Génération automatique de tests de conformité pour les protocoles de télécommunication

Constantin Lucian Ghirvu

UNIVERSITÉ GRENOBLE I - JOSEPH FOURIER
U.F.R. D'INFORMATIQUE ET DE MATHÉMATIQUES APPLIQUÉES

N° attribué par la bibliothèque

T H È S E

pour obtenir le grade de
DOCTEUR DE L'UNIVERSITÉ GRENOBLE I

Discipline : **Informatique**

présentée et soutenue publiquement
par

Constantin Lucian GHIRVU

le 12 juillet 2002

Titre :
GÉNÉRATION AUTOMATIQUE DE TESTS DE CONFORMITÉ POUR LES PROTOCOLES DE TÉLÉCOMMUNICATION

Directeur de thèse :
M. Jean-Claude FERNANDEZ

JURY

M. Jacques VOIRON	,Président
M. Richard CASTANET	,Rapporteur
M. Claude JARD	,Rapporteur
M. Jean-Claude FERNANDEZ	,Directeur de thèse
M. Joseph SIFAKIS	,Examinateur

Remerciements

Je tiens à remercier

Monsieur Jacques Voiron, Professeur à l'Université Grenoble I, pour m'avoir fait l'honneur de présider le jury de cette thèse

Messieurs Richard Castanet, Professeur à l'École Nationale Supérieure d'Électronique, Informatique et Radiocommunications de Bordeaux, et Claude Jard, Directeur de recherches CNRS à l'Institut de Recherche en Informatique et Systèmes Aléatoires de Rennes, de m'avoir fait l'honneur d'accepter de juger ce travail

Monsieur Joseph Sifakis, Directeur de recherches CNRS à VERIMAG, Grenoble, de m'avoir accueilli dans son laboratoire, de m'avoir intégré dans ses projets et de m'avoir fait l'honneur d'accepter de participer au jury de cette thèse

Monsieur Jean-Claude Fernandez, Professeur à l'Université Grenoble I, pour avoir accepté d'encadrer cette thèse, pour le temps lui consacré, pour ses conseils et suggestions qui ont permis la réalisation de cette thèse.

Je tiens à remercier la Région Rhône-Alpes pour le soutien financier accordé à cette thèse.

Je remercie Alain Kerbrat, Chef de projet à Telelogic Technologies Toulouse, d'avoir accepté de co-encadrer cette thèse au début et ensuite de la soutenir de la part des sociétés VERILOG et Telelogic Technologies Toulouse.

Je remercie tous les membres du laboratoire VERIMAG pour le cadre de travail offert. En particulier, je remercie les membres de l'équipe asynchrone : Saddek Bensalem, Susanne Graf, Yassine Lakhnech, Laurent Mounier, Michaël Perin, Anahita Akhavan, Moussa Amrani, Cyril Pachon, Romain Janvier, les anciens membres de l'équipe : Aurore Annichini, Guoping Jia et Jean-Pierre Krimm, ceux qui m'ont beaucoup aidé à la fin : Manuel Aguilar, Ileana et Iulian Ober et un grand merci à Liana et Dorel Bozga (pour tout).

Enfin, je remercie tous mes professeurs et ma famille (pour le soutien lointain).

Table des matières

Table des figures

Liste des tableaux

Chapitre 1

Introduction

1.1 Thématique

Le test est une phase fondamentale du cycle de vie de tout produit industriel. On l'utilise pour s'assurer que la réalisation d'un produit respecte ses spécifications initiales. Il permet de détecter des erreurs d'un système par l'expérimentation. Ce n'est pas un processus exhaustif, il indique la présence des erreurs, pas leur absence.

Le test est une des principales techniques de validation utilisées dans l'industrie (y compris celle des logiciels ([Bei90, XRK99])). La croissance et la diversité de la production des logiciels entraîne aussi un augmentation de l'importance du test (le logiciel WINDOWS NT 5.0 contiendra plus de 48 million lignes de codes dont 7,5 million lignes de codes pour le test cf. [Neu99]).

Pour tester les logiciels on emploie une palette variée de techniques ([Bei90]) : on teste leurs graphes de flot, les chemins, le flot des données, les domaines d'entrée, la syntaxe.

Dans le cadre de notre thèse nous nous intéressons au problème du test de conformité pour les protocoles de télécommunication. Les communications entre les sous-systèmes composants (par exemple, les ordinateurs) des systèmes distribués (y compris les systèmes de télécommunication) sont soumises à des règles décrites par des *protocoles*. Les spécifications des protocoles sont décrites en utilisant des formalismes de niveau supérieur et sont, dans la plupart des cas, standardisées. Les protocoles sont ensuite implémentés à partir de ces standards. Il est nécessaire donc que les *implémentations* des protocoles aient un comportement *conforme* à leurs *spécifications*.

Le *test de conformité des protocoles* est une méthode qui accomplit ce but. Il teste l'implémentation d'une entité de protocole vis-à-vis de sa spécification.

Les organismes ISO[1] et CCITT[2] ont établi une *méthodologie pour le test de conformité* décrite par le standard ISO IS-9646 ([fS92b]). Cette méthodologie voit le test de conformité comme un test *boîte noire*. Cela signifie que (comme c'est le cas dans la pratique) la structure interne de l'implémentation est inaccessible au testeur. Celui peut tester seulement le comportement observable de l'implémentation. Il le fait en utilisant des implémentations des cas de test abstraits obtenus à partir de la spécification. À chaque exécution d'un cas de test sur l'implémentation on attribue un verdict.

L'état actuel d'application dans l'industrie de cette méthodologie consiste dans l'écriture manuelle des séquences de test et l'utilisation de techniques ad-hoc d'application de ces tests avec des résultats mal maîtrisés. Ces pratiques conduisent à un coût de l'effort de test important. La production de tests en grand nombre, avec des garanties de pertinence et de correction, impose l'utilisation de méthodes formelles de génération ([Tre92, Pha94a]).

1.2 Contexte

Notre équipe au laboratoire VERIMAG et LSR/IMAG s'intéresse à la validation et au test des protocoles et des systèmes asynchrones. Depuis plus de dix ans un ensemble d'outils a été développé dans le but de faciliter la mise au point de spécification formelle, et de produire automatiquement des séquences de test. Cet ensemble d'outils est bien diffusé dans le monde académique et chez certains industriels.

La boîte à outils CADP[3] ([FGK+96]) est développée par laboratoire VERIMAG en coopération avec l'équipe VASY de l'INRIA Rhône-Alpes. Elle est composée par plusieurs outils et environnements ; nous en mentionnons quelques-uns :

- ALDÉBARAN ([BFKM98]) est un outil pour effectuer des réductions et comparaisons des graphes en prenant en compte certaines relations d'équivalence et préordres.

- OPEN-CAESAR ([Gar98]) est un environnement ouvert pour construire des prototypes des algorithmes de vérification fondés sur l'exploration exhaustive du modèle. Il offre aussi des représentations implicites (pour les algorithmes de vérification à la volée ([FJJM92])) ou symboliques (fondées sur des diagrammes binaires de décision).

1. International Organization for Standardization
2. Consultative Committee on International Telegraphy and Telephony
3. CAESAR-ALDÉBARAN Distribution Package

Notre équipe développe maintenant IF ([BFG+99b, BFG+99c, BFG+00c]), un environnement de validation pour les systèmes asynchrones temporisés. Cet environnement est fondé sur une philosophie d'intégration des méthodes formelles dans la pratique industrielle de vérification et de test ([Sif99]). Ses principales directions de travail sont : le développement d'une version temporisée du langage SDL (en incluant aussi la création dynamique des instances de processus) qui comprend aussi des aspects similaires à UML[4] (une structure d'états similaire aux STATECHARTS [Har87]) et le traitement de la complexité (en utilisant techniques d'analyse statique [BFG99a, BFG00a, BFG00b], analyse symbolique et abstractions [BL99], génération composante par composante[5] [KM00]). Les principaux atouts de IF sont : un environnement de validation ouvert, permettant d'un côté la connexion avec plusieurs outils de vérification académiques (KRONOS [Yov97], SPIN [Hol91], INVEST [BLO98]) mais aussi l'accès aux plusieurs niveaux de représentation des programmes.

Le laboratoire VERIMAG travaille avec l'équipe PAMPA de l'IRISA sur les fondements théoriques du test de conformité, la formalisation des concepts et la production d'algorithmes et d'outils correspondant aux idées mises en avant dans ces travaux.

Un résultat de cette coopération est TGV[6] [FJJV97, FJJV96, JM97, JM99], un outil pour la génération automatique de tests de conformité. Il illustre l'application de méthodes formelles dans le domaine du test tout en essayant de modéliser le plus possible la pratique industrielle courante dans la génération automatique de tests de conformité. Le principe de TGV consiste à synthétiser à partir d'une spécification formelle d'un protocole et d'un objectif de test (une propriété qu'on désire à vérifier sur une implémentation) un cas de test qui pourra s'appliquer sur l'implémentation. Ce cas de test est décoré par des verdicts qui seront utilisés lors de l'application du cas de test à l'implémentation, pour décider si l'implémentation a réagi conformément à sa spécification formelle. Grâce a une relation de conformité définie formellement, l'algorithme de TGV construit des cas de test valides (ils ne rejettent pas une implémentation conforme à sa spécification). TGV a deux modes distincts de fonctionnement :

- le mode *explicite*, à partir d'une spécification décrite en format ALDÉBARAN,
- le mode *à la volée*, à partir des spécifications formelles (SDL ou LOTOS).

L'objectif de test est décrit en format ALDÉBARAN. Le format de sortie de TGV est le format ALDÉBARAN. Le graphe de test obtenu est déplié sous la forme d'un arbre en

4. [Gro99]
5. *Compositional Generation* en anglais
6. *Test Generation using Verification technology*

format TTCN[7] ([fS92a]), standard de fait dans le monde de test.

TGV a été réalisé au dessus de l'environnement CADP.

L'outil TGV a été utilisé pour des études de cas d'origine industrielle :

- l'étude de cas DREX effectué pour le compte de la DGA, CAP SESA et FRANCE-TÉLÉCOM,
- l'étude de cas SSCOP ([BFG+00d]) fournie par le CNET[8], dans le cadre de l'opération 1 du projet FORMA, projet coordonné par le laboratoire VERIMAG.
- il a été utilisé aussi pour générer des séquences exécutables de tests pour un protocole de cohérence de l'antémémoire[9] ([KVZ98]).

TGV utilise aussi *Object*GEODE ([Tel]) de TTT. *Object*GEODE est un environnement de développement dédié au langage SDL, commercialisé par la compagnie TTT . Il supporte SDL-92, l'utilisation de ASN.1 (norme ITU-T Z-105), le langage MSC (Message Sequence Charts) pour la description des échanges de signaux et la méthode UML ([Gro99]) pour une description orientée objet. Il fournit d'éditeurs graphiques et compilateurs pour chacun de ces langages et un générateur de code C. *Object*GEODE contient un simulateur (pour la mise au point) et un vérificateur basé sur des méthodes de model-checking ([QS82, CES83]). *Object*GEODE a été connecté aux outils d'analyse, de vérification et de génération de test issus de VERIMAG et de l'IRISA.

Une intégration des algorithmes dérivés de TGV, dans l'outil de test TESTCOMPOSER ([KJG99]) de TTT , a été financée par FRANCE-TÉLÉCOM.

1.3 Contribution

La génération automatique des cas de test basée sur l'approche modèle est limitée par le problème d'explosion d'état (les modèles ont une taille exponentielle relativement aux spécifications).

Rappelons-nous que la génération automatique des cas de test est guidée par des objectifs de test (conçus par des experts). Si un ou plusieurs objectifs de test existent alors le problème d'explosion d'état pourrait être contourné (en utilisant par exemple des techniques à la volée [FJJM92]) qui construisent seulement la partie explorée du modèle. Les techniques à la volée sont implémentées dans TGV ([Mor00]).

7. Tree and Tabular Combined Notation
8. Centre de Recherche et Développement de FRANCE TÉLÉCOM
9. *cache memory* en anglais

Mais la conception des objectifs de test pour un modèle d'une spécification est difficile sans le construire explicitement (et cela parce que les paramètres de signaux dans l'objectif de test sont traités en énumération).

Dans notre thèse nous avons essayé de pallier ce problème et par conséquence nous nous avons établi deux objectifs :

- le traitement efficace des données en vue de l'extension de l'outil TGV (tel qu'il existait dans [FJJV97, FJJV96]),
- l'étude d'un langage d'expression pour les objectifs de test.

Ces deux objectifs ont été accomplis par le développement d'une méthodologie de test basée sur des techniques[10] issues des domaines de la vérification et de l'analyse des programmes (analyse statique[11]). Cette méthodologie, décrite dans la figure 1.1[p.18] propose un ensemble de procédures qui ont le but de simplifier la spécification en tenant compte de sa structure ou de la structure des objectifs de test et cela avant de la génération des cas de test. On réduit ainsi la taille des systèmes de transitions étiquetées (STE) qui modélisent les comportements observables de la spécification. Les analyses de programme appliquées sont l'analyse d'activité des variables, l'analyse de l'utilité (du contrôle et des données) et l'analyse d'invariance. Ces analyses, utilisées surtout dans le domaine de la compilation, ont été étendues à notre représentation intermédiaire (qui est un ensemble d'automates étendus communicants par files d'attente).

Il faut mentionner que les simplifications ci-dessus sont conservatrices dans le sens suivant : les résultats obtenus en appliquant TGV, pour deux spécifications (la spécification originale et une variante simplifiée de celle-ci) et pour un même objectif de test, sont identiques.

