Université Joseph Fourier
U.F.R. Informatique &
Mathématiques Appliquées

Institut National Polytechnique
de Grenoble ENSIMAG

I. M. A. G.

ECOLE DOCTORALE
MATHEMATIQUES ET INFORMATIQUE

DEA D'INFORMATIQUE :
SYSTEMES ET COMMUNICATION

Projet présenté par :

Cyril PACHON

Génération automatique de tests symboliques

Effectué au laboratoire : LSR

Date : 21 juin 2001

Résumé

La complexité croissante des protocoles exige implicitement l'évolution et le développement des méthodes et des outils adéquats pour leur validation. Afin de pouvoir garantir le bon fonctionnement d'un protocole, deux méthodes complémentaires existent:

- La vérification formelle consiste à comparer la description d'un système (une spécification) avec un critère donné (une propriété) pour des services attendus.
- Le test de conformité consiste à démontrer la conformité du protocole (implantation) à sa description formelle (la spécification) de référence. Dans la pratique, la conformité est testée à l'aide d'une suite de tests, construit à partir de la spécification et des propriétés que nous voulons tester. L'activité de test est composé, d'une génération d'une suite de tests à partir du modèle de la spécification et des propriétés à tester, puis d'une exécution de tests sur l'implantation à l'aide d'un testeur.

Objectif de ce travail :

Pour rendre automatisable le processus de la génération de tests, nous nous intéressons à une approche basée sur les modèles. Nous voyons bien que cette approche a ses limites : spécification complexe, domaine de variable infini.

Nous proposons d'effectuer de la génération de tests symboliques, c'est à dire, de ne pas instancier certaines variables ou paramètres. Cette approche est au carrefour de trois domaines : vérification par les modèles (model-checking), démonstration automatique (theorem-proving) et analyse statique. Schématiquement, on utilise des techniques de vérification par les modèles pour les algorithmes de génération, les techniques d'analyse statique pour la simplification et l'abstraction. L'objectif de ce projet est de proposer une méthodologie efficace pour la génération de tests symboliques.

Mots clés

Systèmes de transitions étiquetées, automates communicants, spécification, objectif de test, test de conformité, produit synchrone, projection.

Table des matières

Table des figures

Introduction

Les systèmes distribués sont constitués d'un ensemble de composants interagissant entre eux et avec un environnement. L'utilisation de tels systèmes fait partie maintenant de la vie quotidienne (réseaux pour les télécommunications, pour la distribution de billet de banque, …), mais sont aussi au cœur d'applications critiques, comme par exemple, les transports (train, aéronautique, …) ou, font partie intégrante des processus de contrôle industriels (centrales atomiques, usines chimiques, …).

C'est pour tout cela et également parce que la complexité des programmes est grandissante que le développement d'outils d'aides à la conception et à la validation est nécessaire et indispensable au bon fonctionnement des exécutions. Un dysfonctionnement de ces programmes, même temporaire, peut avoir de lourdes conséquences économiques, voire attenter à la sécurité des individus. Leur validation avant leur mise en place est essentielle et doit garantir la correction de l'ensemble des comportements possibles du programme.

Validation

La complexité des systèmes distribués exige implicitement l'évolution et le développement des méthodes et des outils adéquats pour leur validation. Afin de pouvoir garantir le bon fonctionnement d'un système, deux méthodes complémentaires existent :

– *La vérification formelle*

La vérification formelle consiste à comparer la description d'un système (une spécification) avec un critère donné (une propriété) pour des services attendus. Ces comparaisons s'effectuent selon une relation de satisfaction définie elle aussi formellement.

Une approche possible de vérification formelle est l'utilisation de démonstrateurs automatiques ou de systèmes de réécriture. L'utilisation de ces systèmes est en général difficile et au mieux semi-automatisable.

Une autre approche dans ce travail est basée sur la comparaison de modèles (*"model checking"*). Le modèle couramment considéré jusqu'à présent, pour la vérification formelle est un *système de transitions étiquetées.* Le système est un automate dont les transitions entre les états sont étiquetées par les actions du système.

Le processus de vérification formelle se décompose alors en deux phases :

- Une traduction de la description d'un programme vers le modèle,
- Une vérification de ce modèle par un critère de même modèle.

L'intérêt pratique de cette approche est qu'elle est complètement automatisable. Elle est par contre limitée par la taille des modèles générés.

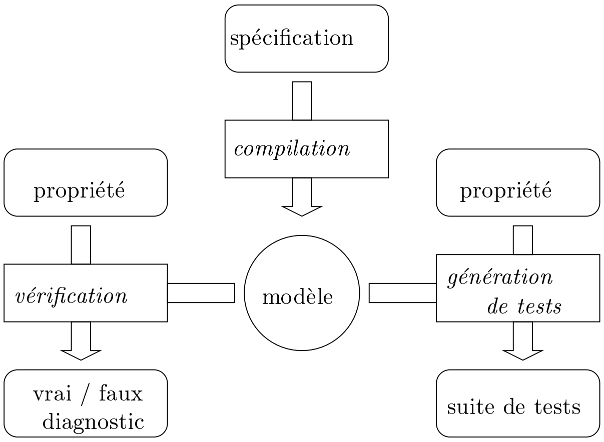

FIG. I.1 – *Validation basée sur les modèles*

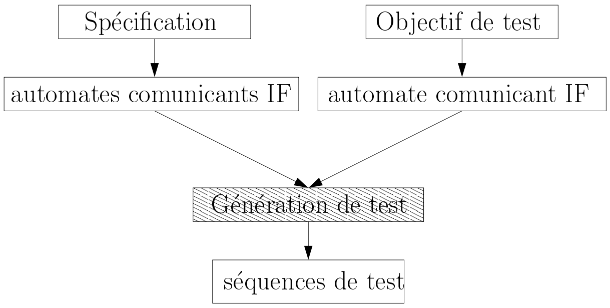

FIG. I.2 – *Représentation de la vérification formelle*

– *Le test de conformité* [Org93]

Il consiste à démontrer la conformité d'une réalisation concrète du système (*l'implantation*) à sa description formelle (*la spécification*) de référence. Là aussi, la relation de conformité peut être définie formellement.

Dans la pratique, la conformité est testée à l'aide d'une *suite de tests*, construite à partir de la spécification et des propriétés que l'on veut tester. Pour les mêmes raisons, nous considérons l'approche basée sur les modèles. Le test est décomposé en deux phases, complètement automatisables : *génération* d'une suite de tests à partir du modèle de la spécification et des propriétés à tester, puis *exécution* de tests sur l'implantation à l'aide d'un testeur. Nous nous intéressons uniquement à la phase de génération de suite de tests, comme le montre la figure I.1.

Historiquement, le test a été la première et principale méthode de validation. Il reste une activité toujours très importante, étant données les difficultés majeures de conception et vérification formelle.

L'application de ces techniques au sein du laboratoire VERIMAG a donné lieu à des outils performants autant pour la vérification que pour le test. En particulier, nous mentionnons l'outil ALDEBARAN dédié à la vérification des propriétés, ainsi que l'outil TGV, réalisé en collaboration avec l'équipe Pampa de l'IRISA, dédié à la génération de suites de tests.

Notre approche se base sur un principe similaire, mais avec l'utilisation *d'automates communicants* qui ont pour sémantique opérationnelle les systèmes de transitions étiquetées (STE). Pour introduire les automates communicants, nous allons utiliser le modèle IF. Le modèle IF est une représentation intermédiaire au langage de description de communication, développé par le laboratoire VERIMAG pour élaborer des analyses statiques sur les automates communicants.

Explosion des états

Un problème lié aux méthodes de validation basées sur les modèles est la taille de ces modèles. Dès que les programmes étudiés atteignent une certaine complexité, la taille du modèle correspondant devient rapidement prohibitive : ce problème est connu sous le nom de *problème de l'explosion des états*. Les principales causes de ce problème sont les suivantes :

- la complexité des composantes séquentielles du programme puisqu'une représentation énumérative associe un état à chaque valuation des variables du programme,
- la mise à plat des opérateurs de composition parallèle puisque la complexité du modèle global est exponentielle en fonction du nombre de composantes parallèles, en représentant le parallélisme par l'entrelacement d'actions.

L'explosion des états s'avère un problème crucial, dans la pratique, pour les outils classiques de vérification. Ces outils se trouvent de fait limités à la vérification de modèles ayant environ quelques millions d'états, ce qui est largement insuffisant pour des cas réalistes.

De nombreux travaux ont été effectués pour trouver des solutions permettant de pallier ce problème en ne représentant qu'une partie des états en mémoire, en comprimant les représentations, en décomposant le système à vérifier ou encore en le traitant partiellement. Les méthodes couramment employées sont les suivantes :

- **vérification à la volée** [FJJM92]

  Plutôt que de construire complètement le modèle, puis de vérifier sur ce modèle, la propriété est vérifiée *à la volée*, pendant la construction du modèle. Les méthodes à la volée offrent l'avantage d'avoir à chaque instant seulement une partie du modèle en mémoire, construite et gérée au fur et à mesure des besoins de vérification.

- **ordres partiels** [God96]

  Une des causes reconnues de l'explosion des états est liée à la représentation du parallélisme par *l'entrelacement* d'actions parallèles asynchrones. Une idée consiste alors à éviter de parcourir tous les chemins construits par entrelacement, pour n'en parcourir qu'un sous-ensemble caractéristique pour la propriété à vérifier. Cette technique se combine très bien avec la vérification à la volée, chaque méthode tirant parti des avantages offerts par l'autre.

- **réduction** [Wei79] [BFG00]

  Cette méthode consiste à construire le modèle *minimal* par rapport à une relation d'équivalence. Celle-ci est choisie de telle manière que le modèle réduit et le modèle du programme initial sont équivalents pour les propriétés à vérifier.

- **abstraction** [Loi94]

  L'abstraction consiste à construire un modèle *abstrait* à partir du modèle d'origine dit modèle concret, tel que chaque action du modèle concret peut être simulée par une action dans le modèle abstrait. Cette approche permet une vérification partielle de propriétés : si une propriété est vérifiée sur le modèle abstrait alors elle est valide sur le modèle concret, sinon nous ne pouvons tirer aucune conclusion.

- **composition** [JL97]

  Cette approche prend en compte la structure du programme et consiste à construire une vérification du programme global à partir de vérifications faites sur les sous-programmes. De même, nous pouvons utiliser cette technique pour la réduction ou l'abstraction du modèle global, en construisant ce modèle à partir des modèles minimaux construits pour chacune des composantes.

Objectif de ce travail

Le test de conformité des protocoles est un test de type boîte noire : de l'implantation on ne connaît le comportement que via ses interactions avec l'environnement. La notion de conformité est donc basée sur les relations d'entrées / sorties de l'implantation. Le principe fondant un outil comme TGV est le suivant : partant d'une spécification et d'une propriété (objectif de test), l'algorithme de génération explore les comportements décrits par la spécification, compatibles avec la propriété, et génère un test qui stimulera le système sous test par des événements émis depuis l'environnement. Des techniques utilisées dans les outils sont dites énumératives : dans l'algorithme de génération, on considère chaque point de contrôle et chaque valeur de variable. Nous voyons bien que cette approche a ses limites : spécification complexe, domaine de variable infini. Nous proposons donc une nouvelle approche, la génération de tests symboliques, c'est à dire, de ne pas instancier certaines variables ou certains paramètres. Cette approche est au carrefour de trois domaines : vérification par les modèles (model-checking), démonstration automatique (theorem-proving) et analyse statique. Schématiquement, on utilise des techniques de vérification par les modèles pour les algorithmes de génération, les techniques d'analyse statique pour la simplification et l'abstraction, les techniques de démonstration automatique pour la génération et la propagation de contraintes. L'objectif de ce projet est de proposer une méthodologie efficace pour la génération de tests symboliques. Pour cela nous allons travailler sur la réduction des tailles des STE. Pour effectuer ces réductions, nous nous baserons sur une forme intermédiaire qui sera plus expressive pour la représentation des STE.

- **modèles de systèmes distribués**

Un système distribué peut-être vu comme un ensemble de programme qui interagissent. La description de modèle comporte *un formalisme de description* très précis. Il est important pour intégrer les aspects essentiels de la programmation de système d'une part, et d'être mieux adapté pour la vérification symbolique d'autre part. Le formalisme que nous avons choisi dans cette étude sont les automates communicants AC, pour lesquels nous avons défini des opérateurs de composition et d'abstraction, en vu de réduire leur taille pour rendre les tests plus efficaces.

– **techniques de validation des automates**

La validation se décompose en trois étapes :

- **abstraction**

L'abstraction se situe au niveau des variables et des files d'attente des automates. Les variables des actions ne seront pas prises en compte au moment des compositions. Les files d'attente seront considérées selon le critère vide / non vide.

- **composition**

Nous nous sommes fixés comme objectif l'étude de quelques techniques d'analyse statique, vues aussi comme une solution potentielles à l'explosion des états. Nous allons déterminer de façon précise un algorithme permettant de composer des automates communicants du modèle IF. Cette description comporte : une expression de composition *le produit synchrone* qui a pour but de ne garder que les séquences pertinentes d'un ensemble d'automate par un critère d'entrée et d'une *projection* qui exploite le résultat de la composition pour redonner une spécification ou chaque automate est réduit.

– **intégration dans un environnement de validation**

Actuellement les tests sont faits à partir des STE. Ces techniques sont déjà dans des outils comme TGV [FJJV96] ou TestComposer de *Object*GEODE. Or, une représentation intermédiaire a été proposée pour représenter les systèmes de manière plus expressifs. Nous allons utiliser ces modèles pour réduire les tailles des STE dans le but de faire des tests. Sur la figure I.2, nous représentons par ‖AC l'ensemble des automates communicants initiaux, par ‖AC* l'ensemble des automates réduits par le critère d'entrée OT , l'Objectif de Test. Nous volons garantir que les STE obtenus à partir des ‖AC* préservent ($\sim$) les comportements initiaux, de telle sorte que les tests générés seront identiques.

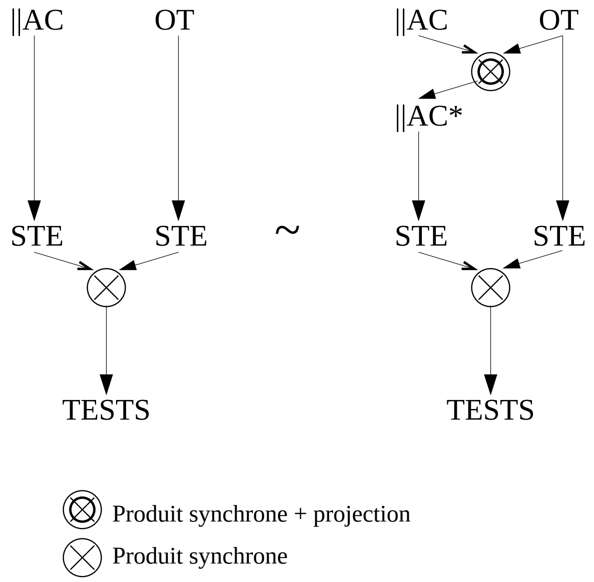

FIG. I.3 – *Nos objectifs*

– **mise en œuvre**

La mise en œuvre est l'élaboration du programme permettant la composition sur des automates communicants selon un critère de départ. Puis, après cette exécution, nous établissons un traitement de recomposition des automates ||AC* par projection, afin d'obtenir une spécification qui simule les STE initiaux.

– **dans cette contribution**

Notre contribution est l'extension d'une technique de réduction (chapitre 4) basée sur les **Feeds**, à une technique prenant compte l'ordre des actions de l'objectif de test. Cette nouvelle technique est basée sur le produit synchrone des automates de contrôle sous-jacents aux automates communicants. Ensuite, nous avons mis en œuvre cette technique dans l'environnement IF, et expérimenté sur le protocole SIP.

Organisation du document

L'organisation de ce document donne les différents thèmes qui nous ont menés à une réalisation, ainsi qu'une expérimentation sur un protocole de communication existant, pour concrétiser notre approche.

Le chapitre 1 définit les modèles de base, les systèmes de transitions étiquetées qui ont faits l'objet de représentation et d'outils existants.

Le chapitre 2 définit les automates communicants. Puis introduit le nouveau modèle intermédiaire IF pour les langages de communication.

Le chapitre 3 présente le test de conformité adapté à la perspective d'obtenir une conformité d'une spécification et d'un critère d'entrée, réalisé avec les nouveaux modèles de représentation.

Le chapitre 4 décrit des analyses statiques. Adaptées aux nouveaux modèles ces représentations sont le support pour définir les conceptions de notre outil. Il s'agira de décrire la méthode construction " le produit synchrone" sur des automates communicants pour obtenir un $\|AC'$ et de redonner par "projection" la spécification $\|AC^*$, ainsi obtenu avec le critère d'entrée.

