, 1427:505–??, 1998.

[3] F. Bourdoncle. Efficient Chaotic Iteration Strategies with Widenings. In *International Conference on Formal Methods in Programming and their Applications*, pages 128–141. Springer-Verlag, LNCS 735, 1993.

[4] M. Bozga, J.-C. Fernandez, L. Ghirvu, C. Jard, T. Jéron, A. Kerbrat, P. Morel, and L. Mounier. Verification and test generation for the SSCOP protocol. *Journal of Science of Computer Programming, special issue on Formal Methods in Industry*, 36(1):27–52, January 2000.

[5] E. Brinskma. A Theory for the Derivation of Tests. In *Protocol Secification, Testing and Verification VIII*, pages 63–74. North-Holland, 1988.

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

[7] R. De Nicola and M. Henessy. Testing Equivalences for Processes. *Theoretical Computer Science*, 34:83–133, 1984.

[8] R.G. De Vries and J. Tretmans. On-the-Fly Conformance Testing Using SPIN. In G. Holzmann, E. Najm, and A. Serhrouchni, editors, *Fourth Workshop on Automata Theoretic Verification with the* SPIN *Model Checker*, ENST 98 S 002, pages 115–128, Paris, France, November 2 1998.

[9] J.-C. Fernandez, H. Garavel, L. Mounier, A. Rasse, C. Rodriguez, and J. Sifakis. A Tool Box for the Verification of Lotos Programs. In *14th International Conference on Software Engineering*, Melbourne, Australia, May 1992.

[10] J.-C. Fernandez, C. Jard, T. Jéron, and G. Viho. Using on-the-fly verification techniques for the generation of test suites. In A. Alur and T. Henzinger, editors, *Conference on Computer-Aided Verification (CAV '96), New Brunswick, New Jersey, USA*, volume 1102 of *LNCS*. Springer-Verlag, July 1996. Egalement disponible en rapport de recherche Irisa n^o 1036.

[11] A. Gill. *Introduction to the Theory of Finite State Machine*. New-York, McGraw-Hill, 1962.

[12] M. Gondran and M. Minoux. *Graphes et algorithmes*. Eyrolles, 1979. Collection de la Direction des Etudes et Recherches d'Electricité de France.

[13] R. Groz, T. Jéron, and A. Kerbrat. Automated test generation from SDL specifications. In Rachida Dssouli, Gregor von Bochmann, and Yair Lahav, editors, *SDL'99 The Next Millenium, 9th SDL Forum, Montréal, Québec*, pages 135–152. Elsevier, June 1999.

[14] ITU-T SG 10/Q.8 ISO/IEC JTC1/SC21 WG7. Information Retrieval, Transfer and Management for OSI; Framework: Formal Methods in Conformance Testing. Committee Draft CD 13245-1, ITU-T proposed recommendation Z 500. ISO - ITU-T, 1996.

[15] C. Jard, T. Jéron, and P. Morel. Verification of test suites. In *TestCom 2000, IFIP TC 6 / WG 6.1, The IFIP 13th International Conference on Testing of Communicating Systems, Ottawa, Ontario, Canada*. Kluwer Academic Publishers, August 2000.

[16] C. Jard, T. Jéron, L. Tanguy, and C. Viho. Remote testing can be as powerful as local testing. In Jianping Wu, Samuel Chanson, and Qiang Gao, editors, *Formal methods for protocol engineering and distributed systems, FORTE XII/ PSTV XIX' 99, Beijing, China*, pages 25–40. Kluwer Academic Publishers, October 1999.

[17] T. Jéron and P. Morel. Abstraction, τ-réduction et déterminisation à la volée: application à la génération de test. In *CFIP'97, Congrès Francophone sur l'Ingéniérie des Protocoles, Liège, Belgique*. Hermes, September 1997.

[18] T. Jéron and P. Morel. Test generation derived from model-checking. In Nicolas Halbwachs and Doron Peled, editors, *CAV'99, Trento, Italy*, volume 1633 of *LNCS*, pages 108–122. Springer-Verlag, July 1999.

[19] H. Kahlouche, C. Viho, and M. Zendri. An Industrial Experiment in Automatic Generation of Executable Test Suites for a Cache Coherency Protocol. In A. Petrenko and N. Yevtushenko, editors, *IFIP TC6 11th International Workshop on Testing of Communicating Systems*. Chapman & Hall, September 1998.

[20] H. Kahlouche, C. Viho, and M. Zendri. Hardware Testing using a Communication Protocol Conformance Testing Tool. In W. Rance Cleaveland, editor, *Tools and*

Algorithms for the Construction and Analysis of Systems (TACAS'99), volume 1579 of *Lecture Notes in Computer Science*, pages 315–329. Springer Verlag, March 1999.

[21] Lee and Yannakakis. Testing Finite State Machines: State Identification and Verification. *IEEE Trans Computer*, 43(3):306–320, 1994.

[22] D. Lee and M. Yannakakis. Principles and Methods of Testing Finite State Machines - A Survey. *Proceedings of the IEEE*, 84(8):1090–1123, August 1996.

[23] 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.

[24] S. Naito and M. Tsunoyama. Fault Detection for Sequential Machines by Transitions Tours. In *Proc. IEEE Fault Tolerant Comput. Symp., IEEE Computer Soc. Press*, pages 238–243, 1981.

[25] A. Petrenko, S. Boroday, and R. Groz. Conforming configurations in efsm. In *Forte-PSTV 99*. Kluwer, oct 1999.

[26] M. Phalippou. *Relations d'implantations et Hypothèses de test sur les automates à entrées et sorties*. PhD thesis, Université de Bordeaux, 1994.

[27] R. Tarjan. Depth-first search and linear graph algorithms. *SIAM Journal of Computing*, 1:146–160, 1972.

[28] J. Tretmans. Testing Labelled Transition Systems with Inputs and Outputs. In 8^{th} *International Workshop on Protocols Test Systems*, Evry - France, September 1995.

[29] J. Tretmans. Test Generation with Inputs, Outputs and Quiescence. In T. Margaria and B. Steffen, editors, *Second International Workshop on Tools and Algorithms for the Construction and Analysis of Systems, TACAS'96, Passau, Germany*. Springer Verlag, LNCS 1055, March 1996.

[30] J. Tretmans. Test generation with inputs, outputs and repetitive quiescence. *Software - Concepts and Tools*, 17, 1996.