Traitement efficace des données

Analyse d'activité

Cette analyse[12] calcule, pour chaque état de contrôle q, l'ensemble des variables qui seront utilisées[13] dans un chemin partant de cet état et sans que leurs valeurs dans l'état q soient rendues inutiles[14] avant leur première lecture. Cette analyse peut être

10. D'ailleurs ces techniques sont aussi applicables dans le domaine de la vérification par modèles : on peut simplifier les spécifications en fonction de formules de logique temporelle
11. *compile-time analysis* en anglais
12. qui est une des plus anciennes analyses statique [Hec77]
13. leurs valeurs seront lues
14. par une instruction qui écrit des valeurs dans ces variables

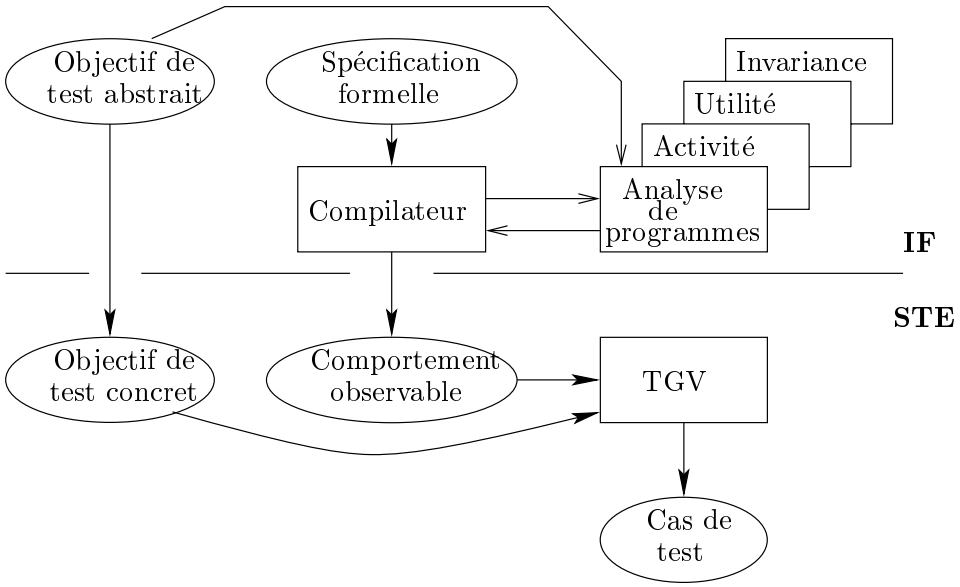

FIG. 1.1 – *Méthodologie de test*

vue comme un cas extrême de l'analyse d'utilité des données (dans ce cas il n'y a pas de critère de tranchage).

Analyse d'utilité

On couvre ici deux aspects : l'utilité du code et celle des données. Par le terme *utilité* on comprend l'utilité par rapport à l'objectif de test et l'ensemble d'entrée contrôlables. L'ensemble d'entrée contrôlables comprend tous les signaux (et les contraintes maximales) que le testeur a l'intention d'envoyer vers l'IUT pendant le test. L'analyse d'utilité du contrôle élimine les transitions de la spécification dont les déclencheurs[15] n'appartiennent pas à l'ensemble d'entrée contrôlables. Ensuite on élimine récursivement toutes les transitions dont l'état source n'a pas de transitions entrantes. L'analyse d'utilité des données est faite en utilisant la technique de tranchage[16]. On analyse l'utilité des variables mais aussi celle des paramètres de sortie/entrée. Intuitivement, une variable est utile dans un état si sa valeur, par transitivité, est utile dans un certain endroit de la spécification[17] (choisi par l'utilisateur).

15. *triggers* en anglais
16. *slicing* en anglais
17. précisé dans le critère de tranchage

La technique de tranchage est une technique fortement utilisé dans diverses domaines (pour déboguer les programmes, par exemple). Le tranchage dans le contexte des programmes concurrents a été utilisé pour la première fois par [Che93], puis pour des processus légers[18] par [Kri98] et pour PROMELA ([Hol97]) par [MT98]. Dans [DHZ99] il y a une étude du tranchage des programmes séquentiels en ce qui concerne une formule de LTL et dans [HCD+99] le tranchage est utilisé pour vérifier et extraire des machines à état fini à partir des programmes JAVA. Dans [CFR+99] il y a une application du tranchage dans la vérification des programmes VHDL.

Analyse des données

Une analyse des données est exprimée par un problème de flot de données ([MR90]). Ceci est un tuple constitué par un treillis (complet), un espace de fonctions (au moins monotones), un graphe de flot et une fonction qui attache à chaque transition du graphe une fonction de transfert (appartenant à l'espace de fonctions). On obtient ainsi un opérateur de flot dont le plus petit (grand) point fixe est la solution du problème de flot. Pour trouver les solutions des problèmes de flot nous avons utilisé des stratégies d'itération existantes dans la littérature ([Hec77, CC92b, Bou93, Muc97, NNH99]). Ces stratégies comportent deux étapes : *l'initialisation* d'une (ou plusieurs) liste(s) de travail et *l'itération* sur ces listes. Dans tous les cas qu'on traite, les listes sont organisées comme des files. Dans le cas d'un treillis de la hauteur finie on utilise soit une seule file, initialisée dans le préordre ou postordre inverse, soit plusieurs files correspondant aux composantes fortement connexes maximales du graphe de flot. Quand le treillis est de la hauteur non-finie il faut détecter un ensemble admissible de points d'élargissement ([Bou93]) pour nous trouver dans le cadre de l'interprétation abstraite ([CC92b]) et pouvoir ainsi calculer une approximation de la valeur des pointes fixes.

Étant donné ce cadre formel on peut obtenir les différentes analyses de données par instanciation de ce canevas : le treillis, l'espace des fonctions et les fonctions de transfert. Ceci est fait dans le chapitre 5[p.93].

Ensuite il faut prouver que les optimisations qu'on a fait obéissent à un certain critère de correction. Ceci est un résultat de bisimulation.

Langage d'expression pour les objectifs de test

Nous avons étendu aussi le concept de l'objectif de test. Les objectifs de test abstraits ont des contraintes symboliques attachées aux paramètres des signaux d'entrée. À partir

18. *threaded programs* en anglais

des contraintes d'entrée les procédures mentionnées ci-dessus calculent des contraintes pour les paramètres des signaux de sortie des objectifs de test abstraits. Ensuite, des objectifs de test concrets pourront être dérivés et en utilisant l'outil de test existant TGV on pourrait générer des tests de conformité.

1.4 Plan

Préliminaires

Dans ce chapitre on rappelle quelques notations (concernant les ensembles, les relations, les algorithmes) et notions théoriques utilisées dans les chapitres suivants (ensembles partiellement ordonnés, treillis complets, points fixes, opérateurs d'élargissement et de rétrécissement).

Validation basée sur les modèles

Dans ce chapitre on présente les principaux modèles (systèmes de transitions étiquetées et automates étendus communicants) et les approches dans la validation des programmes : la vérification et le test (de conformité). On décrit ces deux approches en s'appuyant sur une description commune ([Fer96]). Cette description s'appuie sur le fait qu'on peut voir le test de conformité et la vérification comme une relation `conf` entre l'implémentation et la spécification (ou entre une spécification et une de ses versions plus raffinée).

Verification

Dans le cadre de la vérification la relation `conf` est fréquemment une relation de bisimulation et on mentionne plusieurs approches pour vérifier si IMP `conf` SPEC.

Test de conformité

Dans le cadre du test de conformité la relation `conf` est une relation de conformité. On décrit la place du test dans le domaine v & v ([Hol91]), le problème du test de conformité, son description en utilisant la théorie des automates ([LY96]), le cadre formel de test de conformité ([Tre92, Pha94a]). Ensuite on présente l'approche fondée sur la vérification - algorithme de TGV, suivie par une présentation du test de conformité symbolique (les principaux approches).

Analyse du contrôle

On propose une analyse (statique) du contrôle qui, prenant en compte un objectif de test abstrait et un ensemble d'entrées contrôlables (éventuellement calculé à partir d'un objectif de test abstrait), fournit une spécification (éventuellement) simplifiée dont le modèle est bisimilaire avec celui de la spécification initiale.

Analyse des données

On introduit d'abord le cadre formel pour effectuer les analyses de données. Il s'agit de problèmes de flot de données. La plupart des analyses qu'on fournit fait partie du domaine de la compilation (la propagation des constantes, l'analyse des variables actives) ou du débogage des programmes. Leurs résultats servent à la simplification des spécifications, tout en préservant des modèles bisimilaires avec ceux des spécifications initiales.

Mise en œuvre

On décrit l'environnement de validation IF, son architecture, ses composantes et la représentation intermédiaire proposée. Ensuite on fournit une description de la manière dont on traduit dans IF les principaux aspects du langage de description SDL.

Ensuite on présente la bibliothèque d'analyse statique implémentée. Il s'agit de l'implémentation, dans le cadre d'une bibliothèque d'analyse statique pour IF, des algorithmes présentés dans les chapitres 4[p.85] et 5[p.93].

Enfin on présente les résultats obtenus par leur application sur des protocoles de taille industrielle : SSCOP (un protocole mono-processus) et MASCARA (un protocole multi-processus). Le séquentiel de vol d'ARIANE 5 servira comme exemple de l'application de la propagation de constantes pour des variable temporisées.

Chapitre 2

Préliminaires

2.1 Notations

On utilise les notations suivantes :

1. Comme notations *générales* : $a \doteqdot b$ signifie a est une notation pour b,

   a .. b $\doteqdot \{a, a+1, \ldots b\}$,

2. Pour les *ensembles* : **N** est l'ensemble des entiers naturels, **Z** est l'ensemble des entiers et $\emptyset$ est l'ensemble vide. L'union est notée $A \cup B$, l'intersection $A \cap B$, la différence ensembliste $A \setminus B$, le produit cartésien $A \times B$ où $\prod_{i \in I} A_i$, avec $\prod_{i \in I} A_i \ni a = (a[1], \ldots, a[i], \ldots) = (a_i)_{i \in I}$. L'inclusion des ensembles est notée $A \subseteq B$ et l'appartenance à un ensemble $x \in A$.

   On note 2^A l'ensemble des parties de A et $|A|$ le cardinal de l'ensemble A.

   Si A est un ensemble alors ϵ ($\epsilon \notin A$) est le mot de longueur 0, A^n est l'ensemble des mots sur A de longueur n (ensemble isomorphe avec $\overbrace{A \times A \times \ldots \times A}^{n \text{ fois}}$), A^+ est l'ensemble de mots finis sur A, de longueur strictement positive, A^* est l'ensemble de mots finis sur A et A^ω est l'ensemble de mots infinis sur A.

   Si $a, b \in \prod_{i \in I} A_i$ et $K \subseteq I$ alors $a = b \ mod \ K \doteqdot (\forall k)(k \in K \Rightarrow a[k] = b[k])$. On utilise aussi la notation $a^n \doteqdot \overbrace{a \cdot a \cdot \ldots \cdot a}^{n \text{ fois}}$.

3. Une *relation binaire* R entre les ensembles S et T est $R \subseteq S \times T$.

Une relation $R \subseteq S \times S$ est *réflexive* si $(\forall s)(s \in S \Rightarrow (s,s) \in R)$, *symétrique* si $(\forall s)(\forall t)((s,t) \in R \Rightarrow (t,s) \in R)$, *anti-symétrique* si $(\forall s)(\forall t)((s,t) \in R \wedge (t,s) \in R \Rightarrow s = t)$ et *transitive* si

$(\forall s_1)(\forall s_2)(\forall s_3)((s_1,s_2) \in R \wedge (s_2,s_3) \in R \Rightarrow (s_1,s_3) \in R)$.

Une relation $R \subseteq S \times S$ est un *préordre* si elle est réflexive et transitive, un *ordre partiel* si elle est réflexive, anti-symétrique et transitive, un *ordre total* si c'est un ordre partiel et $(\forall s)(\forall t)((s,t) \in S \times S \Rightarrow ((s,t) \in R \vee (t,s) \in R))$, une *équivalence* si elle est réflexive, symétrique et transitive.

Soient $R_1 \subseteq X \times Y$ et $R_2 \subseteq Y \times Z$ deux relations. La *relation composée* de R_1 et R_2 est $R_1 \circ R_2 = \{(x,z) \mid x \in X \wedge z \in Z \wedge (\exists y \in Y)((x,y) \in R_1 \wedge (y,z) \in R_2)\}$.

La relation *inverse* de $R \subseteq S \times T$ est $R^{-1} \subseteq T \times S$ t.q. $(x,y) \in R \iff (y,x) \in R^{-1}$.

La relation *identité* de S est $1_S = \{(s,s) \mid s \in S\}$.

On utilise la notation $\bar{k}$ $(k \in S)$ pour la relation incluse dans $S \times S$: $\{(s,k) \mid s \in S\}$.

4. Soient S, T deux ensembles. Une relation $R \subseteq S \times T$ est une *fonction partielle de S vers T* si $(\forall s)(\forall t_1)(\forall t_2)((s,t_1) \in R \wedge (s,t_2) \in R \Rightarrow t_1 = t_2)$ et une *fonction (totale) de S à T* si elle est une fonction partielle et $(\forall s)(s \in S \Rightarrow (\exists t)(t \in T \wedge (s,t) \in R))$.

   Par $S \to T$ où T^S on dénote l'ensemble de toutes les fonctions totales de S à T.

   Par $[v/x]e$ on dénote la substitution de la variable x par la valeur v dans l'expression e. La même notation est utilisée pour les fonctions : soient $f : S \to T$, $s \in S$ et $t \in T$. La fonction $[t/s]f : S \to T$ est définie de la manière suivante : $[t/s]f(x) = f(x)$ si $x \neq s$ et t sinon. On utilise la même notation pour les produits cartésiens : si $\theta \in \prod_{i \in I} A_i$, $k \in I$ et $a \in A_k$ alors $[a/k]\theta \in \prod_{i \in I} A_i$ et $[a/k]\theta[j] = \theta[j]$ si $j \neq k$ et a sinon.

   Si $f : S \to T$ est une fonction de S à T et $X \subseteq S$ alors $f(X) \doteqdot \{f(x) \mid x \in X\}$. On applique la même notation dans les cas où le domaine ou l'ensemble des valeurs de f sont des produits cartésiens.