Le chapitre 5 est la mise en œuvre des présentations précédentes.

Le chapitre 6 est consacré aux outils que nous avons réalisés. Nous présenterons à partir du protocole de communication SIP étudié, quelques exemples pour la génération de tests par ce système.

Chapitre 1

Systèmes de transitions étiquetées

Dans ce chapitre nous décrivons les systèmes de transitions étiquetées, qui sont les modèles de base que nous considérons, ainsi que certaines opérations sur ces modèles.

1.1 STE, Présentation

Dans cette section, nous introduisons le modèle des systèmes de transitions étiquetées (STE), ainsi que certaines notations. Les STE sont utilisés pour décrire la sémantique de certains langages de spécification comme SDL[IT92] [CCI88], ou CSP[BHR84], ainsi que les graphes intervenant dans la génération automatique de tests qui seront développés dans la partie 3.3.

La sémantique opérationnelle d'un programme est un STE décrivant l'ensemble de ses comportements possibles. Un STE est donné, par un ensemble d'états et une relation de transition. Un état est caractérisé par un point de contrôle du programme et les valeurs des différentes variables en ce point de contrôle. La relation de transition représente l'évolution du programme.

Définition 1.1.1 (STE : syntaxe) *Un système de transitions étiquetées est un tuple, quadruplet* $(Q, \Sigma, \xrightarrow{a}, q^{init})$ *où,*

- Q *est un ensemble (ensemble d'états),*
- q^{init} *est un état de* Q *(état initial),*
- Σ *est l'ensemble des étiquettes,*

- $a \in \Sigma$,
- $\longrightarrow$ *est une relation ternaire incluse dans* $Q \times \Sigma \times Q$.
- $(q,a,q') \in \longrightarrow$ *est noté* $q \xrightarrow{a} q'$

Nous définissons qu'un état q est prédécesseur de l'état $q^{'}$ par la transition "a" avec la fonction Pre:

- $Pre_a \; : \; 2^Q \; \longrightarrow \; 2^Q$
- $Pre_a(q) \; = \; \{q^{'} \; | \; \exists \; a \; q \xrightarrow{a} q^{'} \; et \; q^{'} \; \in \; Q\}$
- $Pre^*(q) \; = \; \bigcup_{a \; \in \; \Sigma} Pre_a(q)$

Un STE peut-être représenté graphiquement par:

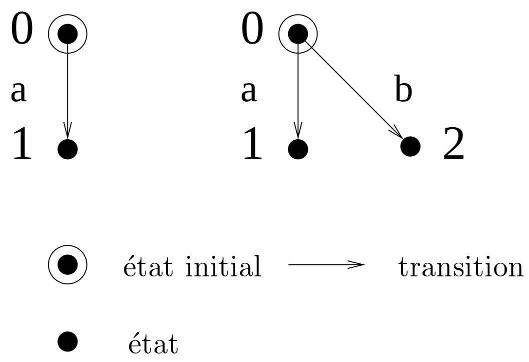

FIG. 1.1 – *Représentation d'un système de transitions étiquetées STE*

Dans un STE un état q est atteignable s'il existe une succession d'états q^i partant de l'état initial q^{init} menant à cet état q.

Définition 1.1.2 (STE : état atteignable) *Soit un STE,* $s = (Q_s, \Sigma_s, \xrightarrow{a}_s, q_s^{init})$. *Un chemin de l'état q à q' est une suite* $(q^0, q^1, \ldots, q^n)$ *telle que* $q = q^0$, $q' = q^n$, *et que* $\forall \; i < n$, $\exists \; a \in \; \Sigma_s \; q^i \xrightarrow{a}_s \; q^{i+1}$.

Définition 1.1.3 (STE : relation de simulation et de bisimulation) *Soit* $S = (Q, \Sigma, \xrightarrow{a}, q^{init})$ *et* $\lambda \subseteq Q \times Q$ *un préordre (relation reflexive et transitive) sur les états.*

- λ *est une relation de simulation si et seulement si :*

$$\forall\ (q_1, q_2) \in\ \lambda, \forall\ a\ \in\ \Sigma$$
$$\forall\ q_1^{'}\ \in\ Q\ \textit{tel que}\ q_1 \xrightarrow{a} q_1^{'},\ \exists\ q_2^{'}\ \textit{tel que}\ q_2 \xrightarrow{a} q_2^{'}\ \textit{et}\ (q_1^{'}, q_2^{'})\ \in\ \lambda$$

– λ *est une relation de bisimulation si et seulement si :*

$$\forall\ (q_1, q_2) \in\ \lambda, \forall\ a\ \in\ \Sigma$$
$$\forall\ q_1^{'}\ \in\ Q\ \textit{tel que}\ q_1 \xrightarrow{a} q_1^{'},\ \exists\ q_2^{'}\ \textit{tel que}\ q_2 \xrightarrow{a} q_2^{'}\ \textit{et}\ (q_1^{'}, q_2^{'})\ \in\ \lambda$$
$$\forall\ q_2^{'}\ \in\ Q\ \textit{tel que}\ q_2 \xrightarrow{a} q_2^{'},\ \exists\ q_1^{'}\ \textit{tel que}\ q_1 \xrightarrow{a} q_1^{'}\ \textit{et}\ (q_1^{'}, q_2^{'})\ \in\ \lambda$$

Une simulation λ est un préordre, donc $\lambda \bigcap \lambda^{-1}$ est une équivalence. Toute bisimulation est une équivalence.

1.2 STE, produit synchrone

L'introduction de l'opération du produit synchrone $\otimes$, nous servira par la suite pour sélectionner les chemins d'atteignabilités des différents STE. Ce produit sera une des bases de l'étude de la section 4 (sur des automates communicants de conception fortement similaire aux STE).

Le produit synchrone est effectué sur des transitions (basé sur une comparaison en utilisant l'union des relations des transitions). Comme nous effectuons le produit entre STE, cela nous précise que les STE sont comparables l'un avec l'autre par les étiquettes des transitions. On pourra obtenir un chemin de transitions qui nous amènera d'un couple d'états initial à un couple d'états final désiré.

Définition 1.2.1 (Produit synchrone : $\otimes$) *Soient deux STE S1 = (Q_1, Σ_1, $\xrightarrow{a}_1$, q_1^{init}) et S2 = (Q_2, Σ_2, $\xrightarrow{a}_2$, q_2^{init}).*

Le produit synchrone s'obtient par composition des deux STE avec l'opérateur $\otimes$ et sera noté S_p = S1 $\otimes$ S2 c'est un tuple (Q_p, Σ_p, $\xrightarrow{a}_p$, q_p^{init}) tel que :

- $Q_p \subseteq Q_1 \times Q_2$ *est l'ensemble des couples formés par les états de S1 et de S2,*
- $\Sigma_p = \Sigma_1 \cap \Sigma_2$ *est l'ensemble des étiquettes représentant les actions possibles dans* S_p*,*
- $a\ \in\ \Sigma_p$ *est une étiquette de l'ensemble* Σ_p*,*
- $q_p^{init} = (q_1^{init}, q_2^{init})$ *est le couple initial de nouveau STE formé des états initiaux de S1 et de S2.*

Soit $\xrightarrow{a}_p$ est le plus petit ensemble vérifiant :

$$\frac{(q_1, q_2) \in Q_p,\ q_2 \underset{2}{\xrightarrow{a}} q_2^{'},\ q_1 \underset{1}{\xrightarrow{a}} q_1^{'}}{(q_1^{'}, q_2^{'}) \in Q_p,\ (q_1, q_2) \underset{p}{\xrightarrow{a}} (q_1^{'}, q_2^{'}),\ a \in \Sigma_p}$$

Exemple 1.1 *Exemple d'un produit synchrone entre les deux systèmes de transitions étiquetées S1 et S2.*

La figure 1.2 montre le STE obtenu en appliquant les règles du produit synchrone sur les deux STE : S1 = ((0,1), (a), 0) et S2 = ((0,1,2), (a,b), 0). L'opérateur de composition du produit synchrone étant toujours $\otimes$ on a :

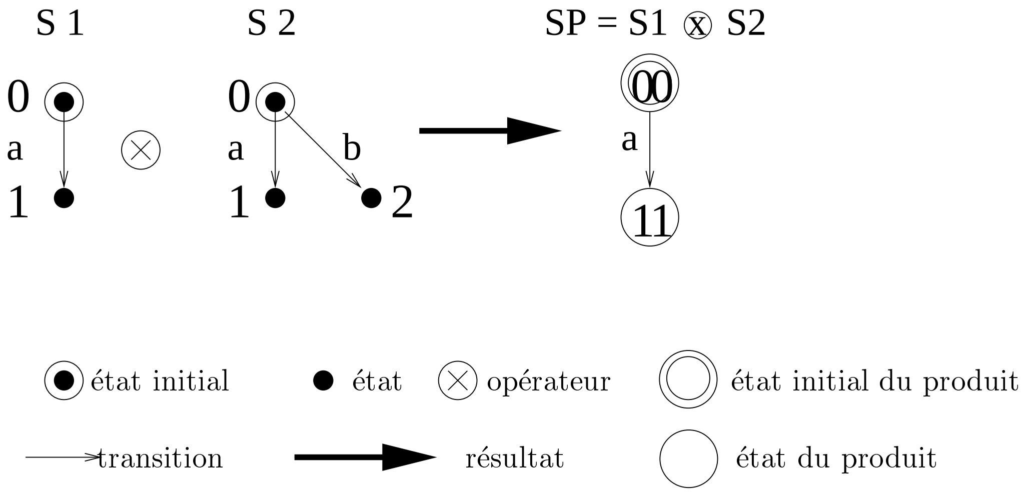

FIG. 1.2 – *Produit synchrone P de deux STE, 1, 2*

Remarque 1.1 (notation des états du produit synchrone) *Les états du produit synchrone sont représentés par un ensemble d'état des STE initiaux. Dans l'exemple ci dessus on a (00) soit 0 du premier STE puis 0 du deuxième STE. De façon générale les états du produit synchrone sont formés dans l'"ordre" des STE.*

Nous nous apercevons que le produit synchrone est obtenu en utilisant la transition $\xrightarrow{a}$ commune aux deux STE S1 et S2 ce qui forme un état initial (00) et un état cible (11).

(00) et (11) sont des états du résultat de la composition faite avec les états de S1 et S2. L'action b n'étant pas commune au deux STE se voit écartée de la composition finale.

1.3 Conclusion

Nous venons de faire une présentation générale des systèmes de transitions étiquetées STE afin d'en comprendre leur fonctionnement. Nous rappelons qu'un STE est utilisé pour représenter l'ensemble des exécutions possibles d'un programme.

Puis, nous avons introduit le produit synchrone $\otimes$. Ce produit réalise une composition entre des STE. Le résultat de cette composition est un STE qui représente l'ensemble des comportements communs aux deux STE. Or les STE sont souvent de grande taille, ce qui est un problème pour garder les compositions en mémoire. L'explosion du nombre d'états est une difficulté que nous devons gérer.

Pour limiter ce problème de taille, nous allons introduire au chapitre 2 les automates communicants AC. La sémantique d'un automate communicant AC est un STE qui représente l'ensemble de ses comportements possibles. Les automates communicants sont plus expressifs en ce qui concerne les données.

Chapitre 2

Automates communicants

Ce chapitre se compose de la façon suivante. La première partie est une présentation générale des automates communicants AC. Ensuite, nous allons introduire le modèle particulier IF, qui est une représentation intermédiaire à base d'automates communicants, (étant donné que la partie réalisation (chapitre 6) du travail sera avec ce modèle). Ce modèle est conçu pour la description et la validation formelle de systèmes asynchrones. De plus, nous regarderons par quelques définitions les différences entre les automates communicants existant avec ceux réalisés dans le modèle IF.

Sur les modèles ainsi obtenus, nous effectuerons des analyses statiques (chapitre 4) telles que l'analyse des variables actives [BFG99a] ou le *slicing* [Wei79] [Tip95] [Gou97] [BFG00], tout cela issue du domaine de l'optimisation de code. Toutes les améliorations sur les automates communicants AC seront un gain de taille et d'efficacité pour la génération de tests automatiques basée sur le produit synchrone.

2.1 Les automates communicants

2.1.1 Syntaxe

Nous utiliserons pour l'expression d'un automate communicant :

Définition 2.1.1 (Automates communicants :) *Une spécification d'un système d'automates communicants est un tuple (S, C, P) où,*

- *S est un ensemble de signaux,*
- *C est un ensemble de files d'attente (interne ou externe), $C = C^{int} \cup C^{ext}$,*
- *P est l'ensemble des processus d'automates communicants.*

Un processus $p \in P$ est un tuple $(X_p, Q_p, T_p, \Sigma_p, q_p^0)$ ou

- X_p *est un ensemble de variables locales,*
- Q_p *est un ensemble d'états,*
- Σ_p *est un ensemble d'actions qui peuvent être effectuées par p,*
- $T_p \subseteq Q_p \times \Sigma_p \times Q_p$ *est un ensemble de transitions.*

Une action peut-être

- *une condition et une affectation $[b]\ x := e$,*
- *une condition et une entrée $[b]\ c?s(x)$,*
- *une condition et une sortie $[b]\ c!s(e)$.*

En prenant :

- *b et e sont des expressions,*
- *$x \in X_p$ est une variable,*
- *$c \in C_p$ est une file,*
- *$s \in S_p$ est un signal.*

2.1.2 Sémantique

La sémantique des automates communicants est donnée en terme de systèmes de transitions étiquetées. Nous considérons le domaine D qui contient les valeurs des variables et les paramètres signaux. Nous supposons que nous disposons dans D des booléens ($\mathbf{t}$,$\mathbf{f}$) ainsi que d'une valeur spéciale non définie $\bot$. Nous notons ε la séquence vide.

Nous définissons :

- $\rho : \bigcup_{p \subseteq P} X_p \longrightarrow D$
  qui associe à chaque variable une valeur v,

– $\delta : C^{init} \longrightarrow (S \times D)^*$

qui associe à chaque queue interne c une séquence $(s_1, v_1), .., (s_k, v_k)$ de messages (s, v) étant noté par $s(v)$ où s est un signal et v est une valeur.

Définition 2.1.2 (Sémantique des automates communicants :) *. La sémantique d'une spécification est donnée par un STE, $s = (Q_s, T_s, q_s^0)$. Les états du STE sont des triplets (ρ, δ, θ) où $\theta = \langle q_1, .., q_n \rangle \subseteq \times_{p \in P} Q_p$,*

– *ρ est un contexte pour les variables,*

– *δ est un contexte pour les files d'attente.*

Les transitions sont construites de la façon suivante :

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

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

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

$$\frac{q_p \xrightarrow{[b]c?s(x)} q'_p \quad \rho(b) = \mathbf{t},\ c \in C^{ext}, v \in D}{(\rho, \delta, \theta) \xrightarrow{c?s(v)} (\rho[v/x], \delta, \theta')}$$

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

$\theta^{'}$ est obtenu à partir de θ en remplaçant q_p par $q_p^{'}$.

Exemple 2.1 *Regardons figure 2.1 la représentation d'un automate communicant S = ((0,1,2,3,4,5),([b] f2?s2(x),[b] f2!s2(x),[b] f1!s1(x),[b] f1?s1(x) ,[b] x:=5,[b] x:=y),0)*

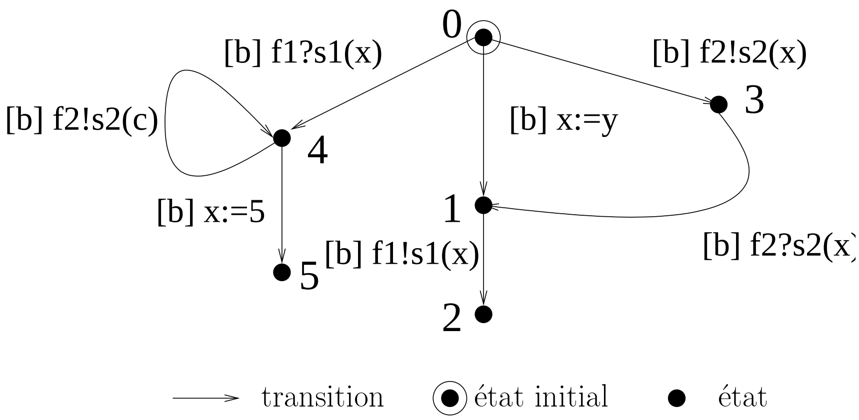

FIG. 2.1 – *Représentation d'automate communicant*

Pour établir des automates communicants à partir d'une spécification , un modèle a été créé. A la section suivante, nous présentons le langage IF qui est une représentation intermédiaire entre un langage de spécification et les STE.

2.2 Le langage IF

IF ([Boz99], [BGMS98]) est un langage pour la description de systèmes, développé au sein du laboratoire VERIMAG. IF est suffisamment expressif pour rester en amont du problème d'explosion d'états et pour simuler simplement des concepts existants dans les formalismes de spécification. Le modèle IF [BFG+99b] est basé sur la notion d'automates temporisés communicants. Plus généralement ce format [BFLG+99] peut-être vu comme un modèle intermédiaire pour les langages de spécification comme SDL [CCI88] ou UML [FEL97]. En fait, les programmes IF gardent de manière explicite un certain nombre d'informations comme par exemple le parallélisme, le temps ou encore les données manipulées. IF a une sémantique opérationnelle complètement définie en terme de STE. Concrètement, la sémantique consiste, par un ensemble des règles à construire, à partir d'un programme IF, l'ensemble de tous les comportements d'une exécution, sous la forme d'un STE. C'est un aspect très important pour la validation automatique basée sur des modèles, étant donné que toutes les techniques existantes de validation s'appliquent à ce niveau d'abstraction. En plus la modélisation explicite des actions va nous permettre de travailler et de distinguer plus finement les transitions à garder pour constituer les séquences de tests.

Un programme IF se compose, d'un système qui est un ensemble d'automates communicants. Ce système est représenté par un tuple (Définition_globale, Processus). Où, Définition_globale, donne des informations communes aux processus du système, et Processus contient des automates communicants. Nous allons introduire de façon formelle le contenu de ce tuple.

2.2.1 Définitions globales

La partie "Définition_globale" est un tuple (Type_donnée, Signaux, variables, files d'attente) représentant les définitions globales des processus du programme IF. Ces définitions servent pour les interactions des automates communicants. Dans cette partie le modèle IF est commun a tous modèles décrivant des automates communicants, et nous avons :

- **Type donnée** est un ensemble de types de donnée = (entiers, booléens, tableaux, enregistrements et des types abstraits),
- **Signaux** est l'ensemble des signaux paramétrés,
- **Variables** est un ensemble de définitions des variables globales,
- **Files d'attente** assurent la réalisation de la communication asynchrone dans un système. Il n'y a pas de restriction d'utilisation des files. Chaque processus peut envoyer des signaux à travers toutes les files du système.

2.2.2 Processus

Un processus est constitué d'un ensemble de variables locales, d'un ensemble d'états de contrôle et un ensemble de transitions. Nous noterons par q un état et Q l'ensemble des états. Le modèle sous-jacent d'un processus est constitué d'automates communicants ainsi que d'informations additionnelles sur les états.

Un processus **P** $\in$ Processus est un tuple P = (Variables, Q, Transitions) où :

- **Variables** est l'ensemble des variables locales,
- **Q** est l'ensemble des états de contrôle du processus. A chaque état q, sont associés les attributs suivants : stable(q),init(q).

  1. stable(q) :

     Un état peut être stable ou instable. Cette propriété concerne l'atomicité des séquences d'exécution.

  - L'état est instable quand il est définit instable (nostable).
  - Aux états stables, peut être associé une condition de progression du temps *stc*(q). Intuitivement, le processus peut rester dans un état tant que cette condition est vraie. *stc*(q) est un prédicat sur les variables "visibles" dans le processus (variables globales et variables locales de **P**), défini quand le temps progresse.

  2. init(q):

     Un état peut être initial, c'est à dire le processus peut démarrer son exécution à partir de cet état.

- Aux états peuvent être associés des filtres sur les files d'attente. Ces filtres permettent, soit de retarder la consommation de certains signaux (*save*(q)) pour des moments ultérieurs, soit la destruction de signaux considérés inutiles (*discard*(q)).

  1. *save*(q) :
     Ce filtre sur les files d'attente est utilisé pour faire la sauvegarde des états de SDL. Cela permet de préserver tous les signaux de la liste de la file, à chaque fois que la condition de la transition est satisfaite.

  2. *discard*(q) :
     Ce filtre sur les files d'attente est utilisé pour réaliser le rejet des signaux inconsommables de SDL. A la prochaine lecture d'un signal d'entrée contenu dans la file, tous les signaux précédemment écartés feront partie de la même transition atomique.

En résumé aux états nous pouvons donner comme exemple, en respectant les notation de IF une description complète d'un état:

state ::=

state-id init stable

discard signal-filter$_1$,... ,signal-filter$_m$

save signal-filter$_1$,... ,signal-filter$_m$

tpc action**;**

end;

signal-filter ::=

signal-id **in** buffer-id **if** expression**;**

- **Transitions** est un ensemble de transitions de contrôle, constitué de deux types de transitions entre les deux états de contrôle q, $q^{'}$ $\in$ **Q**:
  - Les transitions sont représentées par:

$$q \xrightarrow[(u)]{[condition] \longrightarrow action} q^{'}$$

- [**condition**] est un prédicat représentant la condition de la transition qui dépend des variables visibles dans le processus.
- **action**
  - **input**, **output** signaux de entrées / sortie d'un élément d'une file d'attente .
  - Des affectations simples, comme **:=** affecte à la partie gauche la valeur de la partie droite (TASK en écriture IF),
  - Des affectations spécifiques, comme l'indication d'une suppression d'une variable (RESET en IF), ou récupération de la variable (SET en IF).
- L'attribut u $\in$ {**eager, delayable, lazy**} définit le type de l'urgence de chaque transition. **eager** a une absolue priorité, **delayable** peut laisser passer du temps, mais seulement dans une certaine limite, tandis que **lazy** ne progresse pas en fonction du temps. Les notions de types d'urgence est introduit dans [BST97]

2.3 Conclusion

Nous venons de voir dans ce chapitre les automates communicants AC. Ensuite, l'introduction au modèle IF nous servira pour effectuer nos différentes expérimentations sur un protocole de communication SIP.

A partir du produit synchrone, redéfinit sur les automates communicants, nous allons voir comment obtenir des cas de test. C'est dans ce but, que nous allons introduire dans le chapitre suivant 3, les notions de la relation de conformité, de génération automatique de tests, et de cas de test.

Puis dans la chapitre 4, nous utiliserons toutes ces notions appliquées aux automates communicants, afin d'effectuer une analyse statique pour la génération de tests.

Chapitre 3

Génération automatique de tests de conformité

Le test de conformité [Pha94] [Tre92] a pour objectif de démontrer la conformité entre une réalisation et une spécification. Dans la pratique, la conformité est montrée à l'aide de suites de tests (nous utiliserons une représentation de type graphe pour les tests). Chaque test consiste à exécuter une séquence, et doit aboutir à un verdict. Ces tests sont des systèmes de transitions étiquetées. La norme ISO 9646 [ISO92], dans le cas particulier du test de conformité des protocoles, propose trois sortes de verdict : $Pass$(réussi), $Fail$(raté) et *Inconclusive*(inconcluant). Il est possible de réessayer le test inconcluant pour tenter d'obtenir l'un des deux autres verdicts. Une réalisation est non conforme à la spécification, s'il existe un test dont l'exécution conduit au verdict $Fail$. Souvent effectué à la main, l'automatisation de la génération de tests devient indispensable. Les études (comme par exemple Étude pour la génération de Tests pour le compte de la Direction Générale pour l'Armement) montrent que la génération automatique de tests [Mor00] [FJJV96] apporte un gain en productivité et en qualité.

Dans cette partie nous allons faire le parallèle entre les STE et AC. Les tests dans les outils existants sont basés sur un produit synchrone de STE, l'un représentant la sémantique d'une spécification, l'autre de la propriété. Notre but, est de se baser sur cette composition, pour réaliser une analyse à base d'automate communicant. Sachant représenter des AC en STE, et comme nous allons pouvoir réduire la taille des AC par analyse statique chapitre 4, nous allons nous appuyer sur les relations existantes de conformité pour établir:

$$\begin{array}{ccccc} STE & \otimes & STE & \longrightarrow & test \\ \uparrow & & \uparrow & & \\ AC & \otimes & AC & \longrightarrow & test \end{array}$$

3.1 Relation de conformité

La notion de conformité est basée sur les relations entrées/sorties décrite dans [Pha94], [Tre92]. Elle permet de définir quelles implantations sont conformes à une spécification. Elle consiste à tester une implantation vis à vis des comportements de sa spécification [Org93] à travers des points de contrôle et d'observation (PCO). Cette interaction entre testeur et implantation dans la partie 3.3.1 a pour but de détecter la présence d'erreurs par rapport à la spécification.

Définition 3.1.1 (Relation de conformité :) *C'est la relation entre la représentation de la spécification et l'implantation à tester. Si on note R cette relation alors :*

Si R(spécification, implantation) est vrai selon les règles de [Pha94], [Tre92], alors l'implantation est conforme à la spécification (et vis versa).

3.2 Génération automatique de tests

Le principe général de la génération de tests est d'essayer de modéliser la pratique industrielle et d'utiliser des algorithmes efficaces. Elle consiste à dériver automatiquement les tests depuis la spécification. Cela reste un problème difficile d'un point de vu théorique et pratique. Les experts du test utilisent une méthodologie basée sur :

- identification d'une architecture de test, qui est une description de l'environnement d'exécution de la réalisation,
- écriture en langage naturel des objectifs de tests, permettant de tester un aspect particulier de la réalisation,
- dérivation manuel des tests.

Un travail sur la génération de séquences de tests est mené conjointement à l'IRISA et VERIMAG. Ce travail a conduit à la création d'un outil comme TGV [FJJV96] (IRISA-VERIMAG). Une version de TGV est désormais intégrée dans l'outil commercial *Object*GEODE, TGV génère automatiquement des cas de test à partir d'une spécification et d'un objectif de test (STE). Un cas de test produit par TGV [JM99] [FJJV96] est un graphe d'interaction (entrées / sorties) entre le testeur et l'implantation à tester. Les cas de test sont pourvus d'horloges (timers) et de verdicts. Pendant l'exécution d'un cas de test, s'il y a un blocage au niveau communication (blocage) les horloges incluses au cas de test garantissent une fin d'exécution. Les verdicts sont produits en fonction de la conformité ou non de l'implantation en respectant la spécification. C'est sur cette base que nous étudierons une génération de tests en utilisant le modèle IF.

3.3 Les modèles utilisés

Actuellement, les modèles utilisés sont généralement des STE (*à entrées sorties*). Ces systèmes autorisent la distinction entre les actions observables (les sorties de l'implémantation) et les actions contrôlables (les entrées de l'implémentation). Ce mode de communication caractérise le dialogue entre l'implantation et le testeur. Un testeur est un programme qui identifie les traces non conformes d'une implantation vis-à-vis d'une relation de conformité. Le testeur émet des messages et observe en retour, les réactions de la réalisation.

Nous allons dans cette partie, définir ce qu'est un testeur, une spécification, un objectif de test, puis un cas de test. Ensuite dans les sections 3.1 et 3.2, nous regarderons les différentes relations de conformité utilisées.

3.3.1 Testeur et architecture de test

Pour définir un testeur, nous pouvons dire que l'implantation à tester communique directement avec l'environnement de test. Ce qui permet une *contrôlabilité* et une *observabilité*.

Définition 3.3.1 (Contrôlabilité et observabilité :)

contrôlabilité : *Les entrées de la réalisation correspondent aux sorties du testeur (donc contrôlées par le testeur).*

observabilité : *Inversement, les sorties de la réalisation correspondent aux entrées du testeur (donc observées par le testeur).*

Une architecture de test :

Elle comporte les dispositifs utilisées pour tester la réalisation, et, l'interface à tester permettant l'accès à la réalisation sous test. La notion d'interface de test correspond à celle des *Points de Contrôle et d'observation PCO*. L'architecture de test est prise en compte par transformation de la spécification, en entrelaçant les actions observables qui sont observées sur les *PCO* distincts. Voici figure 3.1 , une présentation d'une architecture de test local, avec un seul testeur qui interagit avec la réalisation.

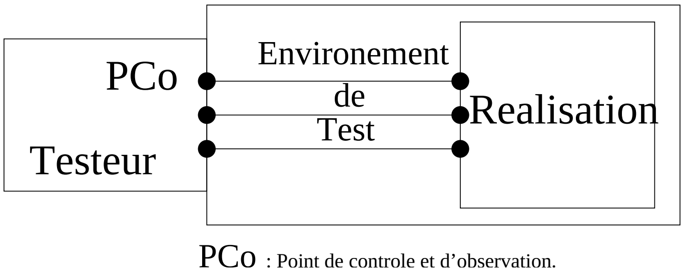

FIG. 3.1 – *Environnement de test*

Lors de l'application d'une séquence de test sur une réalisation, le testeur n'émet et ne reçoit que les interactions décrites par la séquence de test. Il est impossible, du point de vu du testeur, de revenir en arrière et d'observer un branchement non-déterministe de la réalisation. Ce qui fait que deux implantations ayant les mêmes traces ne seront pas distingables.

Remarque 3.1 (Le rôle des observations et des PCO :) *Dans la suite de la vérification, nous nous intéressons qu'aux séquences d'un système respectant un critère. Les informations sont transmises et récupérées par l'intermédiaire des PCO. Les PCO sont au niveau des états du système. Ils nous permettent pendant le déroulement des observations sur les valeurs des variables et de transmette ainsi les états et séquences à garder.*

3.3.2 Spécification

Ici nous tenons compte des actions d'entrées et de sorties ainsi que les actions internes ce qui constituent un ensemble observable. La description de la spécification devient donc :

Définition 3.3.2 (Définition d'une spécification pour le test de conformité :) *La définition de la spécification est basée sur celle des automates étendues vu précédemment.A partir de la spécification nous dérivons une implantation.*

Nous considérons que la spécification est un ensemble d'automates communicants (S, C, P) *où, un processus* $p \in P$ *est un tuple* $(X_p, Q_p, T_p, \Sigma_p, q_p^0)$ *(voir chapitre 2).*

3.3.3 Objectif de test

L'objectif de test un automate communicant muni de deux états marqués "accept" et "reject". Il est relatif à la traduction d'une expression. Cette expression pouvant être donnée en langage naturelle, en formule de logique temporelle ou directement en automate communicant. Un objectif de test décrit généralement un enchaînement d'actions non nécessairement consécutives que nous voulons tester. Il représente de manière informelle quel est l'objectif particulier d'un cas de test. Ainsi, que les contraintes sur les entrées de l'implantation. Une contrainte est une combinaison booléenne d'atomes. Un atome exprime une réduction sur les valeurs utilisées. Un cas de test est une sélection d'une partie de la spécification (série d'actions), afin de tester un point précis de l'implantation. Ce cas de test (3.4) est le résultat de la relation entre la spécification est l'objectif de test.

La définition d'un objectif de test est la suivante:

Définition 3.3.3 (Un objectif de test : nommé ot) *C'est un automate communicant représenté par le tuple* $ot = (Q_{ot}, T_{ot}\ , q^0_{ot}, q^{accept}_{ot}, q^{reject}_{ot}\ )$ *ou*

- Q_{ot} *est un ensemble d'états,*
- $T_{ot} \subseteq Q_{ot} \times \Sigma_{ot} \times Q_{ot}$ *est un ensemble de transitions,*
- Σ_{ot} *est l'ensemble des interactions fixées par l'objectif, il contient les signaux* $c?s(x)$ *et* $c!s(e)$*, ou* $c \in C^{ext}$ *une file externe et* $s \in S$ *un signal et une valeur, et peut aussi comporter* $x := e$ *qui représente une simple affectation, avec* x *une variable, et* e *dénotant une expression,*
- q^{accept}_{ot} *est l'état d'acceptation dans cet automate, sans successeur par* T_{ot}*,*
- q^{reject}_{ot} *est l'état de rejet dans cet automate, sans successeur par* T_{ot}*,*

Exemple 3.1 *Nous pouvons représenter un objectif de test comme le montre la figure 3.2 avec OT = ((0,1,2,3,reject,accept), ([b] f1?s1(x),[b] f1!s1(x) [b] f2!s2(x),[b] f2!s2(x),[b] x:=y,[b] x:=5), 0) :*

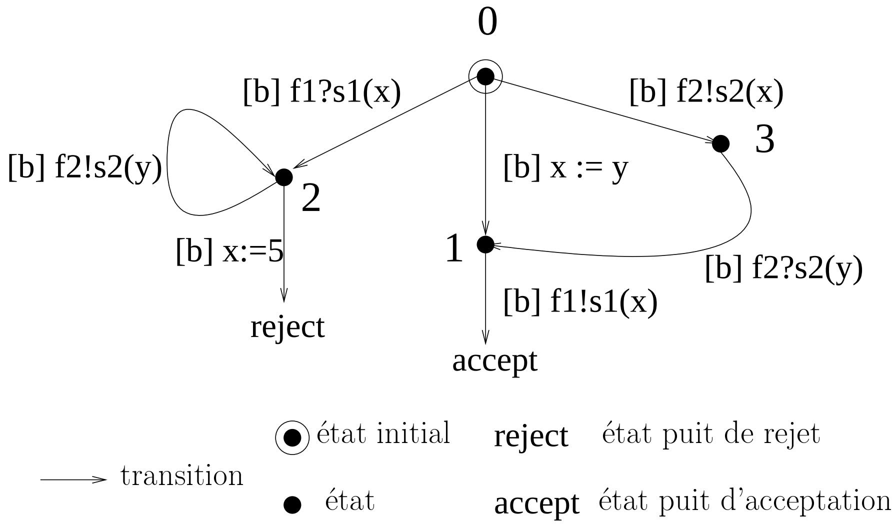

FIG. 3.2 – *Représentation de l'objectif de test OT*

L'état "*accept*" définit le verdict Pass sur les séquences de la spécification reconnues. Par ailleurs, l'état "*reject*" permet de limiter l'exploration du graphe de la spécification en élaguant les séquences reconnues (nous expliquerons de façon plus précise par des techniques de réduction (slicing, réduction du contrôle) dans le chapitre 4, les différentes suppressions de séquences reconnues que nous pouvons faire à partir d'un objectif de test). Ainsi, les séquences de la spécification sont caractérisées "accept", si elles aboutissent à un état "*accept*" ou, reject si elles abboutissent à un état "*reject*".