2.1.1 Algorithmes

Les algorithmes de chapitres suivants sont présentés dans une notation informelle similaire à la notation ICAN ([Muc97]) ou aux langages C et PASCAL :

1. Chaque instruction est suivie par un séparateur (;).

2. L'instruction d'affectation a la syntaxe $x := exp$, l'instruction de branchement est similaire à l'instruction `if` de C :

`if` $(exp - bool)$
  $inst$;
[`endif`]
	`if` $(exp - bool)$
	  $inst$;
	  ⋮
	`else`
	  $inst$;
	  ⋮
	`endif`
		`if` $(exp - bool)$
		  $inst$;
		  ⋮
		`else if` $(exp - bool)$
		  $inst$;
		  ⋮
		`else`
		  $inst$;
		  ⋮
		`endif`

3. Les instructions d'itération sont aussi similaires aux langages C, PASCAL :

`foreach` $(\cdots)$
  $inst$;
[`endfor`]
	`while` $(exp - bool)$
	  $inst$;
	[`endwhile`]	

`foreach` $(\cdots)$
  $inst$;
  ⋮
`endfor`
	`while` $(exp - bool)$
	  $inst$;
	  ⋮
	`endwhile`
		`repeat`
		  $inst$;
		  ⋮
		`until` $(exp - bool)$

4. On utilise les instructions `return` et `break` avec la sémantique du langage C.

On identifie les types avec leur ensemble de valeurs. On ne précise pas les types de base, mais on inclut ici les types booléen `bool` et entier `int`. Les types composés sont obtenus en utilisant les constructeurs suivants :

1. Si T est un type de base alors on peut construire les types *file* et *pile* dont la syntaxe et les opérateurs sont présentés à la table 2.1[p.26]. Si f est une file (pile) de T et e est un ensemble de T alors on utilise la notation $f.Push(e)$ qui signifie une énumération aléatoire de tous les éléments de e avec leurs insertion dans e :
   `foreach`$(s \in e)$
   $f.Push(s)$;

2. De même, si T est un type de base alors on peut construire le type *ensemble* dont la syntaxe et les opérateurs sont présentés à la table 2.2[p.26]. Les opérateurs $\lhd$

Déclarations	Opérateurs	Notations, utilisations
$T' = \begin{cases} \texttt{file de}\ T \\ \texttt{pile de}\ T \end{cases}$	$Push : T' \times T \to T'$	$x.Push(t)$
$x : T'$	$Pop : T' \to T$	$t := x.Pop()$
$t : T$	$Top : T' \to T$	$t := x.Top()$
	$Empty : T' \to \texttt{bool}$	$x = \emptyset$
	$Reverse : T' \to T'$	$x := x.Reverse$

TAB. 2.1 – *Les types file, pile*

Déclarations	Opérateurs	Notations, utilisations
$T' = \texttt{ensemble de}\ T$	$\cup : T' \times T' \to T'$	$z := x \cup y$
$x, y, z : T'$	$\cap : T' \times T' \to T'$	$z := x \cap y$
$t : T$	$\setminus : T' \times T' \to T'$	$z := x \setminus y$
	$Empty : T' \to \texttt{bool}$	$x = \emptyset$
	$In : T' \times T \to \texttt{bool}$	$t \in x$
	$\{\} : T \to T'$	$x := \{t\}$
	$\lhd : T' \to T$	$t \lhd x$
	$\blacktriangleleft : T' \to T$	$t \blacktriangleleft x$

TAB. 2.2 – *Le type ensemble*

et ◀ signifient la sélection aléatoire d'un élément et respectivement la sélection aléatoire suivie par élimination de l'élément.

3. Si $T, T_1, \ldots, T_k$ sont des types de base alors on peut construire les types *tuple* et *fonction* dont la syntaxe et les opérateurs sont présentés à la table 2.3[p.27].

Déclarations	Opérateurs	Notations, utilisations
$T' = T_1 \times \ldots \times T_k$	$[] : T' \times \{1, \ldots, k\} \to \bigcup_{i \in \{1,\ldots,k\}} T_i$	$t := x[j]$
$x : T'$	$() : T_1 \times \ldots \times T_k \to T$	$x := (t_1, \ldots, t_k)$
$t : T_j$		
$t_i : T_i, i \in \{1, \ldots, k\}$		
$T'' = T_1 \times \ldots \times T_k \to T$	$() : T_1 \times \ldots \times T_k \to T$	$t := y(t_1, \ldots, t_k)$
$y : T''$		
$t : T$		
$t_i : T_i, i \in \{1, \ldots, k\}$		

TAB. 2.3 – *Les types tuple et fonction*

Par ailleurs on utilise les types STE et PFD qui dénotent les systèmes de transitions étiquetées (définition 3.1[p.38]) et les problèmes de flot de données (définition 5.1[p.97]). Pour les variables de type STE ou PFD (ex. x : STE) on utilise aussi la notation préfixée $x.$ pour identifier les éléments composants (ex. $x.\Sigma$).

La notation pour les commentaires est identique à celle de C++.

La définition, la déclaration et l'utilisation des procédures obéissent aux règles du langage PASCAL : elles peuvent contenir des définitions des procédures (ou seulement leurs déclarations), des définitions de types. Un exemple de procédure est fourni à la table 2.4[p.28] : la procédure *Proc1* contient, entre autres, la définition de la procédure *Proc3* et la déclaration de la procédure *Proc2*.

2.2 Ensembles partiellement ordonnés, treillis

2.2.1 Ensembles partiellement ordonnés

Définition 2.1 (Ensemble partiellement ordonné)

Un ensemble *partiellement ordonné* est la paire $(L, \sqsubseteq)$, où L est un ensemble et $\sqsubseteq \subseteq L \times L$ est un ordre partiel.

La *hauteur* d'un ensemble partiellement ordonné $(L, \sqsubseteq)$ est

```
procedure Proc1 (a : TypeA; var b : TypeB)
  type
    TypeC = ... ;
  var
    c : TypeC;
  procedure Proc2;
  procedure Proc3 (b1 : TypeB; var c1 : TypeC)
    type
      ...
    var
      ...
    begin
      ...
    endproc Proc3 ;
  begin
    ...
  end.
```

TAB. 2.4 – *Une procédure*

$h(L) = max\{|\,S\,|\;|\;S \subseteq L \wedge S$ totalement ordonné$\}$. ▪

Observation 2.1 *Si $a \sqsubset b \doteqdot a \sqsubseteq b \wedge a \neq b$, on pourrait représenter $(L, \sqsubseteq)$ comme un graphe orienté G_L avec pour nœuds les éléments de L et une arête entre a et b $(a, b \in L)$ si $a \sqsubset b \wedge (\forall x)(x \in L \Rightarrow \neg(a \sqsubset x \sqsubset b))$* [1].

Définition 2.2 (Chaîne)
La séquence $(x_n)_{n\in\mathbf{N}}$ dans $(L, \sqsubseteq)$ est une *chaîne croissante* si $x_i \sqsubseteq x_{i+1}$ pour tout $i \in \mathbf{N}$. ▪

Définition 2.3 (Monotonie)
Une fonction $f : L \to L$ est *monotone croissante* si $(\forall a)(\forall b)(a \in L \wedge b \in L \wedge a \sqsubseteq b \Rightarrow f(a) \sqsubseteq f(b))$. ▪

Définition 2.4 (Bornes)
Soit $S \subseteq L$. Un élément $b \in L$ est :

1. un *majorant* de S (noté $S \sqsubseteq b$) si $(\forall s)(s \in S \Rightarrow s \sqsubseteq b)$.

1. c'est une représentation minimale t.q. si $a \longrightarrow_{G_L} b$ et $b \longrightarrow_{G_L} c$ alors on peut déduire, par transitivité, que $a \sqsubset c$, sans que pour cela une arête soit nécessaire

2. un *minorant* de S (noté $b \sqsubseteq S$) si $(\forall s)(s \in S \Rightarrow b \sqsubseteq s)$.
3. la[2] *borne supérieure* de S (noté $\sqcup S$) si :
   (a) $\sqcup S$ est un majorant de S et
   (b) C'est le plus petit des majorants de S : $(\forall b')(b' \in L \wedge S \sqsubseteq b' \Rightarrow \sqcup S \sqsubseteq b')$.
4. la[3] *borne inférieure* de S (noté $\sqcap S$) si :
   (a) $\sqcap S$ est un minorant de S et
   (b) C'est le plus grand des minorants de S : $(\forall b')(b' \in L \wedge b' \sqsubseteq S \Rightarrow b' \sqsubseteq \sqcap S)$.

▪

Lemme 2.1 (Produit cartésien)
Soient $(L_1, \sqsubseteq_1)$ et $(L_2, \sqsubseteq_2)$ deux ensembles partiellement ordonnés et soit $\sqsubseteq \subseteq L_1 \times L_2$ t.q. $(x_1, x_2) \sqsubseteq (y_1, y_2) \iff x_1 \sqsubseteq_1 y_1 \wedge x_2 \sqsubseteq_2 y_2$. Alors :

1. $(L_1 \times L_2, \sqsubseteq)$ est un ensemble partiellement ordonné,
2. Un élément (b_1, b_2) du produit cartésien est un majorant (minorant) de $S_1 \times S_2 \subseteq L_1 \times L_2$ si et seulement si b_i est un majorant (minorant) de S_i, $i \in 1 \,..\, 2$.
3. La borne supérieure (inférieure) de $S_1 \times S_2 \subseteq L_1 \times L_2$ est l'élément $(\sqcup S_1, \sqcup S_2)$ (resp. $(\sqcap S_1, \sqcap S_2)$).

▪

2.2.2 Treillis

Définition 2.5 (Treillis (complet))
Un *treillis (complet)* est un ensemble partiellement ordonné $(L, \sqsubseteq)$ t.q. toute partie finie (respectivement partie) de L admet une borne supérieure et une borne inférieure.
On le note avec $(L, \sqsubseteq, \sqcup, \sqcap, \bot, \top)$ où $\bot \doteqdot \sqcap L$ et $\top \doteqdot \sqcup L$. Si $x, y \in L$ alors $x \sqcap y = \sqcap\{x, y\}$ et $x \sqcup y = \sqcup\{x, y\}$. ▪

Exemple 2.1

1. Le treillis 2^L de parties d'un ensemble L : $(2^L, \subseteq, \cup, \cap, \emptyset, L)$.

2. si elle existe elle est unique, à cause du fait que $\sqsubseteq$ est anti-symétrique
3. la même observation que pour la borne supérieure

2. Le treillis de vecteurs de bits BV : $(\{0,1\}, \leq, \sqcup, \sqcap, 0, 1)$ (isomorphe avec le treillis de parties d'un ensemble de cardinal 1).

3. Le treillis $Q \to L$, où Q est un ensemble fini et L un treillis complet, est le treillis complet $(Q \to L, \sqsubseteq', \sqcup', \sqcap', \bot', \top')$ avec :

   (a) $f \sqsubseteq' g \iff (\forall l)(l \in L \Rightarrow f(l) \sqsubseteq g(l))$.

   (b) $(f \sqcap' g)(l) = f(l) \sqcap g(l)$.

   (c) $(f \sqcup' g)(l) = f(l) \sqcup g(l)$.

   (d) $\top' = \bar{\top}$.

   (e) $\bot' = \bar{\bot}$.

4. Le treillis de constantes CONST : $(\{\bot, \top, 0, \pm 1, \pm 2, \cdots\}, \sqsubseteq, \sqcup, \sqcap, \bot, \top)$ dont G_{CONST} est représenté à la figure 2.1[p.31].

5. Le treillis d'intervalles entiers INT est $(I_{\mathbf{Z}} \cup \{\bot\}, \sqsubseteq, \sqcap, \sqcup, \bot, [-\infty, \infty])$ où :

   (a) $I_{\mathbf{Z}} = \{[a,b] \mid a \in \mathbf{Z} \cup \{-\infty\}, b \in \mathbf{Z} \cup \{\infty\}, a \leq b\}$,

   (b) $[b,c] \sqsubseteq [a,d] \iff a \leq b \leq c \leq d$,

   (c) $(\forall [a,b])([a,b] \in I_{\mathbf{Z}} \Rightarrow \bot \sqsubset [a,b])$, où $a \sqsubset b \doteqdot a \sqsubseteq b \wedge a \neq b$,

   (d) $(\forall [a,b])([a,b] \in I_{\mathbf{Z}} \Rightarrow \bot \sqcup [a,b] = [a,b] \wedge \bot \sqcap [a,b] = \bot)$,

   (e) $[a,b] \sqcup [c,d] = [min(a,c), max(b,d)]$,

   (f) $[a,b] \sqcap [c,d] = \begin{cases} \bot & , \quad [a,b] \cap [c,d] = \emptyset \\ [max(a,c), min(b,d)] & , \quad \text{sinon} \end{cases}$

   Le graphe G_{INT} est représenté dans la figure 2.1[p.31].