Nous supposons que l'alphabet des actions de la spécification est identique à celui de l'objectif de test ($A_{ot} = A_s$). Cela autorise les actions de l'objectif de test à être des actions internes de la spécification "s" , ce qui est utile dans le cadre du test dans un contexte. En effet, lorsque la spécification est plongée dans un contexte, par exemple un contexte représenté par des files d'attente, il est intéressant de pouvoir décrire dans l'objectif de test les actions de la spécification. Ces actions sont rendues inobservables par le contexte, les actions observables étant celles qui existent entre le contexte et l'environnement.

Remarque 3.2 (relation entre spécification et objectif de test pour une vérification par produit synchrone :) *Pour générer un test à partir d'une specification et d'un objectif de test, il faut prendre en compte que les actions de sorties de la spécification sont les actions d'entrées du test et que les actions d'entrées de la spécification sont les actions de sorties du test. Nous verrons dans la suite, chapitre 4 que nous allons faire un produit synchrone entre une spécification et un objectif de test. Or comme nous venons de voir que les sorties de l'un correspondent aux entrées* *de l'autre, il faut pour le produit synchrone faire une transposition dite de miroir. Cette transformation peut être faite avant ou aprés le produit synchrone.*

Définition 3.3.4 (transposition miroir:) *Dans ce travail, on considère que cette transposition s'effectue avant de faire le produit synchrone. il s'agit au niveau de l'objectif de test de transposer les entrées en sorties et inversement. Soit A_s^i et A_s^o les actions d'entrées et les actions de sorties, on a :*

$$A_s^o \longrightarrow A_s^i$$
$$A_s^i \longrightarrow A_s^o$$

où i,o sont les entrées/sorties (in/out).

3.3.4 Cas de test

De façon générale le cas de test est représenté sous la forme d'un graphe dont chaque suite de transitions décrit une séquence d'interactions entre le testeur et l'implantation à tester. Le rôle du cas de test est de détecter si l'implantation est conforme à sa spécification, donnée par l'intermédiaire précise d'un objectif. Cela est définie comme une conformité cf 3.1. Le cas de test est un sous graphe du produit synchrone Spec $\otimes$ Ot. Il se veut déterministe, acyclique et satisfait la condition de contrôlabilité. Dans un état si une sortie est possible, c'est la seule action possible. Il comporte des diagnostics exprimés sous forme de verdicts.

Les verdicts:

PASS : Le verdict "Pass" traduit la conformité de l'implantation. Ce n'est plus la peine de poursuivre la vérification après l'obtention de "Pass" car nous savons que la spécification est conforme.

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

FAIL : Si l'application produit "Fail", alors le comportement du testeur n'est pas un comportement conforme à la spécification (exemple: une sortie de l'implantation pas spécifiée).

Pour comprendre la construction d'un cas de test nous allons le présenter sous la forme d'un exemple.

Exemple 3.2 *Après la composition par produit synchrone de l'automate de la spécifi-*

cation avec l'automate de l'objectif de test, nous obtenons un cas de test.

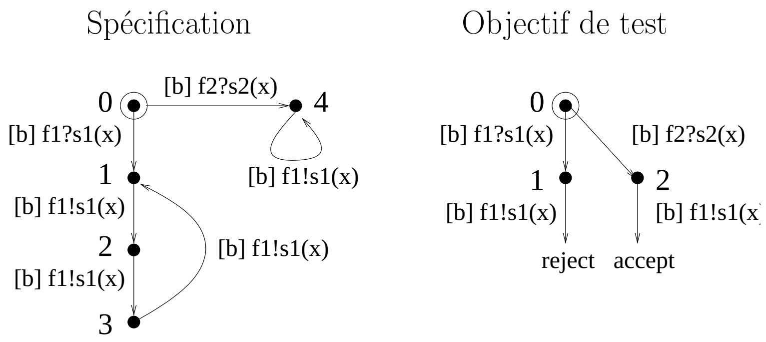

FIG. 3.3 – *Représentation de la spécification et de l'objectif de test*

La figure 3.4 suivante décrit le produit synchrone SP :

SP = (((00),(11),(22),(1reject),(3accept)), ([b] f2?s2(x),[b] s1?f1(x), [b] f1!s1(x)), (00)), ainsi que le cas de test pour de la spécification et de l'objectif de test donnée avant.

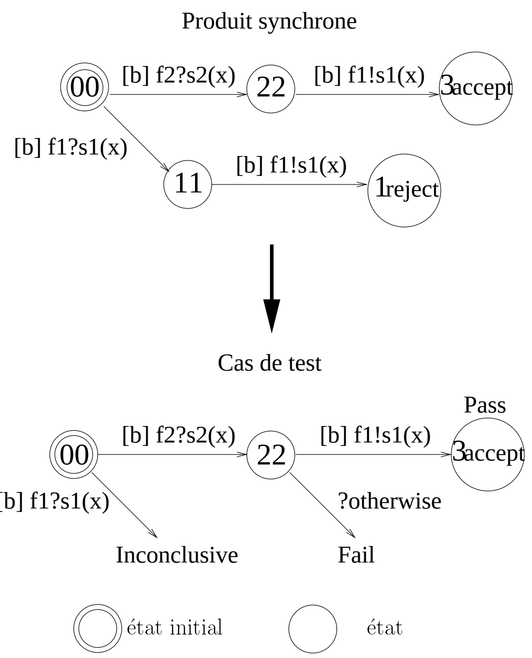

FIG. 3.4 – *Représentation d'un cas de test*

L'exploitation de ces verdicts permet d'obtenir différentes séquences que nous pourrons utiliser pour la détection d'erreur. Ce cas de test a été construit à l'aide du produit synchrone SP. Nous avons garder en intégralité les séquences menant à l'état ($3accept$). Nous considérons dans la définition du cas de test, s'il existe une sortie possible, c'est la seule qui existe d'où l'apparition de $?otherwise$ sur l'état (22). Cela signifie que seule la transition $[b]$ $f2?s2(x)$ mène à un état $Pass$ et que toutes les autres (otherwise) ne sont pas prise en compte. Pas de $?otherwise$ sur l'état (00), car c'est la sortie du testeur en cette état. Le $Inconclusive$ montre ici la non exploration de la partie du graphe $(11), (1reject)$. Elle correspond à l'état ($1reject$) qui est une information remontée au niveau d'un autre point de départ d'un autre chemin, ici celui de $Pass$. D'où la suppression de l'état (11) dans le cas de test. Ce cas de test est donc en relation stricte avec l'objectif de test à savoir, la spécification respecte-t-elle la séquence $(0, 2, accept)$ et la non exploration de toutes les séquences qui sont $(0, reject)$ par une transition précise. Le cas de test a ce niveau isole totalement l'information que nous voulions savoir à travers l'objectif de test.

Remarque 3.3 *Il n'y a pas de à priori limite sur les systèmes de transitions et que le nombre d'interactions peut être infini. Nous ajoutons donc que les graphes d'états des spécifications (la taille de l'objectif étant connue est plus "petite" et finie) peuvent être de taille à priori infinie. C'est pour cela qu'une démarche de la génération de test dite "à la volée" (sans stocker réellement le graphe due à sa taille pouvant être très grande, on ne garde que des informations de passage dans les états et transitions sous formes de marquage) est utilisée dans les réalisations (utilisation et autre explication dans le chapitre 4).*

3.4 Conclusion

Nous avons vu dans ce chapitre que la relation de conformité était importante pour définir les interactions entre la réalisation et l'environnement modélisé par un automate. Ce qui nous a permis d'introduire une modélisation de la génération automatique de tests.

Pour cela avec différentes définitions, nous avons déterminé une architecture de test, qui est le dispositif pour tester une réalisation. Ensuite, nous avons exprimé de façon précise une spécification et un objectif de test à base d'automates communicants. Nous allons maintenant chapitre 4 voir comment l'utilisation du produit synchrone va nous premettre d'obtenir une spécification réduite et conforme à la spécification initiale, en utilisant les notions développées dans cette partie.

Chapitre 4

Analyse statique

Dans ce chapitre, nous allons présenter de façon formelle comment réduire les automates communicants. Pour cela nous allons définir une technique de filtrage (slicing) des automates basée sur un critère d'entrée. Nous entendons par filtrage, la suppression de la partie de graphe n'intervenant pas dans les exécutions selon le critère. Pour cela nous allons utiliser le produit synchrone qui aura comme effet de donner un automate communicant réduit ||AC* ne comportant que des séquences relatives à notre critère d'entrée. En effet l'objectif de test est un critère d'entrée dont l'ordre des actions de ses séquences détermine les séquences d'exécutions de la spécification.

Le problème, lors des traitements, est l'explosion des tailles mémoires. Pour appliquer de façon efficace des méthodes de vérification, il est important de composer sur des STE réduits. En amont, nous voulons réduire la taille des automates et pour cela nous allons étudier une technique existante de réduction, mais qui ne comporte pas l'ordre sur les séquences. Ensuite nous allons adapter cette technique en prenant en compte l'ordre des actions.

La réduction de taille des automates permet de diminuer la quantité d'informations et de ne garder que des séquences pertinentes.

Il existe différentes façons de simplifier les automates :

- Une basée sur une analyse statique avant toutes compositions ou compilations (slicing statique).
- Une autre basée sur une analyse dynamique qui est effectuée pendant la composition ou l'exécution (slicing dynamique).

Nous allons voir dans ce début de chapitre une technique de réduction d'automate avec un critère d'entrée (les Feeds) [BFG00]. Nous introduisons cette représentation pour ensuite, faire le parallèle avec le produit synchrone sur les automates communicants: $\mathrm{AC}_{spec} \otimes \mathrm{AC}_{ot} = \mathrm{AC}'$.

En effet, la technique de slicing que nous allons voir, comporte une spécification et un critère sur les “entrées” du système. Le parallèle est fait en prenant comme critère l'objectif de test au lieu de prendre celui initialement donné. Mais, la différence, est importante, puisque dans le slicing que nous allons présenter nous ne tenons pas compte de l'ordre des actions contenues dans le critère. Or, dans l'objectif de test, l'ordre des exécutions des actions est nécessaire et indispensable pour établir par produit synchrone avec la spécification, des séquences réduites et conformes à la spécification initiale.

Nous pouvons dès maintenant dire que la composition par produit synchrone aura un résultat de réduction sur les automates de la spécification.

La conformité vue au chapitre 3 reste très importante pour les traitements de filtrage. Elle permet de garantir la conformité et l'essentiel des informations des spécifications après suppressions de transitions (d'actions).

4.1 Le slicing

Il existe différentes techniques de filtrage (slicing). Le slicing peut-être effectué de façon dynamique (il est basé sur l'exécution des automates en formalisant les correspondances et relations des instructions, et ainsi supprimer les instructions inutiles), ou de façon statique (il a pour but d'effectuer une réduction de code avant la compilation en écartant toutes les données et actions inutiles pour la réalisation d'un critère d'entrée précis [Tip95], [Wei79]). Nous rappellerons ici une technique de slicing statique [BFG00] basée sur des “entrées” : les **Feeds** définition 4.1.1. Le slicing statique est fait avant la génération de test pour permettre des réalisations de tests sur des spécifications de plus petites tailles.

Définition 4.1.1 (Les Feeds : Σ_f) *Les Feeds est un ensemble de signaux contenu dans l'objectif de test. Les Feeds sont des données qui interviennent dans les échanges entre l'application et son environnement. Ils fournissent un critère de filtrage sur les séquences des différents automates de la spécification concernée.*

Nous mettons en œuvre un slicing statique selon ces entrées en appliquant plusieurs méthodes.

- Analyse de contrôle,
- Analyse des données,
- Propagation des contraintes.

4.1.1 Analyse de contrôle

Nous calculons à partir des données de l'ensemble des Feeds, la détermination et la localisation de ces données dans les différentes séquences des automates de la spécification. Cette partie du filtrage se fait de la façon suivante :

Nous définissons le slicing de la spécification S = (S,C,P) avec l'ensemble des Feeds Σ_f tel que $S\backslash_f \Sigma_f$ = $(S,C,P\backslash_f \Sigma_f)$ où :

$S\backslash_f \Sigma_f$ contient le processus filtré $p^{'}$ pour chaque processus $p \in P$. Le filtrage pour un processus $p = (X_p,Q_p,T_p,q_p^0) \in P$ est définit comme un processus $p^{'} = (X^{'}_p,Q^{'}_p,T^{'}_p,q_p^{'0})$, avec les mêmes ensembles de variables. L'ensemble des états $Q^{'}_p \subseteq Q_p$ et les transitions $T'_p \subseteq T_p$.

$$\frac{-}{q_p^0 \in Q'_p} \qquad \frac{q_p^0 \in Q^{'}_p \ q_p \overset{[b]x:=e}{\longrightarrow} q^{'}_p}{q^{'}_p \in Q^{'}_p \ q_p \overset{[b]x:=e}{\longrightarrow} q^{'}_p \in T'_p} \qquad \frac{q_p^0 \in Q^{'}_p \ q_p \overset{[b]c!s(e)}{\longrightarrow} q^{'}_p}{q^{'}_p \in Q^{'}_p \ q_p \overset{[b]c!s(e)}{\longrightarrow} q^{'}_p \in T'_p}$$

$$\frac{q_p \in Q^{'}_p \ q_p \overset{[b]c?s(x)}{\longrightarrow} q^{'}_p c \in C^{ext} c?s(x) \in \Sigma_f}{q^{'}_p \in Q^{'}_p \ q_p \overset{[b]c?s(x)}{\longrightarrow} q^{'}_p \in T'_p} \qquad \frac{q_p \in Q^{'}_p \ q_p \overset{[b]c?s(x)}{\longrightarrow} q^{'}_p c \in C^{int} \exists r.q_r \overset{[b^{'}]c!s(e)}{\longrightarrow} q^{'}_r \in T'_r}{q^{'}_p \in Q^{'}_p \ q_p \overset{[b]c?s(x)}{\longrightarrow} q^{'}_p \in T'_p}$$

Une fois les séquences isolées par les règles [BFG00] pour obtenir le nouveau processus $p^{'}$, on supprime les parties du graphe ne comportant pas de relation avec les Feeds (tout ce qui n'appartient pas à $p^{'}$).

4.1.2 Analyse des données

L'analyse de contrôle ne tient pas compte de l'ordre des actions, nous verrons comment intégrer ceci grâce au produit synchrone. Le calcul des données est une extension de l'analyse d'activité des variables (Live analysis variable [BFG99a]), pour chaque processus, nous calculons l'ensemble des variables utilisées dans chaque état, à partir des variables intervenant dans les sorties de l'objectif de test, puis on effectue une propagation interautomates. Une variable est utilisée en respectant le critère des sorties de l'objectif de test. Une variable est utilisée dans un état si sa valeur est utilisée pour le calcul de la valeur de paramètres des signaux de sorties dans l'objectif de test, ou s'il existe un chemin état par état ou la variable est utilisée avant d'être redéfinie. Cette recherche des variables utilisées se fait de la façon suivante :

Définition 4.1.2 *Soit* $S = (S,C,P)$ *est une spécification et* $OT = (Q_{ot}, T_{ot}, q_p^0, q_{ot}^{accept})$ *est un objectif de test. L'ensemble* Σ_{ot} *est l'ensemble des signaux de sorties mentionnés dans l'objectif de test. Les variables utilisées sont définies pour chaque processus* $p = (X_p, Q_p, T_p, q_p^0) \in P$ *par la fonction* $Rlv_p : Q_P \longrightarrow 2^{X_p}$ *parcourant les états pour les sous-ensembles de variables. L'ensemble* Rlv_p *des états* $q_p \in Q_p$ *est défini comme le plus petit point fixe de la suite d'équations du système :*

$$Rlv_p(q_p) = \bigcup_{t_p : q_p \xrightarrow{\alpha} q_p'} Rlv_p(q_p') \backslash Def_p(t_p) \ \cup \ Use_p(t_p)$$

$$Def_p(t_p) = \begin{cases} \{x\} \\ \quad if \ t_p = q_p \xrightarrow{[b] \ x:=v} q_p' \\ \quad or \ t_p = q_p \xrightarrow{[b] \ c?s(x)} q_p' \\ \\ \emptyset \ otherwise \end{cases}$$

$$Use_p(t_p) = \begin{cases} vars(b) \ \cup \ vars(e) \\ \quad if \ t_p = q_p \xrightarrow{[b] \ x:=e} q_p' \ et \ x \in Rlv_p(q_p') \\ \quad or \ t_p = q_p \xrightarrow{[b] \ c?s(e)} q_p' \ et \\ \quad\quad c \in C^{ext} \ et \ \exists q_{ot} \xrightarrow{c!s(*)} q_{ot}' \in T_{ot} \ or \\ \quad\quad c \in C^{int} \ et \ \exists r.q_r \xrightarrow{c?s(x)} q_r' \in T_r \ et \ x \in Rlv_p(q_p') \\ \\ vars(b) \ otherwise \end{cases}$$

Les variables utilisées sont calculées simultanément pour chaque processus. Le filtrage fait une réduction du nombre de variables à l'intérieur de chaque processus, relativement aux variables utilisées. Concrètement, les actions qui comportent des variables non utiles sont éliminées.

4.1.3 Propagation des contraintes

La propagation des contraintes permet d'évaluer certaines conditions. Les conditions ne pouvant jamais être atteintes sont potentiellement supprimables. Cette partie est une approche pour simplifier la spécification en n'utilisant que les contraintes imposées par les **Feeds**, les entrées de l'objectif de test. Pour chaque état de contrôle une approximation de l'ensemble des valeurs pour chaque variable est calculée. Cette information est utilisée ensuite, pour évaluer les conditions des transitions et pour éliminer les variables ne pouvant jamais être évaluées selon les conditions.

C'est un problème d'analyse de flot de donnée:

Définition 4.1.3 *Soit S = (C,S,P) est une spécification et Σ_f est l'ensemble des Feeds. Les contraintes sont représentées, pour chaque processus p, comme une fonction Val : $Q_p \longrightarrow 2^D$. Le problème de la propagation des contraintes est définit comme la solution du plus petit point fixe selon les équations du système suivantes :*

$$Val(q_p^{'}) = \bigsqcup_{t_p : q_p \xrightarrow{\alpha} q_p^{'}} Tranfert_{t_p}\ (Val(q_p))$$

avec :

- *Une classe de fonction de $Transfert_{t_p}$ exporte les variables d'état en état, il existe un transfert de donnée pour chaque transition t_p,*
- *La fonction $\bigsqcup$, à chaque état.*

Toutes ces définitions ont pour but de réduire la taille des automates de la spécification. Ce qui va nous permettre de travailler dans le but de faire une génération de tests par produit synchrone en limitant la taille des résultats. En effet nous pouvons par exemple, faire un filtrage des automates de la spécification avant de réaliser les cas de tests. C'est pour cela que nous introduisons le produit synchrone qui ne garde que les séquences pertinentes en relation avec l'objectif de test pour critère de filtrage.

4.2 Le produit synchrone

Nous allons présenter de façon générale le produit synchrone. Pour cela tout au long du développement nous nous appuyons sur un exemple. Cet exemple est constitué, de deux automates communicants pour la spécification et d'un automate pour l'objectif de test. Toutes les définitions faites sur cette parties seront par contre, écrites, pour n automates pour la spécification et un automate pour l'objectif de test. Nous ferons ensuite des considérations notamment au niveau des files d'attente pour obtenir une réduction "maximale" des automates communicants.

Remarque 4.1 *Nous avons choisi de synchroniser totalement les automates. Si nous possédons une synchronisation partielle, nous pouvons la rendre totale. Pour cela, il faut rendre l'automate de l'objectif de test complet, en créant des transitions boucles sur ses états (ceci demande une connaissance exacte des automates de la spécification pendant l'élaboration de l'objectif de test). Où, de mettre l'action * sur sur boucle au niveau état de l'objectif de test pour effectuer la synchronisation. L'action * étant une action spéciale qui se synchronise avec toutes les actions non déterminées par l'objectif de test par rapport à la spécification.*

Rappel 4.1 (relation entre spécification et objectif de test:)

$$A_s^i = A_{ot}^o,\ A_s^o = A_{ot}^i,\ A_s^{interne} = A_{ot}^{interne}$$

- $A_s,\ A_{ot}$ *sont les actions de la spécification et l'objectif de test,*
- *i,o sont les entrées/sorties (input/output).*

Rappel 4.2 (syntaxe des automates de la spécification) *Une spécification est un ensemble d'automates communicants, représenté par le tuple (S, C, P) où, S est un ensemble de signaux, C est un ensemble de files d'attente (interne ou externe), $C = C^{int} \cup C^{ext}$, P est l'ensemble des processus (ou automates).*

Un processus $p \in P$ est un tuple $(X_p, Q_p, T_p, \Sigma_p, q_p^0)$ avec, X_p est un ensemble de variables locales, Q_p est un ensemble d'états, Σ_p est un ensemble d'actions qui peuvent être effectuées par p, $T_p \subseteq Q_p \times \Sigma_p \times Q_p$ est un ensemble de transitions.

La construction de l'objectif de test se fait en connaissant les séquences d'exécutions qui doivent être testées. Toutes les séquences de la spécification que nous ne voulons pas tester sont non décrites dans l'objectif de test. Nous avons donc choisit ici que l'objectif de test ne comporte pas l'action spéciale *.

Rappel 4.3 (syntaxe d'un automate d'un objectif de test) *Cet automate est un tuple est un tuple $(X_{ot}, Q_{ot}, T_{ot}, \Sigma_{ot}, q_{ot}^0, q_{ot}^{reject}, q_{ot}^{accept})$ avec, X_{ot} est un ensemble de variables locales, Q_{ot} est un ensemble d'états, Σ_{ot} est un ensemble d'actions, $T_{ot} \subseteq Q_{ot} \times \Sigma_{ot} \times Q_{ot}$ est un ensemble de transitions.*

4.2.1 Le produit synchrone sans abstraction

Définition 4.2.1 (Produit synchrone des automates communicants :) *Nous avons pour effectuer le produit synchrone* $\otimes$ *(*$(p_1, p_2, \ldots, p_n)$*,ot,*$\Sigma_\otimes$*), ou, (*$p_1, p_2, \ldots, p_n$*) sont les n processus de la spécification et ot l'objectif de test.*

Le produit synchrone est le tuple ($Q_\otimes, T_\otimes, q^0_\otimes$*) tel que:*

- $Q_\otimes \subseteq Q_1 \times Q_2 \times \cdots \times Q_n \times Q_{ot}$ *ensemble des états du produit,*

  Précisons que :
- $\Sigma_\otimes = \bigcup_{i=1}^{n} \Sigma_i \cap \Sigma_{ot}$
- *sans oublier que l'objectif de test comporte deux états l'un* q_{ot}^{accept} *pour déterminer un état d'acceptation, et* q_{ot}^{reject} *pour déterminer les rejets.*
- *ou* $Q_\otimes$ *et* $T_\otimes$ *sont les plus petits ensembles obtenus à partir des règles suivantes:*

Les règles se décomposent de la façon suivante, (relation faite sur un processus de la spécification). Le déroulement reste identique pour chacun des n processus.

$$\frac{-}{(\theta, q^0_{ot}) \in Q_{\otimes e}}$$

$$\frac{(\theta, q_{ot}) \in Q_\otimes \;\; q_i \xrightarrow{[b]c?s(x)} q_i^{'} \;\; q_{ot} \xrightarrow{[b]c?s(x)} q_{ot}^{'}}{(\theta', q_{ot}^{'}) \in Q_\otimes \;\; (\theta, q_{ot}) \xrightarrow{[b]c?s(x)} (\theta', q_{ot}^{'})}$$

$$\frac{(\theta, q_{ot}) \in Q_\otimes \;\; q_i \xrightarrow{[b]c!s(e)} q_i^{'} \;\; q_{ot} \xrightarrow{[b]c!s(e)} q_{ot}^{'}}{(\theta', q_{ot}^{'}) \in Q_\otimes \;\; (\theta, q_{ot}) \xrightarrow{[b]c!s(e)} (\theta', q_{ot}^{'})}$$

$$\frac{(\theta, q_{ot}) \in Q_{\otimes} \;\; q_i \xrightarrow{[b]x := e} q_i^{'} \;\; q_{ot} \xrightarrow{[b]x := e} q_{ot}^{'}}{(\theta', q_{ot}^{'}) \in Q_{\otimes} \;\; (\theta, q_{ot}) \xrightarrow{[b]x := e} (\theta', q_{ot}^{'})}$$

Les règles sont applicables de la même façon sur chacun des états des différentes processus de la spécifications que comportent la réalisation. Une réalisation graphique sera présentée dans la figure 4.3

Regardons à présent le produit synchrone SP $= (p_1, p_2) \otimes$ ot. Nous utilisons des automates abstraits qui ne comportent plus de variables, puis plus de files d'attente.

Exemple 4.1 *Les deux spécifications p_1, p_2 sont représentées de la façon suivante :*

$$p_1 = ((0,1,2,3,4), ([b]\ f1!s1,[b]\ f1?s1), 0),$$
$$p_2 = ((0,1,2,3), ([b]\ f1?s1,[b]\ f1!s1), 0),$$

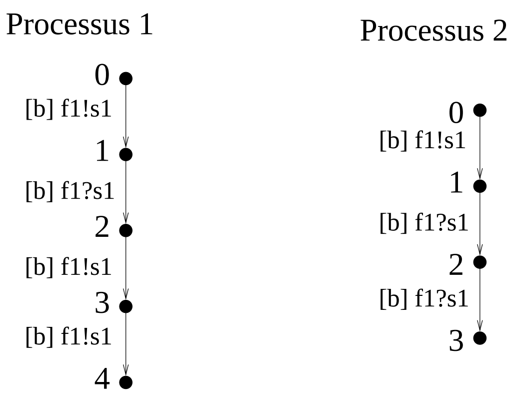

FIG. 4.1 – *Représentation des automates p_1, p_2 pour une spécification*

L'objectif de test, ot, est représenté par :

$$ot = ((0,1,2,accept), ([b]\ f1!s1(x),[b]\ f1?s1(x)), 0),$$

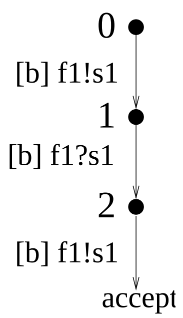

FIG. 4.2 – *Représentation de l'automate pour l'objectif de test (ot)*

SP = (((000),(101),(011),(202),(112),(022),(30accept)), ([b] f1!s1,[b] f1?s1), (000))

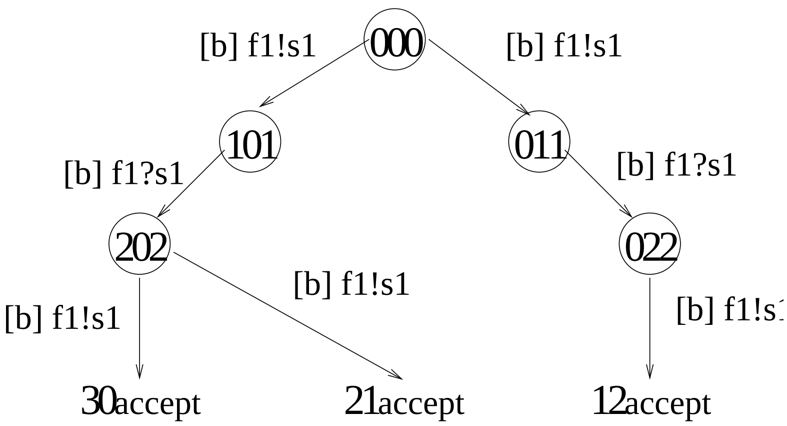

FIG. 4.3 – *Produit synchrone des automates 1, 2 avec ot*

Pour réaliser ce produit nous travaillons sur des automates dont nous faisons l'abstraction des files d'attente et des variables. Voici comment ces abstractions sont définis.

4.2.2 Abstraction sur les variables

Nous nous interressons pas aux valeurs des variables.

Définition 4.2.2 (Abstraction des variables des automates communicants:)

- $\rho : \bigcup_{p \subseteq P} X_p \longrightarrow D$ *qui associe pour chaque variable la même valeur* $v^{'}$,
- $\theta = \langle q_1, .., q_n \rangle \subseteq \times_{p \in P} Q_p$ *est un état de contrôle.*
- $\delta : C^{init} \longrightarrow S^*$

$$\frac{q_p \xrightarrow{[b]x:=e} q'_p}{(\delta, \theta) \xrightarrow{\tau} (\delta, \theta')}$$

$$\frac{q_p \xrightarrow{[b]c!s(e)} q_p^{'} \quad c \in C^{ext}}{(\delta, \theta) \xrightarrow{c!s} (\delta, \theta')} \qquad \frac{q_p \xrightarrow{[b]c!s(x)} q_p^{'} \quad c \in C^{int},\ \delta(c) = w}{(\delta, \theta) \xrightarrow{c!s} (\delta[w.s(v)/c], \theta')}$$

$$\frac{q_p \xrightarrow{[b]c?s(x)} q_p^{'} \quad c \in C^{ext}}{(\delta, \theta) \xrightarrow{c?s} (\delta, \theta')} \qquad \frac{q_p \xrightarrow{[b]c?s(x)} q_p^{'} \quad c \in C^{int},\ \delta(c) = s(v).w}{(\delta, \theta) \xrightarrow{c?s} (\delta[w/c], \theta')}$$

4.2.3 Abstraction sur les files

Nous ne nous intéressons qu'au fait qu'une file a reçu un message au moins une fois. Dans le cadre de ce travail, nous supposons que cette abstraction est faite après l'abstraction sur les variables.

Définition 4.2.3 (abstraction sur les files d'attente des automates communicants :)

- $\rho : \bigcup_{p \subseteq P} X_p \longrightarrow D$ *qui associe pour chaque variable une même valeur* v.
- $\delta : C \longrightarrow S \longrightarrow \{0, 1\}$

- *La valeur initiale de la file d'attente est :* $\delta(c)(s) \;=\; 0$
- $\theta = \langle q_1, .., q_n \rangle \subseteq \times_{p \in P} Q_p$ *est un état de contrôle.*

$$\frac{q_p \xrightarrow{[b]x := e} q_p'}{(\delta, \theta) \xrightarrow{\tau} (\delta, \theta')}$$

$$\frac{q_p \xrightarrow{[b]c!s(e)} q_p^{'} \quad c \in C^{ext}}{(\delta, \theta) \xrightarrow{c!s} (\delta, \theta')} \qquad \frac{q_p \xrightarrow{[b]c!s(e)} q_p^{'}\ ,\ \delta^{'} = \delta(c)(s) = 1}{(\delta, \theta) \xrightarrow{c!s} (\delta', \theta')}$$

$$\frac{q_p \xrightarrow{[b]c?s(x)} q_p^{'} \quad c \in C^{ext}}{(\delta, \theta) \xrightarrow{c?s} (\delta, \theta')} \qquad \frac{q_p \xrightarrow{[b]c?s(x)} q_p^{'} \quad c \in C^{int}, \quad \delta(c)(s) = 1}{(\delta, \theta) \xrightarrow{c?s} (\delta, \theta')}$$

Après les considérations que nous venons de voir, nous nous apercevons que le produit généré est encore composé de séquences qui peuvent être réduites. Nous introduisons maintenant une étude sur les files d'attente.

4.3 Contraintes sur les files

Les contraintes engendrées par l'ordre des communications entre automates peuvent provoquer des interblocages. Le fait de déterminer ces blocages assez tôt vont nous permettre de réduire la taille des automates résultats.

Définition 4.3.1 (Blocage :) *Nous appelons blocage le fait de ne pas pouvoir exécuter la suite de la communication. Nous pouvons avoir un blocage quand un processus demande à consommer un message alors que celui-ci n'est pas disponible dans la file correspondante au moment de l'appel.*

Dans une communication classique entre deux intervenants, il y a échange de messages. Pour connaître si un message a été envoyé, il suffit de savoir s'il est dans la file d'attente.