▪

Lemme 2.2 (Produit cartésien)
Soient $(L_i, \sqsubseteq_i, \sqcup_i, \sqcap_i, \bot_i, \top_i)$ deux treillis complets. Le treillis $(L_1 \times L_2, \sqsubseteq, \sqcup, \sqcap, (\bot_1, \bot_2), (\top_1, \top_2))$, où l'ordre partiel $\sqsubseteq$ est celui de la lemme 2.1[p.29], $(x_1, x_2) \sqcup (y_1, y_2) = (x_1 \sqcup_1 y_1, x_2 \sqcup_2 y_2)$ et $(x_1, x_2) \sqcap (y_1, y_2) = (x_1 \sqcap_1 y_1, x_2 \sqcap_2 y_2)$ est un treillis complet. ▪

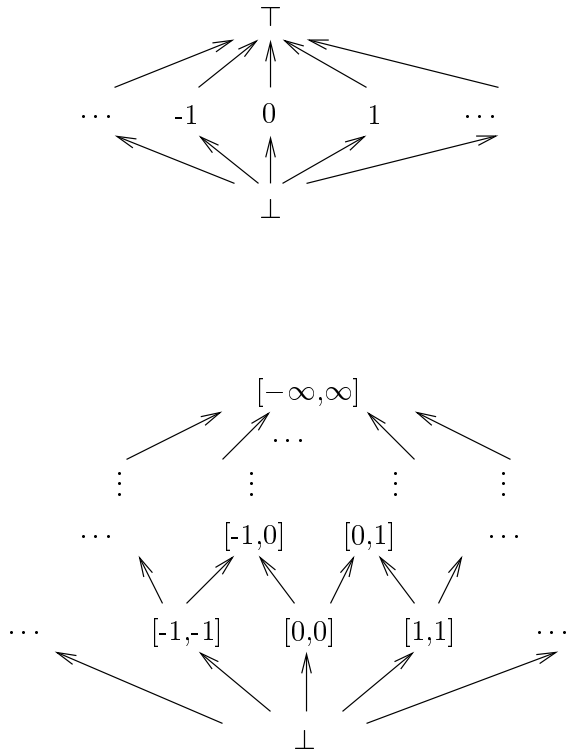

FIG. 2.1 – *Les treillis* CONST *et* INT

Lemme 2.3 (Union disjointe)
Soient $(L_i, \sqsubseteq_i, \sqcup_i, \sqcap_i, \bot_i, \top_i)$ deux treillis complets. Le treillis $(L_1 \times \{1\} \cup L_2 \times \{2\}, \sqsubseteq, \sqcup, \sqcap, \bot, \top)$, où :

$$(a,i) \sqsubseteq (b,j) \iff i < j \vee (i = j \wedge a \sqsubseteq_i b)$$

$$(a,i) \sqcup (b,j) = \begin{cases} (a \sqcup_i b, i) & , \quad i = j \\ (b,j) & , \quad i < j \\ (a,i) & , \quad i > j \end{cases} \qquad \top = (\top_2, 2)$$

$$(a,i) \sqcap (b,j) = \begin{cases} (a \sqcap_i b, i) & , \quad i = j \\ (b,j) & , \quad i > j \\ (a,i) & , \quad i < j \end{cases} \qquad \bot = (\bot_1, 1)$$

est un treillis complet (on utilise la notation $L_1 \cup L_2$). ▪

Définition 2.6 (Continuité)
Soit $(L, \sqsubseteq, \sqcup, \sqcap, \bot, \top)$ un treillis complet. La fonction totale $f : L \to L$ est *continue* si pour toute chaîne $(x_n)_{n \in \mathbf{N}}$, $f(\sqcup_{n \in \mathbf{N}} x_n) = \sqcup_{n \in \mathbf{N}} f(x_n)$. ▪

2.2.3 Points fixes

Définition 2.7 (Point fixe)
Soient $(L, \sqsubseteq, \sqcup, \sqcap, \bot, \top)$ un treillis complet et $f : L \to L$ une fonction monotone. Un *point fixe* de f est un élément x de L t.q. $f(x) = x$. Soient $\Phi_f \doteqdot \{x \mid x \in L \wedge f(x) = x\}$ et $lfp(f) \doteqdot \sqcap \Phi_f$, $gfp(f) \doteqdot \sqcup \Phi_f$. ▪

Lemme 2.4 (Monotonie et hauteur finie implique continuité)
Si L est un treillis complet, de la hauteur finie et $f : L \to L$ est monotone alors f est continue. ▪

Preuve
Voir annexe $B_{[\text{p.173}]}$.

□

Théorème 2.1 (Calcul de point fixe - Kleene-Tarski)
Soient $(L, \sqsubseteq, \sqcup, \sqcap, \bot, \top)$ un treillis complet et $f : L \to L$ une fonction totale. Alors :

1. Si f est monotone alors $\sqcup_{n \in \mathbf{N}} f^n(\bot) \sqsubseteq lfp(f)$ et $gfp(f) \sqsubseteq \sqcap_{n \in \mathbf{N}} f^n(\top)$.
2. Si f est continue alors $lfp(f) = \sqcup_{n \in \mathbf{N}} f^n(\bot)$ et $gfp(f) = \sqcap_{n \in \mathbf{N}} f^n(\top)$.

Preuve
Voir annexe $B_{[p.174]}$.

Observation 2.2
Le théorème précédent a lieu aussi si on prend au lieu de $\top$, $\bot$ tout $a \in L$ avec la propriété : $gfp(f) \sqsubseteq a \sqsubseteq \top$ (respectivement $\bot \sqsubseteq a \sqsubseteq lfp(f)$).

2.2.4 Élargissement, rétrécissement

Les opérateurs d'élargissement et de rétrécissement ont été introduits ([Cou78]) pour permettre et accélérer le calcul des approximations des points fixes dans le cas où le treillis a la hauteur infinie.

Définition 2.8 (Élargissement)
Soit $(L, \sqsubseteq, \sqcup, \sqcap, \bot, \top)$ un treillis complet. L'opérateur binaire $\nabla : L \times L \to L$ est un *opérateur d'élargissement* ([CC92b]) si :

1. $(\forall x)(\forall y)(x \in L \wedge y \in L \Rightarrow x, y \sqsubseteq x \nabla y)$ et
2. Pour toute séquence croissante $(x_n)_{n\in\mathbf{N}}$, la séquence croissante $(y_n)_{n\in\mathbf{N}}$, définie comme suit : $\begin{cases} y_0 = x_0 \\ y_{n+1} = y_n \nabla x_{n+1} \quad , \forall n \in \mathbf{N} \end{cases}$, est constante à partir d'un certain rang.

Définition 2.9 (Rétrécissement)
Soit $(L, \sqsubseteq, \sqcup, \sqcap, \bot, \top)$ un treillis complet. L'opérateur binaire $\triangle : L \times L \to L$ est un *opérateur de rétrécissement* ([CC92b]) si :

1. $(\forall x)(\forall y)(x \in L \wedge y \in L \wedge y \sqsubseteq x \Rightarrow y \sqsubseteq x \triangle y \sqsubseteq x)$ et
2. Pour toute séquence décroissante $(x_n)_{n\in\mathbf{N}}$, la séquence décroissante $(y_n)_{n\in\mathbf{N}}$, définie comme suit : $\begin{cases} y_0 = x_0 \\ y_{n+1} = y_n \triangle x_{n+1} \quad , \forall n \in \mathbf{N} \end{cases}$, est constante à partir d'un certain rang.

Exemple 2.2 *Pour le treillis* INT*, des opérateurs d'élargissement et de rétrécissement sont définis comme suit ([CC92b]):*

1. *Élargissement :*

$$(\forall X)(X \in I_{\mathbf{Z}} \Rightarrow ((\bot \nabla X = X) \wedge (X \nabla \bot = X)))$$

$$[a,b]\nabla[c,d] = [e,f],\ où\ e = \begin{cases} -\infty & , \quad c < a \\ a & , \quad sinon \end{cases} \quad et\ f = \begin{cases} \infty & , \quad b < d \\ b & , \quad sinon \end{cases}$$

2. *Rétrécissement :*

$$(\forall X)(X \in I_{\mathbf{Z}} \Rightarrow ((\bot \triangle X = \bot) \wedge (X \triangle \bot = \bot)))$$

$$[a,b]\triangle[c,d] = [e,f],\ où\ e = \begin{cases} c & , \quad a = -\infty \\ a & , \quad sinon \end{cases} \quad et\ f = \begin{cases} d & , \quad b = \infty \\ b & , \quad sinon \end{cases}$$

Théorème 2.2 (Approximation par élargissement [CC92b])
Soient $(L, \sqsubseteq, \sqcup, \sqcap, \bot, \top)$ un treillis complet, $f : L \to L$ une fonction continue et $\nabla : L \times L \to L$ un opérateur d'élargissement.

Soit la séquence $(x_n)_{n\in\mathbf{N}} : x_0 = \bot$ et $x_{i+1} = \begin{cases} x_i & , \quad f(x_i) \sqsubseteq x_i \\ x_i \nabla f(x_i) & , \quad \text{sinon} \end{cases}$.

Alors $(x_n)_{n\in\mathbf{N}}$ est constante à partir d'un certain rang et sa limite $a \in L$ est t.q. $lfp(f) \sqsubseteq a$ et $f(a) \sqsubseteq a$. ▪

Théorème 2.3 (Approximation par rétrécissement [CC92b])
Soient $(L, \sqsubseteq, \sqcup, \sqcap, \bot, \top)$ un treillis complet, $f : L \to L$ une fonction continue, $\triangle : L \times L \to L$ un opérateur de rétrécissement et $a \in L$ la limite de la séquence croissante du théorème 2.2[p.34].

Soit la séquence $(y_n)_{n\in\mathbf{N}} : y_0 = a$ et $y_{i+1} = y_i \triangle f(y_i)$.

Alors $(y_n)_{n\in\mathbf{N}}$ est constante à partir d'un certain rang et sa limite $a' \in L$ est t.q. $lfp(f) \sqsubseteq a'$. ▪

2.2.5 Itérations chaotiques

Dans le cas où L est un treillis complet et f est une fonction continue de l'arité n ($f : \prod_{i=1}^{n} L \to \prod_{i=1}^{n} L$) on peut utiliser pour le calcul de $lfp(f)$ les itérations chaotiques ([Cou98]) :

Définition 2.10 (Itérations chaotiques)

Soient L un treillis complet, $f : \prod_{i=1}^{n} L \to \prod_{i=1}^{n} L$ une fonction continue et $\beta : \mathbf{N} \to 2^{\{1,\ldots,n\}}$ une stratégie d'itération (t.q. chaque $i \in \{1,\ldots,n\}$ apparaît infiniment souvent : $(\forall i)(\forall k)(1 \leq i \leq n \wedge k \in \mathbf{N} \Rightarrow (\exists j)(j > k \wedge i \in \beta(j)))$).

Une *itération chaotique* (*croissante*) est la séquence $(x^k)_{k\in\mathbf{N}}$, où : $x^0 = (\bot,\ldots,\bot)$ et $x_i^{k+1} = \begin{cases} f_i(x_1^k,\ldots,x_n^k) & , \quad i \in \beta(k) \\ x_i^k & , \quad i \notin \beta(k) \end{cases}$, $\forall i \in 1 \,..\, \mathrm{n}$. ▪

Théorème 2.4 ([Cou78])

La suite $(x^k)_{k\in\mathbf{N}}$ est croissante et sa limite est le point fixe de f ($lfp(f) = \sqcup_{k\in\mathbf{N}} x^k$). ▪

Preuve

Voir annexe $B_{[\mathrm{p.174}]}$. □

Dans le cas des treillis de hauteur non finie (ou ne vérifiant pas la condition de chaîne) on introduit dans la définition des itérations chaotiques croissantes les suites élargies et on obtient les itérations chaotiques approchées supérieurement ([Cou78]) :

Définition 2.11 (Itérations chaotiques approchées supérieurement)

Soient L un treillis complet et $f : \prod_{i=1}^{n} L \to \prod_{i=1}^{n} L$ une fonction continue et $\beta : \mathbf{N} \to 2^{\{1,\ldots,n\}}$ une stratégie d'itération (t.q. chaque $i \in \{1,\ldots,n\}$ apparaît infiniment souvent) $((\forall i)(\forall k)(1 \leq i \leq n \wedge k \in \mathbf{N} \Rightarrow (\exists j)(j > k \wedge i \in \beta(j))))$.

1. Le *graphe de dépendance* de f est un graphe orienté avec les nœuds étiquetés avec $i, i \in \{1,\ldots,n\}$ et des arcs entre (i,j) si dans le calcul du point fixe x de f le calcul de la composante x_j dépend du celui de la composante x_i.

2. Soit $E \subseteq \{1,\ldots,n\}$ un ensemble t.q. tout circuit non trivial du graphe de dépendance de f passe par au moins un point de E.

   Une *itération chaotique croissante approchée supérieurement*) est la séquence $(x^k)_{k\in\mathbf{N}}$, où : $x^0 = (\bot,\ldots,\bot)$ et $x_i^{k+1} = \begin{cases} x_i^k & , \quad i \notin \beta(k) \vee f_i(x^k) \sqsubseteq x_i^k \\ f_i(x^k) & , \quad i \in \beta(k) \setminus E \wedge f_i(x^k) \not\sqsubseteq x_i^k \\ x_i^k \nabla f_i(x^k) & , \quad i \in \beta(k) \cap E \wedge f_i(x^k) \not\sqsubseteq x_i^k \end{cases}$.

▪

Théorème 2.5 ([Cou78])

Si f est continue, la suite $(x^k)_{k\in\mathbf{N}}$ est constante à partir d'un certain rang et sa limite x est un post-point fixe de f ($f(x) \sqsubseteq x$). ▪

Chapitre 3

Validation basée sur les modèles

La validation des systèmes répartis (protocoles) consiste à s'assurer que les implémentations des systèmes sont conformes à leurs spécifications (ensembles des propriétés qu'on attend d'un système). Plus formellement : valider le système S signifie à vérifier si la spécification SPEC de S et son implémentation IMP satisfont la relation `conf` : IMP `conf` SPEC.

On suppose toujours qu'on peut associer aux implémentations une sémantique opérationnelle, donnée par un STE[1] (même dans le cas où on ne connaît pas la structure de celle-ci). En fonction de l'information qu'on possède sur la structure des implémentations on peut appliquer les techniques de validation : *la vérification formelle* (section 3.3[p.51]) et *le test* (section 3.4[p.52]).

On commence par une présentation des modèles formels qu'on utilise : il s'agit de systèmes de transitions étiquetées (section 3.1[p.37]) et d'automates étendus communicants (section 3.2[p.46]).

3.1 Systèmes de transitions étiquetées

Les *systèmes de transitions étiquetées* (STE) sont utilisés comme modèle séquentiel sémantique[2] (en utilisant la technique d'entrelacement) pour les systèmes concurrents : CCS ([Mil80]) et CSP ([Hoa85]) mais aussi comme modèle théorique dans les domaines du test de conformité ([Bri88]) ou de la vérification des systèmes [QS83, CES83, Sif82] : comportementale énumérative (en utilisant relations de simulation et de bisimulation :

1. système de transitions étiquetées
2. De manière générale, la sémantique opérationnelle d'un langage permet d'associer à un programme un STE

ALDÉBARAN ([Fer88, Mou92, Ker94, BFKM98]) ou [BdS92]) ou logique (par évaluation de formules de logique temporelle [Cle90, FM95]).