Si le message que l'on désire est contenu dans cette file, il sera facile de le récupérer et ainsi de continuer les communications du processus. C'est pour cela que pour chaque variable il faut effectuer sa propagation pour savoir qui pose un message et qui le récupère.

Nous allons d'abord montrer sur un petit exemple de la figure 4.4, quelles sont les conditions de blocages. Nous essaierons ensuite d'apporter une réponse pour détecter et ainsi rendre un verdict plus proche de la situation.

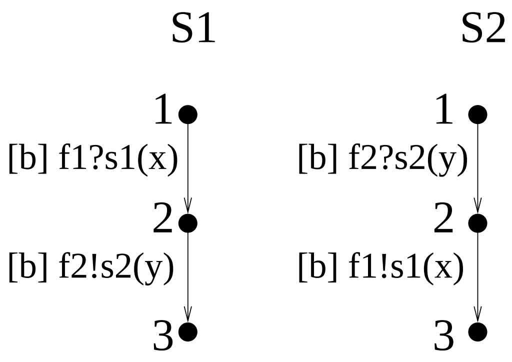

FIG. 4.4 – *Situation de blocage de deux processus*

En regardant cet exemple ou il y a une communication entre deux processus, les deux sont en attente l'un avec l'autre. Nous pouvons à ce stade redéfinir le produit synchrone en tenant compte du contenu des files. Les files d'attente sont réduites par une interrogation 0 pour une file vide ou 1 pour une file contenant un message, mais sans distinction des messages en appliquant $\delta\ : C \longrightarrow S \longrightarrow \{0, 1\}$

Définition 4.3.2 (Produit synchrone des automates communicants :) *Nous avons pour effectuer le produit synchrone* $\otimes$ *(*$(p_1, p_2, \ldots, p_n)$*,ot,*$\Sigma_\otimes$*), ou, (*$p_1, p_2, \ldots, p_n$*) sont les n processus de la spécification et ot l'objectif de test.*

Le produit synchrone est le tuple $(Q_\otimes, T_\otimes, q^0_\otimes)$ *tel que:*

- $Q_\otimes \subseteq Q_1 \times Q_2 \times \cdots \times Q_n \times Q_{ot}$ *ensemble des états du produit,*
- $\Sigma_\otimes = \bigcup_{i=1}^{n} \Sigma_i \cap \Sigma_{ot}$ *et*
- *ou* $Q_\otimes$ *et* $T_\otimes$ *sont les plus petits ensembles obtenus à partir des règles suivantes:*

$$\frac{-}{(\theta^{init}, q_{ot}^{0}, \delta) \in Q_{\otimes}}$$

$$\frac{(\theta, q_{ot}, \delta) \in Q_{\otimes} \; q_i \xrightarrow{[b]c?s(x)} q_i^{'} \; q_{ot} \xrightarrow{[b]c?s(x)} q_{ot}^{'} \;\wedge\; \delta(c)(s) = 1}{(\theta^{'}, q_{ot}^{'}, \delta) \in Q_{\otimes} \; (\theta, q_{ot}, \delta) \xrightarrow{[b]c?s(x)} (\theta^{'}, q_{ot}^{'}, \delta)}$$

$$\frac{(\theta, q_{ot}, \delta) \in Q_{\otimes} \; q_i \xrightarrow{[b]c!s(x)} q_i^{'} \; q_{ot} \xrightarrow{[b]c!s(x)} q_{ot}^{'} \;\wedge\; \delta^{'} \;=\; \delta(c)(s) = 1}{(\theta^{'}, q_{ot}^{'}, \delta^{'}) \in Q_{\otimes} \; (\theta, q_{ot}, \delta) \xrightarrow{[b]c!s(x)} (\theta^{'}, q_{ot}^{'}, \delta^{'})}$$

$$\frac{(\theta, q_{ot}, \delta) \in Q_{\otimes} \; q_i \xrightarrow{[b]e := v} q_i^{'} \; q_{ot} \xrightarrow{[b]x := e} q_{ot}^{'}}{(\theta^{'}, q_{ot}^{'}, \delta) \in Q_{\otimes} \; (\theta, q_{ot}) \xrightarrow{[b]x := e} (\theta^{'}, q_{ot}^{'}, \delta)}$$

le fait d'introduire les files d'attente permet de réduire le résultat. Résultat que nous allons interprété grâce à une projection.

A ce stade nous avons fait $\|AC \otimes OT = \|AC^{'}$. Maintenant nous vous obtenir les automates réduit au niveau la spécification, et cela s'obtient en appliquant la projection, projection($\|AC^{'}$)=$\|AC^*$.

4.4 Projection

La projection est la relation qui permet de garder que les séquences pertinentes des automates de la spécification originelle. Le produit synchrone a déterminé les séquences à couper, la projection les coupe. Les définitions et principes qui vont être présentés vont se baser sur les vecteurs d'état. Il s'agit de récupérer les états puis les transitions, pour redonner les automates communicants.

4.4.1 Définitions et principes

Tout d'abord nous précisons, les conditions d'acceptation pour établir une projection.

- la reconnaissance d'un couple d'acceptation permet à la projection d'avoir l'état final d'une séquence pertinente,
- chemin d'acceptation à partir de l'état initial jusqu'à l'état final,
- La fonction de projection est noté projection(q) où q est un état à projeter,
- Enfin les états faisant partie d'un vecteur d'état, ce vecteur sera composé $((q_1^i, q_1^j, \ldots), q_{ot}^l)$ ou $q_1^i, q_1^j, \ldots$ sont les différents états des automates de la spécification et q_{ot}^l un état de l'objectif de test ou l'on peut représenter un état quelconque ou un état spécial comme "accept" ou "reject".

Définition 4.4.1 (Couple d'état d'acceptation :) *Dans le déroulement de l'exécution nous voulons garder des séquences conformes à nos attentes. C'est pour cela que nous utilisons un objectif de test comportant l'état* **accept***. Dés la parution de* $(\theta,\ q_{ot}^{accept})$ *avec* $\theta = (\ldots, (q_s^{'}, \ldots))$*, nous pouvons dire que nous avons un couple d'acceptation, nous utiliserons donc :*

- $Acc = Q_{\otimes} \cap \theta, q_{ot}^{accept}\ \{(\theta,\ q_{ot}^{accept}) \mid \theta \in Q_1 \times \ldots Q_2 n\ \}$

Définition 4.4.2 (Chemin d'acceptation en accord avec le produit synchrone :) *Soit la suite* $(\theta^{init}, q_{ot}^{init})$*,*$(\theta^i, q_{ot}^i)$*, ... ,* $(\theta^n, q_{ot}^{accept})$*, est un chemin d'acceptation tel que* $\forall\ i < n\ \exists\ \theta^i \in Pre(\theta^{i+1})$.

Un chemin est noté par Acc = Pre*(Acc)*

Définition 4.4.3 (Projection :) *La projection des automates communicants de la spécification est obtenu à partir de* $\|AC \otimes OT = AC^{'}$ *(X,Q,*$q_i, \ldots$*).*

Soit : Projection($AC^{'}$*) = AC*.*

Soit S=(S,C,P) une spécification, OT un objectif de test. On définit la projection note $p_i^{'}$ *du produit synchrone sur chaque processus* p_i *de la manière suivante: Si* p_i*=* $(X_p^i, Q_p^i, T_p^i, \Sigma_p^i, q_p^{0i},)$

alors $p_i^{'}$*=* $(X_p^{'i}, Q_p^{'i} T_p^{'i}, \Sigma_p^{'i}, q_p^{0^{'}i},)$

où $Q_p^i = \{\ q_i\ \in\ Q_p^i \mid (\theta, q_{ot}) \in Pre^*(Acc)\ et\ \theta^i = q^i\}$

Si θ*=*$(q_1, \ldots, q_n)$*, alors* $\theta^i\ =\ q^i$ *(la projection d'un élément du produit cartésien sur la ième composante).*

où $T_p^{'i}$ *est l'ensemble des transitions sortantes des états de* $Q_p^{'i}$

Proposition 4.1 *Soit s = (S,C,P) un spécification, OT un objectif de test et* $s^{'}$ *le résultat du produit synchrone suivit d'une projection.*

Le produit synchrone entre $s \otimes OT$ *et le produit synchrone entre* $s^{'} \otimes OT$ *sont bisimilaires.*

4.5 Conclusion

Nous venons de voir dans cette partie que la difficulté à gérer dés que nous voulons faire des tests, est la taille des automates. Nous avons introduit une technique permettant de réduire cette taille basée sur un cirtère d'entrée. Pour établir cette réduction, nous nous sommes basés sur un produit synchrone qui nous a donné les séquences d'exécution possibles à partir de nos objectifs de tests. Pour améliorer cette réduction et comme nous travaillons sur des représentations expressives, nous avons pu introduire une restriction sur les files d'attente. Le fait de prendre en compte des files d'attente dans la composition, nous permet de détecter des conditions ou nous ne pouvons pas obtenir la donnée souhaiter et donc réduire l'automate par suppression de la séquence. Comme les travaux actuels concerne les STE nous avons du faire une projection du résultat du produit synchrone pour retrouver des automates de la spécification, et ainsi réobtenir une spécification réduite.

Chapitre 5

Réalisation

Dans ce chapitre nous allons présenter l'algorithme du produit synchrone d'un automate pour la spécification et d'un objectif de test. Cette partie comporte un decription de l'algorithme, ainsi que sa mise en œuvre avec les différentes structures de données employées.

5.1 Algorithme du produit synchrone

L'algorithme est basé sur le parcours en profondeur des automates que comporte la spécification, dans le but d'établir une synchronisation avec l'automate de l'objectif de test.

Deux actions sont faites en parallèle : établir la relation entre les actions des automates de la spécification et celui de l'objectif de test tandis qu'une pile est gérée dans le but de garder les séquences communes entre spécification et l'objectif de test.

L'algorithme procède comme suit:

Au départ il place dans la pile les états initiaux des différents automates. Ce vecteur d'état contient un vecteur d'état initiaux des automates de la spécification et l'état initial de l'objectif de test. Notons dès maintenant que, s'il n'existe pas d'état dans le vecteur, il n'y a pas de traitement et de plus si l'état initial est soit "accept", soit "reject" il y a fin du traitement. A partir du vecteur stocké en sommet de pile il y a une recherche des transitions sortantes communes à l'état de la spécification et à celui de l'objectif de test pour en former un couple. Dès que tous les couples de transitions sont rangés dans la pile, il y a début de l'algorithme en profondeur d'abord en suivant une séquence de l'objectif de test relatif à une séquence d'un des automates de la spécification. On empile un nouveau vecteur d'état, à partir de la transition α qui comporte un état cible dans l'automate de la spécification et l'état cible dans l'objectif de test comme suit : $(q_s,\ q_{ot}) \xrightarrow{\alpha} (q_s^{'},\ q_{ot}^{'})$, α devenant une transition analysée. Si $q_s^{'}$ est un état d'acceptation alors on remonte l'information jusqu'à obtenir un état q $\in$ à la séquence menant à $q_s^{'}$ et dont il existe une transition sortante (autre que celle de la séquence considérer en acceptation) qui est commune à une transition sortante de l'objectif de test à ce niveau, alors on reprend la recherche en profondeur par cette séquence. Quand toutes les transitions sont visitées on dépile les vecteurs d'états restant de la pile en prenant soin de garder les états faisant parties des séquences que nous devons garder.

Rappel 5.1

Définition 5.1.1 (Parcours en profondeur) *Le parcours en profondeur est la manière de parcourir les automates des processus. Le principe est de partir de l'état initial est de progresser tant qu'il y a une transition sortante de l'état qui n'a pas été empruntée. Dés que nous ne pouvons plus avancer, il suffit de remonter sur l'état précédent et ainsi de suite tant que les états précédents ont toutes leurs transitions sortantes vues. Dès qu'une des transitions est à analyser, on recommence la progression par ce chemin. Le parcours s'arrête lorsque toutes les transitions ont été visitées.*

Définition 5.1.2 (Construction à la volée) *La construction à la volée [FJJM92] est utilisée pour minimiser le stockage des informations. Elle détermine, si nous avons effectuer une exécution complète en marquant les passages. Le principe est de parcourir le sous-graphe nécessaire à la vérification de la propriété (alogrithme de model checking) ou à la construction de cas de test.*

5.2 Structures de données

Pour réaliser le produit synchrone de façon abstraite (parcours à la volée), nous allons utiliser une structure de pile voir figure 5.1. Cette manière de représenter nos informations pourra nous permettre de faire un parcours en profondeur dans les automates de la spécification. Pour garder le fait d'avoir vu une transition ou un état nous allons établir un **marquage**. Ensuite le couple état spécification et état objectif de test sera intégré à la pile. Ceci nous l'appellerons **empiler**. La pile se construit par récupération des états cibles grâce aux couples de transitions sortantes des états déjà empilés.

Définition 5.2.1 (Marquage des éléments :) *Le terme marquage sera décliné en deux propositions :*

- *Par* **dansPile :**

  il est à 1 si un élément est dans la pile, 0 sinon

- *Par* **visite** *:*

il est à 1 c'est une simple visite d'un élément,

il est à 2 pour une visite d'un élément faisant partie d'une séquence d'acceptation.

il est à 3 pour une visite d'un élément faisant partie d'une séquence de rejet.

Pour éviter le stockage abondant des informations dans la pile, dès l'obtention d'un **couple d'état d'acceptation**, nous mettons à jour les couples d'états visités qui ont amenés à ce couple d'acceptation et nous retirons de la pile les couples ainsi marqué. Puis **depiler** jusqu'à ré-obtenir un état menant à un autre chemin (principe du **parcours en profondeur**) ou obtenir le couple initial de la pile.

Regardons d'abord par une figure 5.1, la structure pile.

– Pour un processus de spécification, un objectif de test.

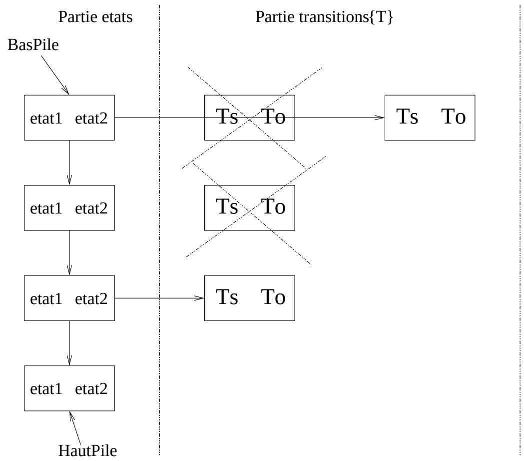

FIG. 5.1 – *Pile et Structure de donnée*

– Pour n processus de spécification, un objectif de test (n étant un nombre fini de processus).

La partie état range les vecteurs d'état, la partie transition range tous les couples de transitions formés d'une transition sortante d'état d'un processus de la spécification et d'une transition sortante de l'objectif de test.

– Principe des différents marquages :

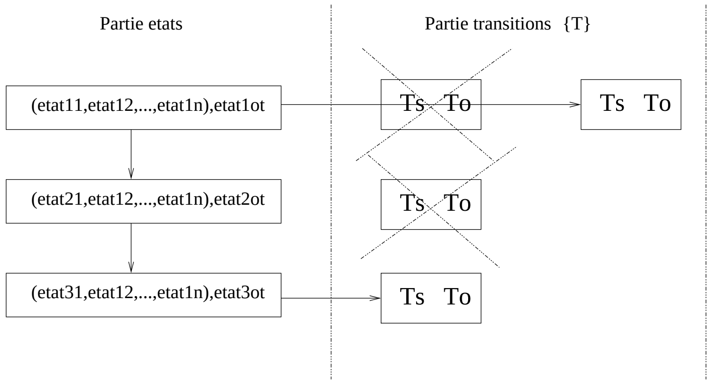

FIG. 5.2 – *Pile et Structure de donnée avec vecteur d'état*

La mise à jour remplit les tableaux de **marquage** des différents couple d'état suit les règles suivantes : Pour la réalisation il existe deux tableaux différents.

– Un qui est destiner pour la situation de dans Pile ou non dans Pile.

Vecteur d'etats	dansPile
etat1, etat2 ,.....etatot	0 ou 1

FIG. 5.3 – *Structure d'un tableau de marquage d'un état dans une pile*

– Et un qui correspond à la visite de l'état.