Définition 3.1 (Système de transitions étiquetées)
Un *système de transitions étiquetées* S est un tuple $(Q, \Sigma, \{\xrightarrow{a} | \ a \in \Sigma\}, q_0)$, où Q est l'ensemble des états, Σ est l'ensemble des étiquettes (actions), les relations de transition sont t.q. $(\forall a \in \Sigma)(\xrightarrow{a} \subseteq Q \times Q)$ et $q_0 \in Q$ est l'état initial. ▪

On note $\mathcal{STE}(\Sigma)$ l'ensemble de tous les STEs avec l'ensemble des étiquettes Σ. Pour identifier les éléments d'un STE (ex. S) on utilise les notations : $S.Q$, $S.\Sigma$, $S.q_0$.

On utilise la notation $q \xrightarrow{a} q' \doteqdot (\exists q)(\exists q')(\exists a)(q \in Q \wedge q' \in Q \wedge a \in \Sigma \wedge (q, q') \in \xrightarrow{a})$. Si l'un des éléments du triplet (q, a, q') est fixé (ex. q) alors on omet la condition d'appartenance au domaine (ex. $q \in Q$) : $q \xrightarrow{a} q' \doteqdot (\exists q')(\exists a)(q' \in Q \wedge a \in \Sigma \wedge (q, q') \in \xrightarrow{a})$. La notation $q \xrightarrow{a}$ signifie $(\exists q')(q' \in Q \wedge q \xrightarrow{a} q')$.

Les fonctions *prédécesseur* et *successeur* sont $Pre_a, Post_a : 2^Q \to 2^Q$, définies pour tout $a \in \Sigma$:

$$Pre_a(S) = \{q \mid q \in Q \wedge (\exists q')(q' \in S \wedge q \xrightarrow{a} q')\}$$
$$Post_a(S) = \{q' \mid q' \in Q \wedge (\exists q)(q \in S \wedge q \xrightarrow{a} q')\}$$

Si $\Lambda \subseteq \Sigma$, on utilise les notations :

$$Pre_\Lambda(S) \doteqdot \{q \mid q \in Q \wedge (\exists a)(a \in \Lambda \wedge q \in Pre_a(S))\} \qquad Pre(S) \doteqdot Pre_\Sigma(S)$$
$$Post_\Lambda(S) \doteqdot \{q \mid q \in Q \wedge (\exists a)(a \in \Lambda \wedge q \in Post_a(S))\} \quad Pre(S) \doteqdot Post_\Sigma(S)$$

Si $S = \{q\}$, on utilise les notations $Pre(q)$ et $Pre_a(q)$ (et de même pour $Post$).

S est *déterministe* si $(\forall a)(\forall q)(a \in \Sigma \wedge q \in Q \Rightarrow |Post_a(q)| \leq 1)$.

La *fermeture réflexive et transitive* de relations $\{\xrightarrow{a} | \ a \in \Sigma\}$ est l'ensemble des relations $\{\xrightarrow{\sigma} | \ \sigma \in \Sigma^* \wedge \xrightarrow{\sigma} \subseteq Q \times Q\}$, définies comme suit : $\xrightarrow{\epsilon} = 1_{Q \times Q}$ et si $a \in \Sigma$ et $\sigma \in \Sigma^*$ alors $\xrightarrow{a \cdot \sigma} = \xrightarrow{a} \circ \xrightarrow{\sigma}$. On utilise les notations $\xrightarrow{*} \doteqdot \bigcup_{\sigma \in \Sigma^*} \xrightarrow{\sigma}$ et $\xrightarrow{+} \doteqdot \bigcup_{\sigma \in \Sigma^+} \xrightarrow{\sigma}$.

Si $p \in Q, \sigma \in \Sigma^*$ alors p `after` $\sigma \doteqdot \{q \mid q \in Q \wedge p \xrightarrow{\sigma} q\}$ est l'ensemble des états accessibles de p par la trace σ (si S est déterministe alors le cardinal de cet ensemble est 1) et $trace(p) \doteqdot \{\sigma \mid \sigma \in \Sigma^* \wedge p$ `after` $\sigma \neq \emptyset\}$ est l'ensemble des traces de S commençant avec p.

Un *chemin* γ de S (de longueur n) est une séquence : $q_0 \xrightarrow{a_1} q_1 \xrightarrow{a_2} \ldots \xrightarrow{a_n} q_n$, où $q_i \in Q, \forall i \in 0 \,..\, \mathrm{n}, a_i \in \Sigma, \forall i \in 1 \,..\, \mathrm{n}$ et les états $q_1, \ldots, q_n$ sont distincts.

Les chemin γ est un *cycle* si $q_0 = q_n$.
S est *acyclique* s'il ne contient pas de cycles.

Dans la suite on rappelle la notion de composition asynchrone des STEs et quelques relations permettant d'établir des critères de comparaison entre STEs.

Définition 3.2 (Produit asynchrone de STEs)
Soient p_1, p_2 deux STEs : $(Q_i, \Sigma_i, \{\xrightarrow{a}_i |\ a \in \Sigma_i\}, q_0^i)$, $i \in 1 \,..\, 2$. Leur *produit asynchrone* $p_1 \,|||\, p_2$ est le STE : $(Q_{12}, \Sigma_{12}, \{\xrightarrow{a}_{12} |\ a \in \Sigma_{12}\}, (q_0^1, q_0^2))$, où $\Sigma_{12} = \Sigma_1 \cup \Sigma_2$ et $Q_{12} \subseteq Q_1 \times Q_2$, $\{\xrightarrow{a}_{12} |\ a \in \Sigma_{12}\}$ sont construits inductivement, en appliquant les règles :

$$\frac{}{(q_0^1, q_0^2) \in Q_{12}} \tag{3.1}$$

$$\frac{(q_1, q_2) \in Q_{12} \quad q_1 \xrightarrow{a}_1 q_1'}{a \in \Sigma_{12} \quad (q_1', q_2) \in Q_{12} \quad (q_1, q_2) \xrightarrow{a}_{12} (q_1', q_2)} \tag{3.2}$$

$$\frac{(q_1, q_2) \in Q_{12} \quad q_2 \xrightarrow{a}_2 q_2'}{a \in \Sigma_{12} \quad (q_1, q_2') \in Q_{12} \quad (q_1, q_2) \xrightarrow{a}_{12} (q_1, q_2')} \tag{3.3}$$

▪

L'algorithme qui implémente les équations 3.1[p.39], 3.2[p.39] et 3.3[p.39] est montré à la table 3.1[p.40].

Lemme 3.1 (Associativité)
Si $p_1, p_2, p_3 \in \mathcal{STE}(\Sigma)$ alors $(p_1 \,|||\, p_2) \,|||\, p_3 = p_1 \,|||\, (p_2 \,|||\, p_3)$ (l'égalité des tuples dans le sens ensembliste). ▪

Preuve
La preuve résulte immédiatement par inclusion réciproque.

□

Le lemme précédent suggère qu'on peut omettre les parenthèses quand on écrit le produit asynchrone de plusieurs STEs.

Dans la suite on introduit une opération entre STEs qui sera utile dans les analyses de flot de données inter-processus.

Définition 3.3 (Union de STEs)

```
procedure Async
  (   (Q_1, Σ_1, {—a→_1 | a ∈ Σ_1}, q_0^1) : STE;
      (Q_2, Σ_2, {—a→_2 | a ∈ Σ_2}, q_0^2) : STE;
      var (Q_12, Σ_12, {—a→_12 | a ∈ Σ_12}, q_0^12) : STE    )
var
  S : pile de Q_1 × Q_2;
  x, succ : Q_1 × Q_2;
  y : Q_1 ∪ Q_2;
begin
  q_0^12 = (q_0^1, q_0^2); Q_12 := ∅; Σ_12 := Σ_1 ∪ Σ_2;
  foreach (a ∈ Σ_12)
    —a→_12 := ∅;
  S := ∅;
  S.Push((q_0^1, q_0^2));
  while (S ≠ ∅)
    x := S.Pop();
    Q_12 := Q_12 ∪ {x};
    foreach (a ∈ Σ_1)
      foreach (y ∈ Post_a^1(x[1]))
        succ := (y, x[2]);
        —a→_12 := —a→_12 ∪{(x, succ)};
        if (succ ∉ S ∪ Q_12)
          S.Push(succ);
      endfor
    foreach (a ∈ Σ_2)
      foreach (y ∈ Post_a^2(x[2]))
        succ := (x[1], y);
        —a→_12 := —a→_12 ∪{(x, succ)};
        if (succ ∉ S ∪ Q_12)
          S.Push(succ);
      endfor
  endwhile
end.
```

TAB. 3.1 – *Calcul du produit asynchrone*

Soient p_1, p_2 deux STEs disjoints[3] : $(Q_i, \Sigma_i, \{\xrightarrow{a}_i | \ a \in \Sigma_i\}, q_0^i)$, $i \in 1 \,..\, 2$.

On considère $\Sigma_1, \Sigma_2 \subseteq \Sigma$ (Σ est l'univers des étiquettes) et soit l'opérateur partiellement défini $\|$: $\Sigma \times \Sigma \to \Sigma$.

Leur *union* $p_1 \cup p_2$ est le STE : $(Q_{12}, \Sigma_{12} \times \{0,1,2\}, \{\xrightarrow{a} | \ a \in \Sigma_{12} \times \{0,1,2\}\}, q_0^{12})$, où : Σ_{12}, $Q_{12} = (Q_1 \cup Q_2 \setminus \{q_0^1, q_0^2\}) \cup \{q_0^{12}\}$ et $\{\xrightarrow{a} | \ a \in \Sigma_{12} \times \{0,1,2\}\}$ sont construits inductivement, en appliquant les règles :

$$\frac{q_0^i \xrightarrow{a}_i q_i' \quad i \in 1 \,..\, 2}{(a,i) \in \Sigma_{12} \times \{0,1,2\} \quad q_i' \in Q_{12} \quad q_0^{12} \xrightarrow{(a,i)} q_i'} \tag{3.4}$$

$$\frac{q_i \in Q_{12} \quad q_i \xrightarrow{a}_i q_i' \quad q_i' \neq q_0^i \quad i \in 1 \,..\, 2}{(a,i) \in \Sigma_{12} \times \{0,1,2\} \quad q_i' \in Q_{12} \quad q_i \xrightarrow{(a,i)} q_i'} \tag{3.5}$$

$$\frac{q_i \in Q_{12} \quad q_i \xrightarrow{a}_i q_0^i \quad i \in 1 \,..\, 2}{(a,i) \in \Sigma_{12} \times \{0,1,2\} \quad q_i \xrightarrow{(a,i)} q_0^{12}} \tag{3.6}$$

$$\frac{q_1, q_2 \in Q_{12} \quad q_1 \xrightarrow{a_1}_1 q_1' \quad q_2 \xrightarrow{a_2}_2 q_2' \quad a_1 \parallel a_2 \in \Sigma}{(a_1 \parallel a_2, 0) \in \Sigma_{12} \times \{0,1,2\} \quad q_1' \in Q_{12} \quad q_2 \xrightarrow{(a_1 \| a_2, 0)} q_1'} \tag{3.7}$$

▪

Exemple 3.1 *Voir l'exemple* $3.2_{[p.43]}$. *Ici on considère que les deux* STEs $1, 2$ *sont à la fois des* STEs *mais aussi des* τ-STEs. *Le* STE $1 \cup 2$ *est le* STE *union et* $\tau - 1 \cup \tau - 2$ *est le* τ-STE *union de* $1, 2$ *(vus dans ce cas comme des* τ-STEs*).*

Observation 3.1

La construction ci-dessus s'étend facilement pour n STEs : on remplace tous les états initiaux par un autre état initial et on construit les transitions étiquetées avec $a \parallel b$.

L'opérateur $\|$ sera utilisé dans la suite pour modéliser le concept de communication synchrone ou asynchrone entre les deux STEs. ▪

On utilise aussi les notions de *simulation* et de *bisimulation* ([Par81, Mil80]). Plus généralement ces relations peuvent être paramétrées par une famille de langage réguliers Λ ($\Lambda \subseteq 2^{(\Sigma \cup \{\tau\})^*}$)[4], mais on aura besoin seulement du cas $\Lambda = \Sigma$.

Soient $S_i = (Q_i, \Sigma, \{\xrightarrow{a}_i | \ a \in \Sigma\}, q_0^i)$, avec $i \in 1 \,..\, 2$, deux STEs.

Définition 3.4 (Simulation)

3. dans le sens ensembliste
4. [Mou92]

La relation $\text{SIM} \subseteq Q_1 \times Q_2$ est une *simulation* si

$$(\forall p_1)(\forall p_2)(\forall a)(\forall q_1) \quad (\quad (p_1, p_2) \in \text{SIM} \ \wedge \ p_1 \xrightarrow{a}_1 q_1 \Rightarrow (\exists q_2)(p_2 \xrightarrow{a}_2 q_2 \ \wedge \ (q_1, q_2) \in \text{SIM})).$$

▪

Définition 3.5 (Bisimulation)
La relation de simulation $\text{BSIM} \subseteq Q_1 \times Q_2$ est une *bisimulation* si BSIM^{-1} est une relation de simulation :

$$(\forall p_1)(\forall p_2)(\forall a) \quad (\quad (p_1, p_2) \in \text{BSIM} \wedge (\forall q_1)(p_1 \xrightarrow{a}_1 q_1 \Rightarrow (\exists q_2)(p_2 \xrightarrow{a}_2 q_2 \ \wedge \ (q_1, q_2) \in \text{BSIM})) \wedge (\forall q_2)(p_2 \xrightarrow{a}_2 q_2 \Rightarrow (\exists q_1)(p_1 \xrightarrow{a}_1 q_1 \ \wedge \ (q_1, q_2) \in \text{BSIM}))).$$

▪

La relation de bisimulation utilisée dans la suite est la relation de bisimulation forte $\sim$ ([Par81]) construite comme suit (lorsque les ensembles Q_1, Q_2 sont finis) : $\sim = \bigcap_{n \in \mathbf{N}} \sim_n$ et

$$\begin{cases} \sim_0 \ = \ Q_1 \times Q_2 \\ \sim_{n+1} \ = \ \{(p_1, p_2) \mid (p_1, p_2) \in Q_1 \times Q_2 \wedge (\forall a)(a \in \Sigma \wedge \\ \qquad (\forall q_1)(q_1 \in Q_1 \wedge p_1 \xrightarrow{a}_1 q_1 \Rightarrow \\ \qquad\quad (\exists q_2)(q_2 \in Q_2 \ \wedge \ p_2 \xrightarrow{a}_2 q_2 \ \wedge \ q_1 \sim_n q_2)) \wedge \\ \qquad (\forall q_2)(q_2 \in Q_2 \wedge p_2 \xrightarrow{a}_2 q_2 \Rightarrow \\ \qquad\quad (\exists q_1)(q_1 \in Q_1 \ \wedge \ p_1 \xrightarrow{a}_1 q_1 \ \wedge \ q_1 \sim_n q_2)))\} \end{cases}.$$

3.1.1 Actions internes

Une catégorie particulière de STEs est constituée par les STEs avec des actions internes (non-observables), dénotées par l'étiquette τ. Un τ-STE est le STE : $(Q, \Sigma_\tau, \{\xrightarrow{a} \mid a \in \Sigma_\tau\}, q_0)$, où $\Sigma_\tau \doteqdot \Sigma \cup \{\tau\}$.

On utilise la notation $\mathcal{STE}_\tau(\Sigma)$ pour l'ensemble de tous les τ-STEs avec l'ensemble des étiquettes Σ et $\mathcal{STE}_\tau$ pour l'ensemble de tous les τ-STEs.

En utilisant la fermeture réflexive et transitive de relations $\{\xrightarrow{a} \mid a \in \Sigma_\tau\}$: $\{\xrightarrow{\sigma} \mid \xrightarrow{\sigma} \in Q \times Q \wedge \sigma \in \Sigma_\tau^*\}$, on construit les relations de transition suivantes : $\{\overset{\sigma}{\Longrightarrow} \mid \overset{\sigma}{\Longrightarrow} \in Q \times$

$Q \wedge \sigma \in \Sigma^*\} : \stackrel{\epsilon}{\Longrightarrow} = \stackrel{\tau^*}{\longrightarrow}$ et si $a \in \Sigma$ et $\sigma \in \Sigma^*$ alors $\stackrel{a}{\Longrightarrow} = \stackrel{\epsilon}{\Longrightarrow} \circ \stackrel{a}{\longrightarrow} \circ \stackrel{\epsilon}{\Longrightarrow}$ et $\stackrel{a\cdot\sigma}{\Longrightarrow} = \stackrel{\epsilon}{\Longrightarrow} \circ \stackrel{a}{\longrightarrow} \circ \stackrel{\epsilon}{\Longrightarrow} \circ \stackrel{\sigma}{\Longrightarrow} \circ \stackrel{\epsilon}{\Longrightarrow}$.

Si $p \in Q, \sigma \in \Sigma^*, A \subseteq \Sigma$ alors p `after` $\sigma \doteqdot \{q \mid q \in Q \wedge p \stackrel{\sigma}{\Longrightarrow} q\}$, $trace(p) \doteqdot \{\sigma \mid \sigma \in \Sigma^* \wedge p$ `after` $\sigma \neq \emptyset\}$, p `after` σ `refuses` $A \doteqdot (\exists p')(p' \in Q \wedge p \stackrel{\sigma}{\Longrightarrow} p' \wedge (\forall a \in A)(\neg p' \stackrel{a}{\Longrightarrow})$ et p `after` σ `deadlocks` $\doteqdot p$ `after` σ `refuses` Σ.

Définition 3.6 (Union de τ-STEs)

Soient p_1, p_2 deux τ-STEs disjoints : $(Q_i, \Sigma^i_\tau, \{\stackrel{a}{\longrightarrow}_i \mid a \in \Sigma^i_\tau\}, q^i_0)$, $i \in 1 \,..\, 2$.

On considère $\Sigma^1_\tau, \Sigma^2_\tau \subseteq \Sigma_\tau$ (Σ_τ est l'univers des étiquettes) et soit l'opérateur partiellement défini $\| : \Sigma_\tau \times \Sigma_\tau \to \Sigma_\tau$.

Leur *union* $p_1 \cup p_2$ est le τ-STE : $(Q_{12}, \Sigma^{12}_\tau \times \{0,1,2\}, \{\stackrel{a}{\longrightarrow} \mid a \in \Sigma^{12}_\tau \times \{0,1,2\}\}, q^{12}_0)$, où : Σ^{12}_τ, $Q_{12} = (Q_1 \cup Q_2 \cup \{q^{12}_0\}$ et $\{\stackrel{a}{\longrightarrow} \mid a \in \Sigma^{12}_\tau \times \{0,1,2\}\}$ sont construits inductivement, en appliquant les règles :

$$\frac{-}{(\tau,0) \in \Sigma^{12}_\tau \times \{0,1,2\} \quad q^i_0 \in Q_{12} \quad q^{12}_0 \stackrel{(\tau,0)}{\longrightarrow} q^i_0} \tag{3.8}$$

$$\frac{q^i_0 \stackrel{a}{\longrightarrow}_i q'_i \quad i \in 1 \,..\, 2}{(a,i) \in \Sigma^{12}_\tau \times \{0,1,2\} \quad q'_i \in Q_{12} \quad q^{12}_0 \stackrel{(a,i)}{\longrightarrow} q'_i} \tag{3.9}$$

$$\frac{q_1, q_2 \in Q_{12} \quad q_1 \stackrel{a_1}{\longrightarrow}_1 q'_1 \quad q_2 \stackrel{a_2}{\longrightarrow}_2 q'_2 \quad a_1 \parallel a_2 \in \Sigma_\tau}{(a_1 \parallel a_2, 0) \in \Sigma^{12}_\tau \times \{0,1,2\} \quad q'_1 \in Q_{12} \quad q_2 \stackrel{(a_1 \| a_2, 0)}{\longrightarrow} q'_1} \tag{3.10}$$

∎

Exemple 3.2 (Union des (τ-)STEs) *Dans la figure* 3.1[p.44] *on a représenté deux* STE*s, notés par* 1 *et* 2 *et leur unions :* $1 \cup 2$ *et* $\tau - 1 \;\cup\; \tau - 2$ *(on a considéré que* 1 *et* 2 *sont aussi des* τ-STE*s). Dans les cas de* $1 \cup 2$ *les états initiaux de* 1 *et* 2 *sont remplacés par un seul état initial (règles* 3.4[p.41] *et* 3.6[p.41]*) et dans le cas de* $\tau - 1 \;\cup\; \tau - 2$ *on ajoute un nouveau état (qui devient l'état initial) et des nouvelles transitions de cet état vers les anciens états initiaux de chaque* τ-STE *(conformément à la règle* 3.8[p.43]*). On a considéré aussi que l'opérateur* $\|$ *est défini seulement dans le cas de* $c \parallel d$ *et on a ajouté la transition conformément aux règles* 3.7[p.41] *ou* 3.10[p.43].

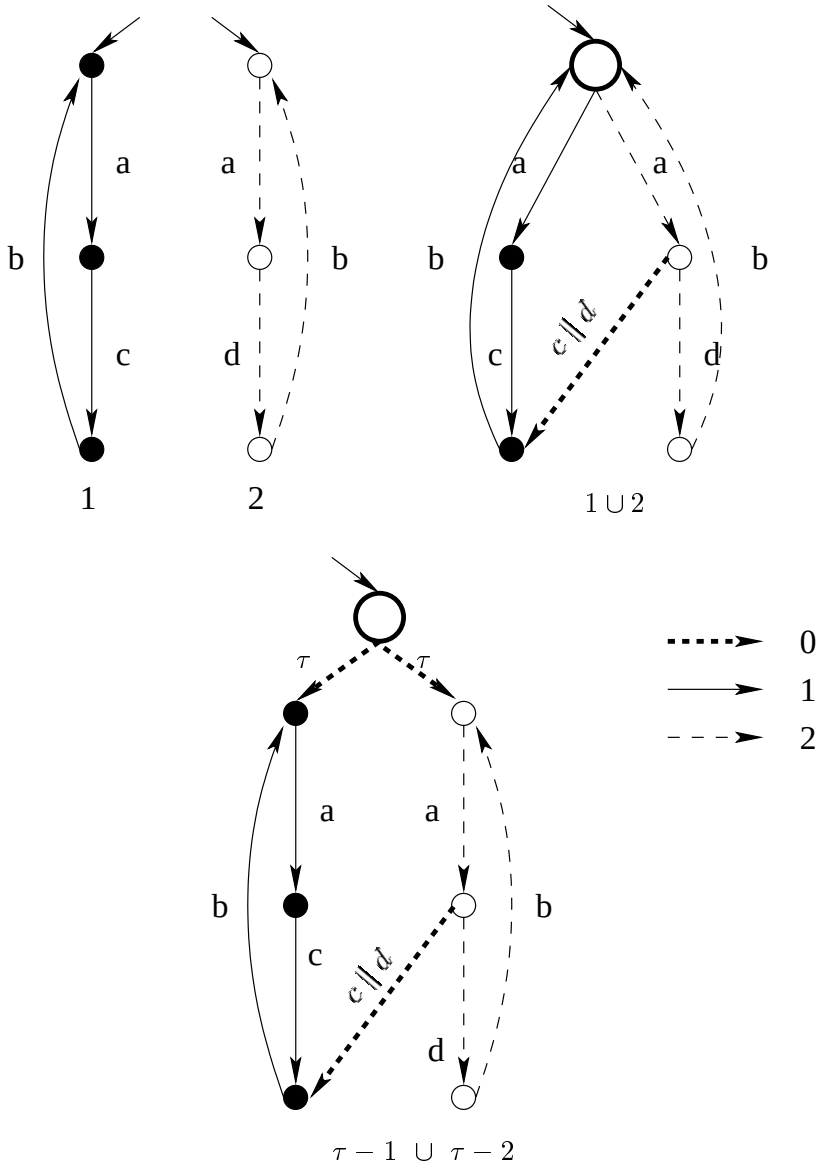

FIG. 3.1 – *Union des* STE*s*

3.1.2 Actions d'entrée et de sortie

Une sous-catégorie de τ-STEs est constituée par les STEs à entrée-sortie (IOSTEs). Ceux-ci ont l'ensemble d'étiquettes Σ partitionné en deux sous-ensembles :Σ_I, l'ensemble d'étiquettes d'entrée et Σ_O, l'ensemble d'étiquettes de sortie.

On utilise la notation $\mathcal{IOSTE}(\Sigma_I, \Sigma_O)$ pour l'ensemble de tous les IOSTEs avec l'ensemble des étiquettes d'entrée Σ_I et celui des étiquettes de sortie Σ_O et $\mathcal{IOSTE}$ pour l'ensemble de tous les IOSTEs.

Pour utiliser les IOSTEs dans la modélisation des systèmes distribués on leur impose quelques conditions :

1. Dans [LT89] on impose la condition *forte d'entrée autorisée*[5] : dans chaque état et pour chaque action d'entrée il existe une transition étiquetée avec cette action : $(\forall a \in \Sigma_I)(\forall p \in Q)(p \xrightarrow{a})$. Dans [Pha94a] cette condition est la condition de *complétude*.

2. Dans [Tre96] on impose la condition d'*faible d'entrée autorisée*[6] : dans chaque état et pour chaque action d'entrée a de Σ_I il existe une trace $\stackrel{a}{\Longrightarrow}$: $(\forall a \in \Sigma_I)(\forall p \in Q)(p \stackrel{a}{\Longrightarrow})$.

3. Dans [FJJV97, FJJV96] on définit la condition de *contrôlabilité* : dans chaque état si une sortie[7] est autorisée[8] alors elle est l'étiquette de la seule transition sortante de cet état.

4. Dans [Pha94a] on impose l'existence d'une correspondance bijective entre les ensembles Σ_I et Σ_O : $\Sigma_I = \{? \ a \mid a \in L\}$ et $\Sigma_O = \{! \ a \mid a \in L\}$.

Les IOSTEs sont utilisés pour modéliser des systèmes avec des communications asymétriques[9] (et dans ce cas les étiquettes d'entrée signifient les actions contrôlables et les étiquettes de sortie les actions observables). Les conditions d'entrée autorisée (forte ou faible) signifient que le système (modélisé par le IOSTE) ne peut jamais refuser d'interagir avec l'environnement quand l'initiative appartient à celui-ci (dans tout état, toute entrée est autorisée et sera acceptée par le système).

5. *strong input enabling* en anglais
6. *weak input enabling* en anglais
7. dans le cas où le IOSTE modélise un observateur où une entrée dans le cas où le IOSTE modélise une implémentation
8. *enabled* en anglais
9. les communications dont on peut établir à qui appartient l'initiative de la communication

La condition de contrôlabilité est utile dans la génération de test : dans un cas de test, dans chaque état, on peut soit attendre une émission du système sous test soit lui envoyer un message.

3.2 Automates étendus communicants

Les *automates étendus communicants* (AECs) constituent aujourd'hui un modèle largement utilisé dans la spécification et la validation des protocoles. L'évolution vers ce modèle a commencé par l'utilisation des *automates* pour modéliser formellement les protocoles. Il est mentionné dans [Hol91] que leurs utilisation a commencé avec [BSW69]. Les *automates étendus* ([Hol91, Pha94a]) constituent un modèle plus expressif que les STEs en ce qui concerne l'expression et la manipulation des données (en utilisant des variables). Les *automates communicants* ont été introduits par [BZ83]. Ils enrichissent les STEs en permettant l'expression des interactions entre plusieurs processus et l'environnement (par échanges des messages via canaux de communication).