Vecteur d'etats	Visite
etat1, etat2 ,.....etatot	0 ou 1 ou 2 ou 3

FIG. 5.4 – *Structure d'un tableau de marquage d'une visite d'un état*

– On empile

Lorsque l'on empile un état $(p^{'}, q^{'})$ les marques sont mises à jour de la manière suivante:

dansPile$(p^{'}, q^{'}) \longleftarrow 1;$

visite$(p^{'},q^{'}) \longleftarrow q^{'}$;

$q^{'}$ = “accept” alors 2;

$q^{'}$ = “reject” alors 3;

sinon 1;

– On dépile

Lorsque l'on depile la marque est **dansPile** = 0 et garde les marquages:

la marque de **visite** = 1 si nous sommes sur un chemin sans dictinction particulière

la marque de **visite** = 2 si accept ou si nous sommes sur un chemin accept

la marque de **visite = 3** si reject ou si nous sommes sur un chemin reject

5.3 Mise en œuvre

Corps du l'algorithme Principale:

empiler(sepc,ot,Trans(q,p))
tant que (Pile != $\emptyset$) faire

Soit (p,q,T) le sommet de pile
Si T != $\emptyset$ alors
$(p', q') =$ récupererEtatCible(t)
supprimer(t,T)

Si $(p', q') \notin$ Pile $\wedge \neg$ visite(p', q')alors
empiler$(p', q', Trans(p', q'))$;
visite$(p', q') = 1$;

Si $(p', q') \in$ Pile alors
Selon visite (p', q')

- p'=1 → boucle : L'état p' est dans la pile est fait partie d'une boucle par la dernière transition empruntée. Pas de changement du niveau du marquage de l'état. On recommence au niveau de cet état p pour un nouveau parcourt en profondeur.
- p'= 2 → cas de accept : L'état p change de marquage : marq(p) = 2, nous gardons dans le tableau résultat l'état p.
- p'= 3 → cas de reject : L'état p change de marquage : marq(p) = 3, s'il reste des transitions sortante l'état p est en attent d'être gardé. Si il ne reste pas d'autre transition sortant, la marque reste 3 mais pas d'état p dans le résultat final.

Fin Si

Sinon
depiler(p,q);
fin tant que

FIG. 5.5 – *Algrorithme principal pour effectuer un produit synchrone*

Remarque 5.1 *Pas d'actualisation des transitions car le principe de la pile fait en sorte que la transition est analysée et donc le couple de transitions est définitivement traité.*

Trans : la fonction qui à partir d'un couple d'état (p,q) donne le couple s'il existe (p',q') null sinon

Trans(p,q) = { (p,q)$\xrightarrow{\alpha}$ (p',q') | p $\xrightarrow{\alpha}_{spec}$ p' $\wedge$ q $\xrightarrow{\alpha}_{ot}$ q'}

Variables:

- De type Liste chaînée de couple de $IfTransition$ **T** est l'ensemble des couples (T_s ,T_o) transitions communes sortantes de la spécification et de l'objectif de test $\rightarrow$ ensemble noté : **T**
- De type couple de $IfTransition$, **($\mathbf{T}_s$,$\mathbf{T}_o$)** est, T_s la transition sortante d'un état source *spec* et T_o la transition sortante d'un état source *ot*. noté dans **t** dans l'algorithme. La constitution d'un couple est fait tel que les transitions expriment le même signal. Ces deux transitions sont alors comparables et sont identiques.
- De type $IfState$, **p** et **q** représentent les états sources de T ou initiaux pour le premier pas du corps du programme.
- De type $IfSate$, **p'** et **q'** représentent les états cibles de par **t** de **T**.
- De type ($IfState$ et $Iftransition$), **Pile** est la structure de parcours du produit synchrone.

  Les variables suivantes sont des tableaux de marquages des états selon s'ils sont, ont été, ou non visités. Ou si les états sont ou ne sont plus dans la pile.
- De type ($IfState$ *entier*), **visite** est la structure tableau des états visité.
- De type ($IfState$ *entier*), **dansPile** est la structure tableau des états qui sont dans la pile.

5.4 Interprétation

L'algorithme présenté construit un produit synchrone à la volée. Pour réaliser cet algorithme nous avons choisit une structure de pile qui part son fonctionnement nous fait parcourir tous les états de la spécification en ne gradant qu'un minimun d'information. Il ne s'agit pas de reconstruire dans une pile les chemins d'acceptation, mais de marquer les états et transitions qui font parties des chemins. Cette technique nous permet de traiter des spécifications relativement importantes en nombre d'états et de transitions. Dans cette interprétation nous allons montrer le fonctionnement de la structure de donnée pile couplé avec les marquages. Puis nous interpréterons l'algorihme du produit synchrone lui-même.

Avec le marquage des états par l'intermédiaire du tableau visite, nous allons pouvoir établir une projection afin d'extraire les états contenus dans les séquences et l'informations d'eventuels rejets.

Chapitre 6

Expérimentation

Le but de cette présentation est d'effectuer des tests symboliques sur le protocole SIP. Pour générer ces tests, nous faisons une composition par produit synchrone des automates communicants du protocole SIP et de l'objectif de test. L'étude est faite avec des automates communicants abstraits. L'abstraction est basée sur les variables et les files d'attente, comme cela a été vu dans le chapitre 4. Puis nous réalisons une projection.

Ce chapitre est constitué de la présentation du protocole SIP : Modélisation du protocole, application de l'algorithme du produit synchrone, et projection.

6.1 Le protocole SIP

6.1.1 Présentation

Le protocole SIP (Session Initiation Protocol) [HSS] est un protocole de communication entre un Client et un Serveur ("server"). Son utilisation reste courante pour les conférences multi-média. Il correspond à un algorithme d'échanges de messages. Ces messages sont placés dans les files d'attente pendant une communication.

Après la communication établit le protocole fonctionne en mode UDP (User Datagram Protocol).

Remarque 6.1 *Le principe UDP ne fait pas de récupération des erreurs, pas de contrôle de flux. Il fait une encapsulation des donnée en message au port destinataire. Les communications en mode TCP (Transmission Control Protocol) restent optionnelles.*

6.1.2 Modélisation

Nous avons considéré pour l'expérimentation une communication entre le client et le serveur. Trois phases pour une communication sont montrées sur la figure 6.1, une demande de communication, une communication et un arrêt de communication. Nous verrons comment on peux les tester avec un objectif de test.

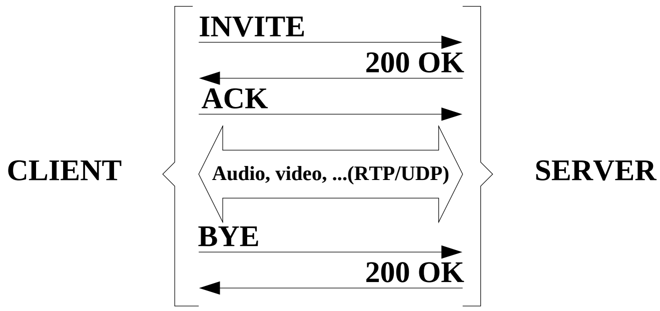

FIG. 6.1 – *Exemple de communication du protocole SIP*

Interprétation de la figure 6.1

La connexion se fait par une INVITE du "client". Si le "server" est près à recevoir la communication, il le fait savoir par un code 200. Le "client" répond et la communication type UDP commence. La fin est stipulée par un BYE du "client", puis la réponse de fin du "server" par le même code 200.

Nous allons regarder maintenant comment se composent le client et le serveur. Nous allons faire une présentation graphique des deux intervenant, puis nous donnerons quelques caractéristiques une fois traduit en modèle IF.

6.1.2.1 Représentation graphique

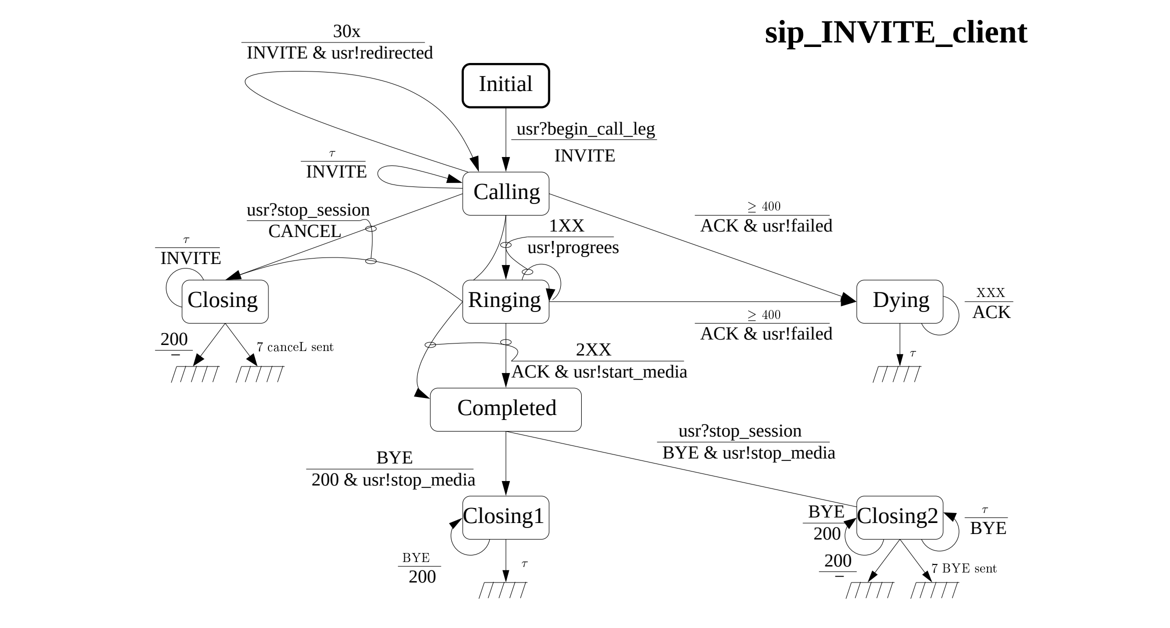

Fig. 6.2 – *Représentation de l'automate Client du protocole SIP*

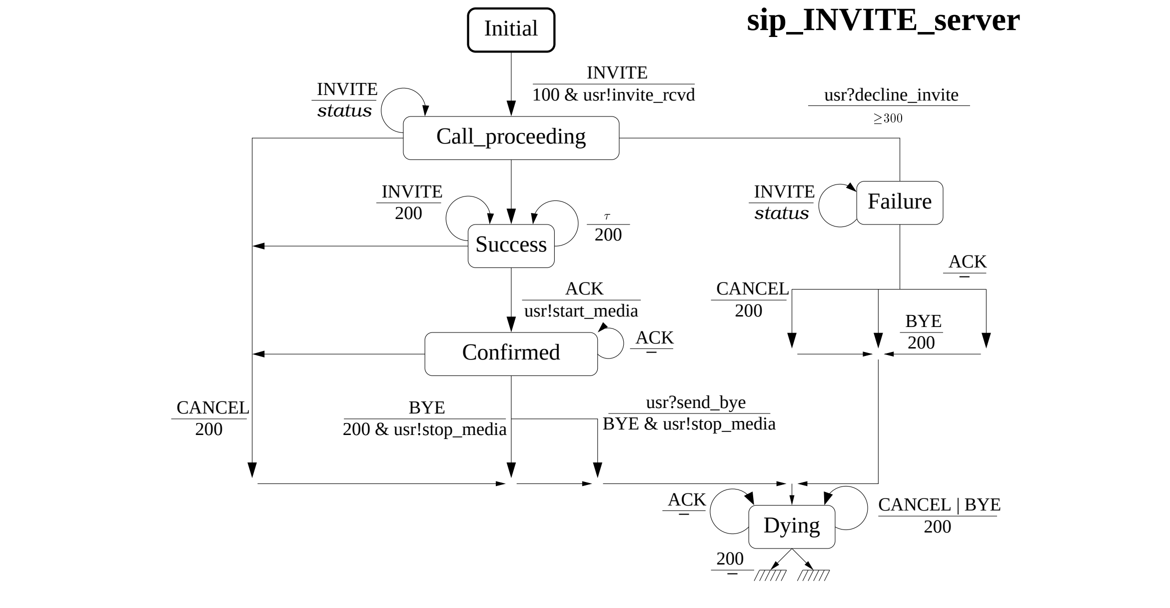

Fig. 6.3 – *Représentation de l'automate Server du protocole SIP*

6.1.2.2 Modèle IF

Nous allons donné le nombre des états et transitions que comporte les deux automates. Une présentation plus détaillée des automates du protocole SIP et des objectifs de test que nous verrons dans la section 6.1.3.1 est mise dans l'annexe B.

– **Caractéristique de la représentation en modèle IF**

1. le processus client :

```
*----------------------------------------------*
processus 0 : de nom client_i0 de symbole record
A pour nombre d'etats : 98
A pour nombre de transitions : 121
```

2. le processus serveur :

```
*----------------------------------------------*
processus 1 : de nom server_i0 de symbole record
A pour nombre d'etats : 65
A pour nombre de transitions : 80
```

6.1.3 Validation

D'abord la donnée initiale du programme est un fichier comportant les automates de la spécification et l'automate de l'objectif de test. Pour faire ce test, nous utilisons des données représentées dans le modèle IF (dont nous avons montré un exemple de code 6.1.2.2). Dans la suite de ce travail, nous allons présenter par un exemple, une composition. Pour cela un objectif de test sera écrit.

6.1.3.1 Objectif de test

Nous allons présenter deux objectifs de test :

1. Nous allons tester d'abord si une demande de communication se fait bien. Pour cela l'objectif de test contient un automate réalisant la séquence suivante qui sera terminer par un "accept" :

   $Initial \longrightarrow Calling \longrightarrow Ringing \longrightarrow accept$

   Cette automate comporte :

```
*----------------------------------------------*
processus 2 : de nom ot de symbole record
A pour nombre d'etats : 14
A pour nombre de transitions : 14
Etat : accept
```

2. Nous ferons ensuite le test ou nous ne voulons pas vérifier certaines séquences. Pour cela les séquences de l'objectif terminerons par un "reject". Nous pouvons reprendre la séquence :

   $Initial \longrightarrow Calling \longrightarrow Ringing \longrightarrow reject$

   Cette automate comporte :

```
*----------------------------------------------*
processus 2 : de nom ot de symbole record
A pour nombre d'etats : 14
A pour nombre de transitions : 14
Etat : reject
```

6.1.3.2 Résultat avant projection

Nous présentons ce que nous obtenons dans les tableaux de marquages des états.

1. D'abord pour la séquence comportant l'"accept".

```
FIN
e1:       q18 e2:    accept visite : 2  Dans Pile : 0
e1:   ringing e2:   ringing visite : 2  Dans Pile : 0
e1:       q16 e2:       q16 visite : 2  Dans Pile : 0
e1:       q15 e2:       q15 visite : 2  Dans Pile : 0
e1:       q11 e2:       q11 visite : 2  Dans Pile : 0
e1:        q8 e2:        q8 visite : 2  Dans Pile : 0
e1:   calling e2:   calling visite : 2  Dans Pile : 0
e1:        q6 e2:        q6 visite : 2  Dans Pile : 0
e1:        q5 e2:        q5 visite : 2  Dans Pile : 0
e1:        q4 e2:        q4 visite : 2  Dans Pile : 0
e1:        q3 e2:        q3 visite : 2  Dans Pile : 0
e1:   initial e2:   initial visite : 2  Dans Pile : 0
e1:        q1 e2:        q1 visite : 2  Dans Pile : 0
e1:     start e2:     start visite : 2  Dans Pile : 0
```

Le champ nommé "visite" correspond à un état faisant parti d'une séquence d'acceptation. Le couple d'acceptation est la première référence du tableau. Le chemin d'acceptation est la séquence d'état partant de "start" jusqu'à q18. Ici la séquence est complète car nous avons pris comme objectif de test une seule séquence. Dans le cas ou il y a plusieurs séquence c'est la projection qui reconstitue les séquence complète par lecture du tableau et en récupérant les transitions entrant des états. En effet nous ne stockons pas les transitions dans le tableau ce qui représente trop de place mais nous gardons l'information initiale des transitions dans les vecteurs de transition du modèles IF.

2. Puis, pour la séquence comportant le "reject".

```
FIN
e1:      start e2:      start visite : 3  Dans Pile : 0
```

Nous avons considéré que tout chemin menant à un état "reject" ne devait pas être explorer. Or comme nous n'avons dans cet exemple qu'une seule séquence la non exploration se fait dés le début au niveau de l'état "start", car les états ne comportent qu'une seule transition sortante donc toute la séquence n'est pas à explorée.

6.1.3.3 Résultat de projection

D'aprés les deux exemples de test nous obtenons une spécification après projection comportant que les états des tableaux résultats.

– Pour le premier exemple 14 états et 14 transitions, au lieu de 98 états et 121 transitions,

– Pour le deuxième exemple 1 état et 1 transition, au lieu des 65 états et 80 transitions.

6.1.4 Observation

En vu des résultats nous pouvons conclure que le produit synchrone suivit de la projection est bien un filtrage au niveau des automates. Les études montrent que les gains de place lorsque nous utilisons des automates devient indispensable, tant les automates sont importants en états et en transitions. Or les tests sont souvent sur des petites parties de ces automates, et le fait de travailler sur des automates réduits facilite leur exploitation.

Conclusion

Bilan

Ce travail se situe dans le cadre de méthodes formelles pour la conception et la validation des systèmes distribués. De nombreux travaux sur les tests sont fait à partir de systèmes de transitions étiquetées. Le problème sur les STE est leur taille importante et donc l'explosion des états lors des traitements. Notre approche et d'essayer de réduire les STE, pour cela nous passons par une représentation intermédiaire qui étant plus expressives permet une étude plus fine pour les réductions (variables,actions, ...). Il s'agit des automates communicants AC représenté à l'aide d'un modèle IF. Nous voulons après réduction et transformation en STE, que les STE obtenus, simulent les systèmes initiaux pour obtenir des tests de même ordre.

D'abord nous avons présentés les STE qui sont utilisés dans les compositions pour réaliser des tests. Ensuite, nous avons introduit la notion de test pour en comprendre le fonctionnement et présenter une génération de tests symboliques, basée sur des automates communicants AC. Un test est caractérisé par un objectif de test. Ce dernier définit informellement la fonctionnalité de la spécification qui doit être testée. Pour cela, nous sommes d'abord appuyés sur une représentation intermédiaire expressive IF pour décrire le fonctionnement des systèmes en terme d'automates communicants AC. Le modèle IF aètè choisi car il a une sémantique opérationnelle complètement définie en termes de systèmes de transitions étiquetées. Concrètement, la sémantique consiste dans un ensemble des règles pour construire, à partir d'un programme IF, l'ensemble des comportements, sous la forme d'un système de transitions étiquetées. C'est un aspect trés important pour la validation automatique basée sur des modèles, étant donné que toute technique exitante de validation s'applique habituellement à ce niveau d'abstraction.

Puis par analyse statique, nous avons proposé, une composition par produit synchrone d'automates communicants abstraits (l'abstraction portant sur les variables et les files d'attente). Cette composition a pour but de réduire la taille des automates en ne gardant que les séquences pertinentes décrites par l'objectif de test. La composition est représentée comme suit, $\|\mathrm{AC} \otimes \mathrm{OT} = \| \mathrm{AC}^{'}$. Enfin, nous avons défini, l'action de projection du résultat $\|\mathrm{AC}^{'}$ décrit par projection$(\|\mathrm{AC}^{'}) = \|\mathrm{AC}$* Cette projection, nous a permis d'obtenir à nouveaux des automates communicants concret pour la spécification.

L'utilisation de l'analyse statique est nécessaire, étant donnée la complexité des systèmes distribués. Nous avons adapter des études qui utilisaient des entrées au système. Ces études n'utilisaient pas l'ordre des actions. Or, ici, l'ordre est indispensable, ce que nous avons adapté. Pour cela nous nous sommes basés sur une composition de type produit synchrone. Nous avons pris comme entrée au système un objectif de test. Cette objectif est une description de ce que l'on veut vérifier. Représenter en séquence d'exécution le produit synchrone appliqué à une spécification à la propriété de donner comme résultat les séquences de la spécification relative à l'objectif de test. Cette analyse a l'avantage de réduire les tailles des automates communicants de la spécification, en gardant l'essentiel des séquences à tester. Limiter les tailles peut garantir un test efficace.

Nous avons conçu un algorithme pour générer les automates réduits. Nous avons, pour cette réalisation, utiliser un algorithme en profondeur, pour effectuer le parcours des automates communicants de la spécification. Le principe de cette génération est basée sur un objectif de test. L'objectif intervient pour vérifier si la spécification comporte les séquences d'actions consécutives qu'il décrit. Nous remplissons pendant ce parcours une pile, qui de par son principe, garde les états relatifs aux séquences d'actions "entières" et communes entre l'automate de la spécification et l'automate de l'objectif de test.

Perspectives

Avec des analyses supplémentaires, comme l'activité des variables, ou la propagation de contraintes, augmenteraient considérablement les performances de la validation et la génération de test. L'étude que nous avons fait réduit la taille de système de transitions étiquetées qui serviront à faire des tests. Pour l'instant les outils n'utilisent que les STE, une amélioration serait de pourvoir réaliser des tests directement avec les automates communicants.

Bibliographie

[BFG99a] Bozga (Marius), Fernandez (Jean Claude) et Ghirvu (Lucian). – State space reduction based on live varibles analysis. *In : SAS*. LNCS. – Venezia, 1999.

[BFG+99b] Bozga (Marius), Fernandez (Jean-Claude), Ghirvu (Lucian), Graf (Susanne), Krimm (Jean-Pierre) et Mounier (Laurent). – IF: An intermediate representation and validation environment for timed asynchronous systems. *In : World Congress on Formal Methods (1)*, pp. 307–327.

[BFG00] Bozga (Marius), Fernandez (Jean-Claude) et Ghirvu (Lucian). – Using static analysis to improve automatic test generation. *In : Tools and Algorithms for Construction and Analysis of Systems*, pp. 235–250.

[BFLG+99] Bozga (Marius), Fernandez (Jean-Claude), Lucian Ghirvu (Susanne Graf), Krimm (Jean-Pierre), Mounier (Laurent) et Sifakis (Joseph). – If: An intermediate representation for sdl and its applications. *In : R. Dssouli, G. Bochmann, and Y. Lahav, editors, Proceedings of SDL FORUM'99.(Montreal, Canada),Elsevier*, pp. 423–440.

[BGMS98] Bozga (Marius), Graf (Susanne), Mounier (Laurent) et Sifakis (Joseph). – The intermediate representation if. – 1998. (Internal document, don't distribute).

[BHR84] Brookes (S. D.), Hoare (C. A. R.) et Roscoe (A. W.). – A theory of communicating sequential processes. *Journal of the ACM*, vol. 31, n° 3, juillet 1984, pp. 560–599.

[Boz99] Bozga (Marius). – *Vérification symbolique pour les protocoles de communication.* – Thèse de PhD, Vérimag, 1999.

[BST97] Bornot (Sebastien), Sifakis (Joseph) et Tripakis (Stavros). – Modeling urgency in timed systems. *In : COMPOS*, pp. 103–129.

[CCI88] CCITT/SGx/WP3-1. – Specification and description langage, sdl. – CCITT Recommendation ITU Z.100, 1988.

[FEL97] France (Robert), Evans (Andy) et Lano (Kevin). – The UML as a formal modeling notation. *In : Proceedings OOPSLA'97 Workshop on Object-oriented Behavioral Semantics*, éd. par Kilov (Haim), Rumpe (Bernhard) et Simmonds (Ian). pp. 75–81. – Technische Universität München, TUM-I9737.

[FJJM92] Fernandez (Jean-Claude), Jard (Claude), Jéron (Thierry) et Mounier (Laurent). – On the fly verification of finite transition systems. *Formal Methods in System Design*, vol. 1, 1992.

[FJJV96] Fernandez (Jean-Claude), Jard (Claude), Jeron (Thierry) et Viho (Cesar). – Using on-the-fly verification techniques for the generation of test suites. *In : Computer Aided Verification*, pp. 348–359.

[God96] Godefroid (Patrice). – *Partial-order methods for the verification of concurrent systems: an approach to the state-explosion problem.* – New York, NY, USA, Springer-Verlag Inc., 1996volume 1032, 142p.

[Gou97] Gouranton (Valérie). – *Dérivation d'analyseurs dynamiques et statiques à partir de spécifications opérationnelles.* – Thèse de PhD, Université de Rennes, 1997.

[HSS] Handley (M.), Schulzrinne (H.) et Schooler (E.). – SIP: Session initiation protocol, Internet Draft, Internet Engineering Task Force, July 1997. Work in progress.

[IT92] ITU-T. – ITU-T: Specification and Description Language (SDL), ITU-T Recommendation Z.100, International Telecommunication Union, Geneva, 1992.

[JL97] J.P. Krimm et L. Mounier. – Compositional state space generation from Lotos programs. *In : Tools and Algorithms for the Construction and Analysis of Systems*, éd. par E. Brinksma. pp. 239–258. – Enschede, The Netherlands, 1997.

[JM99] Jeron (Thierry) et Morel (Pierre). – Test generation derived from model-checking. *In : Computer Aided Verification*, pp. 108–121.

[Loi94] Loiseaux (Claire). – Vérification symbolique de programmes réactifs à l'aide d'abstractions, 1994.

[Mor00] Morel (Pierre). – *Une algorithmique efficace pour la génération automatique de tests de conformité.* – Thèse de PhD, UFR IFSIC/ laboratoire IRISA, février 2000.

[Org93] Organization (I.). – Open systems interconnection - osi conformance testing methodology and framework, international standards organization. open systems interconnection - osi conformance testing methodology and framework, part 3: Ttcn extensions,iso/iec jtc 1 dam-1, 1993.

[Pha94] Phalippou (Marc). – *Relations d'Implantation et Hypothèses de Test sur les Automates à Entrées et Sorties.* – Thèse de PhD, Université de Bordeaux, 1994.

[Tip95] Tip (Franck). – A survey of program slicing techniques. *Journal of programming languages*, vol. 3, 1995, pp. 121–189.

[Tre92] Tretmans (J.). – *A Formal Approach to Conformance Testing.* – Thèse de PhD, University of Twente, Enschede, Th Netherlands, 1992.

[Wei79] Weiser (M.). – *Program slices: formal, psychological, and practical investigations of an automatic program abstraction method.* – Thèse de PhD, University of Michigan, Ann Arbor, 1979.

Annexe A

Présentation des alogrithmes

A.1 Fonctions et actions utilisées

Rappel A.1 (Empiler des éléments :) *La première étape est de placer dans le bas de pile les états initiaux de chaque automate. Ces états peuvent comporter des transition permettant d'accéder en simultané à deux autres états de la spécification et de l'objectif de test. Ces transitions sont alors placer en liste au niveau du couple d'état. Le couple des états cibles est placé au dessus du couple des états sources et le couple de transition permettant cette accessibilité est supprimer de la liste des transitions. La construction de cet pile donne parmi le produit synchrone des chemins soient d'acceptation, soient de rejet, soient des chemins non vérifiable par le produit.*

Remarque A.1 *Deux transitions sont égales, si leur signal sont de même expréssion (ici de même nom).*

Rappel A.2 (Dépiler des éléments :) *Enlève de la pile le couple d'état de haut de pile met ou pas dans le tableau de récupération des chemins l'information de ce couple. La procédure de "depiler" se fait jusqu'à obtenir un couple d'état qui comporte encore une liste de couple de transition non vide , ou jusqu'au bas de la pile. A partir des transitions non vides, on recommence un nouveau traitement "empiler", le bas de pile arrêt le traitement.*

Description des différentes fonctions utilisées :

récupeEtatInit(*system : system*) $\rightarrow$ un état ei : Un système IF au niveau d'un processus.

ef : Rend le premier état du processus s'il exite.
Rend NULL si pas d'état.

état = system→GetState()→GetAt(0);
return(état);

récupeEtatCible(*IfTransition: t1 IfTransition: t2*) → un état

Récupération des états cibles en utilisant les transitions données. Ici utilisation de la définition du produit synchrone.

ei : Une transition donnée t

ef : Rend un état cible s'il existe par rapport à la transition t.
Rend NULL si pas d'état.

état = t →GetTarget();
return (état);

Description des différentes procédures utilisées :

empiler (*IfState : état1, IfState état2*)

Met en haut de pile les états *état1 et état2* ainsi que tous les couples des transitions sortantes de ces états. L'ensemble des couples constituent T

depiler (*IfState : état1, IfState état2*)

Enlève de la pile le couple met ou pas dans le tableau de recuperation des chemins l'information de ce couple. La procedure de "depiler" se fait jusqu'a obtenir un couple d'état qui comporte une liste de couple de transition non vide, ou jusqu'au bas depile. Les transitions non vide recommence un nouveau traitement, le bas de pile arrêt le traitement.

supprimer (*elementTransPile : Ce*)

Enléve de la liste des couples de transition le couple Ce

A.2 Algorithme

```
// De la CLASSE execution dans projection.h
// Methode exécutant le travail
// Produit synchrone expilcite.
// Recherche de chemins Accept reject ou rien
// Mise en forme d'une table d'acceptation
void execution::creer()
{
 transition t(system);

 int indiceOt = system->GetProcesses()->GetCount()-1;
 int indiceOt = system->GetProcesses()->GetCount()-1;
 IfProcess* processSpec = system->GetProcesses()->GetAt(0);
 IfProcess* processOt = system->GetProcesses()->GetAt(indiceOt);
 IfState *spec = processSpec->GetStates()->GetAt(0);
 IfState *specOt = processOt->GetStates()->GetAt(0);

 elementPile* tPile;

 elementPile *celluleEtat = NULL;
 celluleEtat = new elementPile(spec,specOt);
 // Donne q et ses transitions sortantes pour sucesseur.
 pile.ajouterElementEtatPile(celluleEtat);

 elementTableau* eT = NULL;
 eT = new elementTableau(spec,specOt,2,0);
 dansTableau = tableauPile.dansPile(eT);
 tableauPile.ajouterElementDansTab(eT);
 t.recupererTransitionPourPile(spec,specOt);

 IfTransition* t1;
 IfTransition* t2;
 IfState *s;
 IfState *sOt;

 // Récupération des états.
 tPile = pile.donnerElementPile(&tableauVisite,&tableauPile);
```

```
 while (pile.basPile!=NULL)
  {
   if(tPile!=NULL)
    {
     t1 = tPile->trans1;
     t2 = tPile->trans2;
     spec = t1->GetTarget();
     specOt = t2->GetTarget();
     celluleEtat = new elementPile(spec,specOt);

     elementTableau* eT = NULL;
     eT = new elementTableau(spec,specOt,1,1);

     dansPile = tableauPile.dansPile(eT);
     dansTableau = tableauVisite.visiteDeTableau(eT);

     if (dansPile==0 && dansTableau==0)
      {
       tableauPile.ajouterElementDansTab(eT);
       pile.ajouterElementEtatPile(celluleEtat);
       t.recupererTransitionPourPile(spec,specOt);
      }
      if (dansPile==1)
       tPile = pile.donnerElementPile(&tableauVisite,&tableauPile);
    }
    else
      tPile = pile.donnerElementPile(&tableauVisite,&tableauPile);
  }
 printf("FIN\n");
 tableauVisite.afficherTableau();
}
```

Annexe B

Automates utilisés

Nous présentons de façon plus précise comment est composé un automate en modèle IF

1. D'abord le protocole est constitué d'un système comportant les automates communicants, ou les variables globales, les signaux, les files d'attente exérieures, sont définies pour l'emsemble des automates.

```
system sip;

type
  Character = range 1 .. 256;

signal
  invite_rcvd;
  decline_invite;
  accept_invite;
  send_bye;
  ...

buffer
  env : queue :toenv of * ;
  q_client_i0 : queue of
    code,
    bye,
  ...
```

2. Puis les processus de la spécification sont définis un par un :
   - d'abord le processus client :

```
process client_i0 :buffer q_client_i0;
var
  sender : pid;
  ...

state
start :init;
q1 :nostable;
initial
  discard
    code,
    bye,
    stop_session in q_client_i0;
  end
...

transition
from start
  to q1;
from q1
  to initial;
from initial
  input begin_call_leg from q_client_i0
  to q3;
  ...
```

   - Puis le processus server :

```
process server_i0 :buffer q_server_i0;
var
  sender : pid;
  parent : pid;
  offspring : pid;
  t : timer;
  n : int;
  dying_time(4) : int;
  ...

state
start :init;
q1 :nostable;
initial
  discard
    send_bye,
    decline_invite,
    accept_invite,
    twohund,
    ack,
    bye,
    cancel in q_server_i0;
  end;
q3 :nostable;
q4 :nostable;
q5 :nostable;
call_proceeding
  discard
    send_bye,
    twohund,
    ack,
    bye in q_server_i0;
  end;
q7 :nostable;
...
```

```
transition
from start
  to q1;
from q1
  to initial;
from initial
  input invite from q_server_i0
  to q3;
from q3
  do {
    output code(1, 0, 0) to q_client_i0
  }
  to q4;
from q4
  do {
    output invite_rcvd to env
  }
  to q5;
from q5
  to call_proceeding;
from call_proceeding
  input invite from q_server_i0
  to q7;
from q7
  to call_proceeding;
from call_proceeding
  input accept_invite from q_server_i0
  to q8;
...
```