On utilise les AECs comme langage de spécification. Plus précisément les spécifications sont constituées d'un nombre *fini* de processus asynchrones parallèles qui communiquent par des messages échangés via canaux de communication (files d'attente non-bornées). Un message est constitué d'un signal et de paramètres. Un processus est un AEC avec un nombre fini d'états. Les processus ont une mémoire locale (des variables locales) et ils exécutent des actions sur les files d'attente et les variables locales. Par souci de simplicité, les actions sont des commandes gardées simples.

Ce modèle est une version simplifiée de IF ([BFG+99b, BFG+99c, BFG+00c]), un modèle intermédiaire à base d'automates temporisés [AD94] communicants. On a simplifié dans le modèle IF les aspects temporisés et les actions (dans IF elles sont constituées par plusieurs commandes et une garde).

Syntaxe

Définition 3.7 (Spécification)
Une *spécification SP* est le tuple (S, C, P) où S est un ensemble des signaux, C est l'ensemble des canaux de communication et P est un ensemble des AECs. ▪

L'ensemble des canaux, C, est partitionné en deux sous-ensembles : C^{int} (les canaux internes, fermés à l'intérieur de la spécification) et C^{ext} (les canaux externes, ouverts à l'environnement).

Soient D l'ensemble des constantes logiques (vrai, faux) et entières auxquelles on ajoute une constante spéciale $\oslash$ qui dénote une valeur non-définie : $\mathbf{Z} \cup \{\mathbf{t}, \mathbf{f}\} \cup \{\oslash\}$, X un ensemble fini de variables (qui contient une variable spéciale Ω pouvant prendre toute valeur de D dans tout état de contrôle) et AE_X, BE_X les langages des expressions arithmétiques et logiques générés par les règles de production R_{ae} et R_{be} indiquées à la table 3.2[p.47].

R_{ae}	AE	$::=$	$k, \forall k \in D \setminus \{\mathbf{t}, \mathbf{f}\}$
	AE	$::=$	$v, \forall v \in X$
	AE	$::=$	$AE + AE \mid AE - AE$
R_{be}	BE	$::=$	$\mathbf{t} \mid \mathbf{f}$
	BE	$::=$	$\neg BE \mid BE \vee BE$
	BE	$::=$	$AE < AE \mid AE = AE$

TAB. 3.2 – *Règles de production*

Soient X, S et C avec les significations ci-dessus. L'univers des *commandes gardées* $\Sigma(X)$ est l'ensemble :

$$\{[be]\ x := ae \mid be \in BE_X \wedge x \in X \wedge ae \in AE_X\} \cup$$
$$\{[be]\ c\ ?\ s(x) \mid be \in BE_X \wedge x \in X \wedge c \in C \wedge s \in S\} \cup$$
$$\{[be]\ c\ !\ s(ae) \mid be \in BE_X \wedge ae \in AE_X \wedge c \in C \wedge s \in S\} \cup$$

$$\{[be]\ nil \mid be \in BE_X\} \cup$$
$$\{[be]\ c\ !\ s(\oslash) \mid be \in BE_X \wedge c \in C \wedge s \in S\} \cup$$
$$\{[be]\ c\ ?\ s(\Omega) \mid be \in BE_X \wedge c \in C \wedge s \in S\}$$

Définition 3.8 (Automate étendu communicant)
Un *automate étendu communicant* $p \in P$ est le tuple $(X_p, (Q_p, \Sigma(X_p), \{\xrightarrow{a}_p \mid a \in \Sigma(X_p)\}, q_0^p))$, où X_p est l'ensemble des variables, et $(Q_p, \Sigma(X_p), \{\xrightarrow{a}_p \mid a \in \Sigma(X_p)\}, q_0^p)$ est le STE sous-jacent.

Dans la suite les commandes $\langle [be]\ nil \rangle$, $\langle [be]\ c\ !\ s(\oslash) \rangle$ et $\langle [be]\ c\ ?\ s(\Omega) \rangle$ ne seront utilisées que dans le cas de l'analyse de l'utilité des données (chapitre 5[p.93]). ▪

Si $a \in \Sigma(X_p), a = [be] \cdots$ alors on dénote par *garde* l'expression logique be ($guard(a) \doteqdot be$) et par *action* ($act(a)$) le reste de l'action. Pour une action de type $x := ae$ on dénote par $lhs(x := ae) \doteqdot x$ et $rhs(x := ae) \doteqdot ae$.

Le fait que dans notre modèle il n'existe pas de mémoire partagée s'exprime formellement par $(\forall p)(\forall q)(p \in P \wedge q \in P \Rightarrow X_p \cap X_q = \emptyset)$.

On peut considérer, sans restreindre la généralité, que, pour tout AEC de la spécification, les files d'entrées et celles de sortie diffèrent.

La composition asynchrone de plusieurs AECs est aussi un AEC dont l'ensemble des variable est l'union des ensembles des variables locales et le STE sous-jacent est la composition asynchrone des STEs locaux :

Définition 3.9 (Produit asynchrone de AECs)
Soient $p, r \in P$ deux AECs :
$(X_p, (Q_p, \Sigma(X_p), \{\xrightarrow{a}_p |\ a \in \Sigma(X_p)\}, q_0^p))$ et $(X_r, (Q_r, \Sigma(X_r), \{\xrightarrow{a}_r |\ a \in \Sigma(X_r)\}, q_0^r))$.
Leur *produit asynchrone* $p \,|||\, r$ est l'AEC
$(X_p \cup X_r, (Q_p, \Sigma(X_p), \{\xrightarrow{a}_p |\ a \in \Sigma(X_p)\}, q_0^p) \,|||\, (Q_r, \Sigma(X_r), \{\xrightarrow{a}_r |\ a \in \Sigma(X_r)\}, q_0^r))$. ▪

Définition 3.10 (Union d'AECs)
Soient $p, r \in P$ deux automates étendus communicants :
$(X_p, (Q_p, \Sigma(X_p), \{\xrightarrow{a}_p |\ a \in \Sigma(X_p)\}, q_0^p))$ et $(X_r, (Q_r, \Sigma(X_r), \{\xrightarrow{a}_r |\ a \in \Sigma(X_r)\}, q_0^r))$.
Soit $\|$: $\Sigma(X_p \cup X_r) \times \Sigma(X_p \cup X_r) \to \Sigma(X_p \cup X_r)$ un opérateur partiellement défini t.q. si $\sigma_1, \sigma_2 \in \Sigma(X_p \cup X_r)$, $\sigma_1 \parallel \sigma_2$ est défini ssi $\sigma_1 = [be_1]\ c\ ?\ s(x)$ et $\sigma_2 = [be_2]\ c\ !\ s(e)$ et dans ce cas $\sigma_1 \parallel \sigma_2 = \langle [\mathbf{t}]\ x\ :=\ e \rangle$.
L'*union* $p \cup r$ (avec l'univers des étiquettes $\Sigma(X_p \cup X_r)$) est l'AEC :
$(X_p \cup X_r, (Q_p, \Sigma(X_p), \{\xrightarrow{a}_p |\ a \in \Sigma(X_p)\}, q_0^p) \cup (Q_r, \Sigma(X_r), \{\xrightarrow{a}_r |\ a \in \Sigma(X_r)\}, q_0^r))$. ▪

Sémantique

On considère que la communication entre les AECs de la spécification est asynchrone. Les *canaux* de communication *internes* sont des *files d'attente non-bornées*.

La communication avec l'environnement est aussi asynchrone mais en voulant modéliser une communication synchrone entre le testeur et l'IST on considère les canaux de communication externes comme des files d'attente bornées (avec la borne 1).

Définition 3.11 (Contextes)
Soit la spécification SP.
Un *contexte de variables* est une fonction $\sigma : \bigcup_{p \in P} X_p \to D$. On note aussi avec σ son extension pour expressions (on impose $\sigma(e) = \oslash$ si l'expression e contient une variable x t.q. $\sigma(x) = \oslash$).
Un *contexte de files* est une fonction $\rho : C^{\text{int}} \to (S \times D)^*$. ▪

Définition 3.12 (Sémantique de la spécification)
La *sémantique* de la spécification SP est le STE :$\widehat{SP} = (\widehat{Q}, \widehat{\Sigma}, \{\xrightarrow{a}| \ a \in \widehat{\Sigma}\}, \hat{q}_0)$, où :

1. Les états sont des tuples constitués par les états de contrôle de chaque processus, les contextes de variables et les contextes de files internes :
   $\widehat{Q} \subseteq (\prod_{p \in P} Q_p) \times (\bigcup_{p \in P} X_p \to D) \times (C^{\text{int}} \to (S \times D)^*)$,

2. L'ensemble d'étiquettes $\widehat{\Sigma}$ est
   $\{\tau\} \cup \{c \ ? \ s(v) \mid c \in C^{\text{ext}} \wedge s \in S \wedge v \in D\} \cup \{c \ ! \ s(v) \mid c \in C^{\text{ext}} \wedge s \in S \wedge v \in D\}$,

3. L'état initial est constitué par l'état initial de chaque processus, les variables initialisées avec 0 et les files internes initialisées avec ϵ :
   $\hat{q}_0 = ((q_0^p)_{p \in P}, 0^{\sum_{p \in P} |X_p|}, \epsilon^{|C^{\text{int}}|})$.

Les ensembles $\widehat{Q}$ et $\{\xrightarrow{a}| \ a \in \widehat{\Sigma}\}$ sont construits inductivement, en appliquant les règles :

$$\frac{-}{\hat{q}_0 \in \widehat{Q}} \tag{3.11}$$

$$\frac{(\theta, \sigma, \rho) \in \widehat{Q} \quad q_p \xrightarrow{[b]\, x:=e}_p q'_p \quad \sigma(b) = \mathbf{t} \quad \sigma(e) = v}{([q'_p/p]\theta, [v/x]\sigma, \rho) \in \widehat{Q} \quad (\theta, \sigma, \rho) \xrightarrow{\tau} ([q'_p/p]\theta, [v/x]\sigma, \rho)} \tag{3.12}$$

$$\frac{(\theta, \sigma, \rho) \in \widehat{Q} \quad q_p \xrightarrow{[b]\, c!s(e)}_p q'_p \quad \sigma(b) = \mathbf{t} \quad \sigma(e) = v \quad c \in C^{\text{ext}}}{([q'_p/p]\theta, \sigma, \rho) \in \widehat{Q} \quad (\theta, \sigma, \rho) \xrightarrow{c!s(v)} ([q'_p/p]\theta, \sigma, \rho)} \tag{3.13}$$

$$\frac{(\theta, \sigma, \rho) \in \widehat{Q} \quad q_p \xrightarrow{[b]\, c!s(e)}_p q'_p \quad \sigma(b) = \mathbf{t} \quad \sigma(e) = v \quad c \in C^{\text{int}} \quad \rho(c) = w}{([q'_p/p]\theta, \sigma, [w \cdot (s, v)/c]\rho) \in \widehat{Q} \quad (\theta, \sigma, \rho) \xrightarrow{\tau} ([q'_p/p]\theta, \sigma, [w \cdot (s, v)/c]\rho)} \tag{3.14}$$

$$\frac{(\theta, \sigma, \rho) \in \widehat{Q} \quad q_p \xrightarrow{[b]\, c?s(x)}_p q'_p \quad \sigma(b) = \mathbf{t} \quad c \in C^{\text{ext}} \quad v \in D \setminus \{\oslash\}}{([q'_p/p]\theta, [v/x]\sigma, \rho) \in \widehat{Q} \quad (\theta, \sigma, \rho) \xrightarrow{c?s(v)} ([q'_p/p]\theta, [v/x]\sigma, \rho)} \tag{3.15}$$

$$\frac{(\theta, \sigma, \rho) \in \widehat{Q} \quad q_p \xrightarrow{[b]\, c?s(x)}_p q'_p \quad \sigma(b) = \mathbf{t} \quad c \in C^{\text{int}} \quad \rho(c) = (s, v) \cdot w \quad v \in D \setminus \{\oslash\}}{([q'_p/p]\theta, [v/x]\sigma, [w/c]\rho) \in \widehat{Q} \quad (\theta, \sigma, \rho) \xrightarrow{\tau} ([q'_p/p]\theta, [v/x]\sigma, [w/c]\rho)} \tag{3.16}$$

Les règles 3.11$_{[p.49]}$ ÷ 3.16$_{[p.49]}$ sont les règles sémantiques de base pour les spécifications. On y ajoute quelques règles concernant les commandes $[b]\ nil$, $[b]\ c!s(\oslash)$ et $[b]\ c?s(\Omega)$. Ces règles sont utilises seulement dans le cas de l'analyse de l'utilité des données (chapitre 5$_{[p.93]}$) et si on ne précise pas, on considère la sémantique des spécifications donnée seulement par les règles 3.11$_{[p.49]}$ ÷ 3.16$_{[p.49]}$ (et dans ce cas on omet de mentionner la prémisse $v \in D \setminus \{\oslash\}$ dans les règles 3.15$_{[p.49]}$ et 3.16$_{[p.49]}$).

$$\frac{(\theta,\sigma,\rho) \in \widehat{Q} \quad q_p \xrightarrow{[b]\ nil}_p q'_p \quad \sigma(b) = \mathbf{t}}{([q'_p/p]\theta,\sigma,\rho) \in \widehat{Q} \quad (\theta,\sigma,\rho) \xrightarrow{\tau} ([q'_p/p]\theta,\sigma,\rho)} \tag{3.17}$$

$$\frac{(\theta,\sigma,\rho) \in \widehat{Q} \quad q_p \xrightarrow{[b]\ c?s(\Omega)}_p q'_p \quad \sigma(b) = \mathbf{t} \quad c \in C^{\text{ext}} \quad v \in D}{([q'_p/p]\theta,\sigma,\rho) \in \widehat{Q} \quad (\theta,\sigma,\rho) \xrightarrow{c?s(v)} ([q'_p/p]\theta,\sigma,\rho)} \tag{3.18}$$

$$\frac{(\theta,\sigma,\rho) \in \widehat{Q} \quad q_p \xrightarrow{[b]\ c?s(\Omega)}_p q'_p \quad \sigma(b) = \mathbf{t} \quad c \in C^{\text{int}} \quad \rho(c) = s(v)\cdot w \quad v \in D}{([q'_p/p]\theta,\sigma,\rho[w/c]) \in \widehat{Q} \quad (\theta,\sigma,\rho) \xrightarrow{\tau} ([q'_p/p]\theta,\sigma,\rho[w/c])} \tag{3.19}$$

$$\frac{(\theta,\sigma,\rho) \in \widehat{Q} \quad q_p \xrightarrow{[b]\ c!s(\oslash)}_p q'_p \quad \sigma(b) = \mathbf{t} \quad c \in C^{\text{ext}}}{([q'_p/p]\theta,\sigma,\rho) \in \widehat{Q} \quad (\theta,\sigma,\rho) \xrightarrow{c!s(\oslash)} ([q'_p/p]\theta,\sigma,\rho)} \tag{3.20}$$

$$\frac{(\theta,\sigma,\rho) \in \widehat{Q} \quad q_p \xrightarrow{[b]\ c!s(\oslash)}_p q'_p \quad \sigma(b) = \mathbf{t} \quad c \in C^{\text{int}} \quad \rho(c) = w}{([q'_p/p]\theta,\sigma,\rho[w\cdot s(\oslash)/c]) \in \widehat{Q} \quad (\theta,\sigma,\rho) \xrightarrow{\tau} ([q'_p/p]\theta,\sigma,\rho[w\cdot s(\oslash)/c])} \tag{3.21}$$

On peut remarquer que la constante spéciale $\oslash$ ne paraît pas dans les contextes de variables (c.à-d. il n'existe aucun état (θ,σ,ρ) et aucune variable x t.q. $\sigma(x) = \oslash$). ▪

Le lemme suivant montre que si on effectue une composition asynchrone de tous les AECs dans une spécification on obtient une nouvelle spécification dont le modèle est bisimilaire avec celui de la spécification initiale :

Lemme 3.2

Soit $SP = (S, C, P)$ une spécification, $P = \{p_1, p_2, \ldots, p_n\}$. Soit r l'automate étendu communicant : $r = p_1 \,|||\, p_2 \,|||\, \ldots \,|||\, p_n$ et $SP' = (S, C, \{r\})$

Alors $\widehat{SP} \sim \widehat{SP'}$. ▪

3.3 Vérification formelle

La vérification formelle peut être appliquée si la structure interne des implémentations est connue. Selon le formalisme utilisé pour représenter les spécifications, on distingue des méthodes de vérification pour les *spécifications logiques* ou pour les *spécifications comportementales*.

3.3.1 Spécifications logiques

Les spécifications logiques décrivent des propriétés système à valider (ex. l'absence du blocage, l'exclusion mutuelle). Une spécification logique est un ensemble de formules d'une *logique temporelle*: linéaire LTL ([Pnu77]), arborescente CTL ([CES86]) ou le μ-calcul arborescent ([Koz83]). Vérifier que IMP `conf` SPEC revient à décider si IMP est un modèle, dans le sens logique, de toute formule $\phi \in$ SPEC : IMP $\models \phi$. Il existe plusieurs approches pour décider si IMP $\models \phi$ ([MOSS99]) : on mentionne l'*approche sémantique* (qui revient à calculer l'ensemble d'états qui satisfont la formule, et donc calculer le point fixe d'un opérateur ayant comme domaine de définition et comme ensemble de valeurs $2^{Q_{\text{IMP}}}$, où Q_{IMP} est l'ensemble d'états de IMP) et l'*approche fondé sur la théorie des automates* (qui revient à construire deux automates : A_{IMP} qui accepte tous les chemins du modèle IMP et A_ϕ qui accepte tous les chemins qui satisfont la formule ϕ. On vérifie ensuite si $L(A_{\text{IMP}}) \subseteq L(A_\phi)$, donc $L(A_{\text{IMP}}) \cap \complement L(A_\phi) = \emptyset$, donc $L(A_{\text{IMP}}) \cap L(A_{\neg\phi}) = \emptyset$ et donc $L(A_{\text{IMP}} \times A_{\neg\phi}) = \emptyset$).

3.3.2 Spécifications comportementales

Les spécifications comportementales décrivent le comportement attendu du système, observé à un certain niveau d'abstraction. Dans ce cas les spécifications sont aussi modélisées par des STEs et vérifier que IMP `conf` SPEC revient à décider si IMP $\sim$ SPEC, avec $\sim$ une relation de bisimulation quelconque.

Ce calcul passe d'abord par la minimisation de IMP qui permet le calcul du quotient IMP$/_\sim$. Le quotient est construit par raffinements successifs d'une partition initiale de Q_{IMP}. Il existe des algorithmes efficaces pour ce calcul (ex. l'algorithme de Paige et Tarjan [PT87] qui a une complexité polynômiale dans la taille du STE initial).

L'étape suivante consiste dans une comparaison entre les STEs IMP$/_\sim$ et SPEC (ou SPEC$/_\sim$ si c'est le cas).

Il faut remarquer que, en pratique, les implémentations et les spécifications sont décrites en utilisant des formalismes de description divers, plus expressifs que les STEs (mais

qui ont une sémantique opérationnelle à base de STEs) : *industriels* (des *techniques de description formelle* comme ESTELLE ([fS88]), LOTOS ([fS89]), SDL ([Sec94]) ou UML ([Gro99])) ou *académiques* (des algèbres de processus comme CSP ([BHR84, Hoa78]) CCS ([Mil80]), MEIJE ([Bou85]), ACP ([BK85])).

3.4 Test

Le test peut être appliqué le cas où la structure interne des implémentations est connue mais surtout dans le cas quand celle-ci n'est pas connue. Dans le premier cas il s'agit du *test boîte blanche* (ou *test structurel*) et dans le deuxième cas il s'agit du *test boîte noire* (ou *test fonctionnel*, ou *test de conformité*).

Dans le cas du test de conformité l'enjeu est de construire un scénario de test pour montrer que IMP $\neg$`conf` SPEC (car, généralement, on ne peut pas déduire, moyennant le test de conformité, la relation IMP `conf` SPEC).

Dans la suite on présente les principales techniques de construction des scénarios de test, techniques groupées en fonctions des formalismes utilisés pour modéliser SPEC et IMP : il s'agit des FSMs[10] (machines à état fini), des STEs (systèmes de transitions étiquetées), des IOSTE (systèmes de transitions à entrée-sortie) et des AECs (automates étendus communicants).

3.4.1 Machines à état fini

Dans ce cas SPEC et IMP sont modélisées par des machines de Mealy déterministes[11]. Le test de conformité des machines de Mealy utilise comme relation de conformité la relation d'isomorphisme entre ce type de machines. Étant donnée SPEC $= (I, O, S, \lambda, \delta)$ on voudrait construire à partir de SPEC une *séquence de vérification* $x \in I^*$ t.q. x distingue SPEC de toute autre machine qui n'est pas isomorphe avec elle.

Dans la littérature concernant le test de conformité des machines à état fini ([Hol91, LY96]) on prèsente certaines contraintes imposées sur SPEC et IMP (en vue de rendre possible la construction d'une séquence de vérification, sinon pour toute SPEC et toute séquence $x \in I^*$ on peut construire IMP t.q. IMP $\neg$`conf` SPEC mais x ne pouvant pas distinguer SPEC de IMP) :

1. En vue de remplacer la relation d'isomorphisme avec la relation d'équivalence on

10. *Finite State Machine* en anglais

11. Toutes les notions théoriques utilisées ensuite et qui concernent les machines de Mealy peuvent être trouvées dans [Koh78].

demande que SPEC soit fortement connexe et minimisée et que le nombre d'états de IMP ne dépasse pas celui de SPEC[12].

2. SPEC et IMP doivent être complètes (les fonctions λ sont complètement définies).

3. Les contraintes suivantes concernent la *testabilité* de SPEC et de ses implémentations. On ajoute les entrées suivantes :

   (a) STATUS : à la réception de STATUS la machine émet l'état courant et reste dans le même état,

   (b) SET : à la réception de SET la machine exécute une transition vers l'état indiqué pas SET,

   (c) RESET : à la réception de RESET la machine exécute une transition vers un état fixé. Le RESET peut être *fiable* ou pas.

L'algorithme *de base*[13] de construction d'une séquence de vérification procède de la manière suivante : on construit des sous-séquences pour chaque transition de SPEC en ensuite on les enchaînent pour obtenir une séquence de vérification.

Dans le cas quand IMP a toutes les propriétés STATUS, RESET, SET la sous-séquence pour la transition $(p, a/\lambda(p,a), \delta(p,a))$ est $\text{RESET} \cdot (\text{SET}, p) \cdot a \cdot \text{STATUS}$.

Remplacement de RESET

L'entrée RESET peut être remplacée par une *séquence de retour*[14]. On construit une séquence de retour ([LY96]) en maintenant une collection de sous-ensembles de S, l'ensemble d'états de SPEC. On commence avec $\{S\}$ et on applique une séquence de séparation[15] pour au moins deux états de $\{S\}$. On calcule l'image de la collection après l'application de la séquence de séparation et on recommence avec un sous-ensemble de cardinal plus grand que 1. Si tous les sous-ensembles ont le cardinal 1 alors la séquence obtenue est une séquence de retour. L'algorithme présenté a une complexité polynômiale.

12. De manière plus générale cette hypothèse signifie que les erreurs de l'IMP proviennent seulement de l'implémentation erronée des fonctions δ et λ.

13. Dans le sens que l'IMP est supposée avoir les propriétés STATUS, RESET, SET et ses variantes diffèrent par le moyen de remplacement d'une ou de plusieurs propriétés manquantes.

14. Une séquence de retour est une séquence d'entrées qui permet l'identification de l'état de la machine après l'application de la séquence.

15. Une séquence d'entrées est une séquence de séparation pour deux états si elle produit deux séquences de sorties différentes

Remplacement de SET

Si IMP n'a pas la propriété SET alors, pour la remplacer, on construit des *tours Eulériens de transitions*[16]. Si SPEC est fortement connexe et symétrique (pour chaque état le nombre des transitions sortantes est égal au celui des transitions entrantes) alors un tour Eulérien de transitions peut être construit ([Hol91]) en deux étapes: d'abord on construit, à partir d'un état quelconque (*la racine*) une arborescence recouvrante[17] et ensuite on construit un tour de transitions à partir de la racine et donnant priorité minimale aux transitions couvertes par l'arborescence. Si SPEC n'est pas symétrique on applique un algorithme qui la rend symétrique ([Hol91]): celui-ci commence par étiqueter chaque état avec un entier positif, négatif ou 0 qui représente la différence entre le nombre des transitions sortantes et le nombre de transitions entrantes. Pour chaque paire d'états $(-,+)$ on construit le plus court chemin entre eux et on duplique ses transitions. On augmente/diminue les étiquettes de la paire et si SPEC n'est pas symétrique on recommence avec une autre paire d'états. Les deux algorithmes présentés ont une complexité polynômiale. Étant donné une séquence d'entrées correspondante à un tour Eulérien de transitions, une séquence de vérification s'obtient en intercalant des entrées RESET.

Remplacement de STATUS

L'entrée STATUS peut être remplacée par:

1. des *séquences de distinction*[18].

   Une séquence de distinction *pré-établie* se construit en utilisant un algorithme de *back-tracking*: supposons-nous que $K \subseteq S$ est un ensemble d'états pour lequel on a pu construire $x_K \in I^*$ une séquence de distinction. On cherche maintenant à construire une séquence de distinction $x_{K\cup\{s\}}$ pour l'ensemble $K\cup\{s\}$, où $s \in S\setminus K$ est un état-candidat. On a plusieurs cas:

   (a) Si x_K reste une séquence de distinction pour $K \cup \{s\}$ alors $x_{K\cup\{s\}} = x_K$.

   (b) L'ensemble $\{s_1 \mid s_1 \in K \wedge \lambda(s_1, x_K) = \lambda(s, x_K)\} \neq \emptyset$. Si on peut choisir $s_1 \in K$ t.q. $\delta(s_1, x_K) \neq \delta(s, x_K)$ alors $x_{K\cup\{s\}} = x_K \cdot ss(\delta(s_1, x_K), \delta(s, x_K))$,

16. Un tour Eulérien de transitions est une séquence de transitions qui commence et finit dans le même état et dont chaque transition apparaît exactement une fois

17. *spanning arborescence* en anglais

18. Une séquences de distinction est une séquence d'entrées qui permet, en fonction de sorties observées, de décider quel est l'état ou elle a été appliquée

où $ss(s, s')$ est une séquence de séparation pour les états s, s' [19]. Sinon, on rejette l'état-candidat s et on choisit un autre.

On peut remarquer que l'algorithme ci-dessus a un complexité temps (dans le pire cas) exponentielle (cela est dû au fait que c'est un algorithme de back-tracking) et aussi le fait que, en effet, c'est possible qu'une machine de Mealy (même minimisée) ne possède pas de séquences de distinction pré-établies (voir l'exemple de la figure ??$_{[\text{p.}??]}$, tiré de [LY94]).

Pour pallier les difficultés liées à la construction des séquences de distinction pré-établies on utilise les séquences de distinction *adaptatives*. Une séquence de distinction adaptative est un arbre de décision avec les nœuds internes étiquetés avec des entrées, les nœuds feuille étiquetés avec des états et les arêtes étiquetées avec des sorties t.q. si un nœud feuille est étiqueté avec $s \in S$ alors on a $y = \lambda(s, x)$, où $y \in O^*$ et $x \in I^*$ sont les séquences de symboles obtenues sur le chemin de la racine vers le nœud s.

Les algorithmes ([LY94, LY96]) de décision de l'existence et de construction des séquences de distinction adaptatives ont une complexité temps (dans le pire cas) polynômiale :

(a) Pour vérifier si la machine $M = (I, O, S, \lambda, \delta)$ possède une séquence de distinction adaptative on utilise l'algorithme suivant : on maintient une partition π de S (initialisée avec $\{S\}$) et tant qu'il existe B bloc de π et $a \in I$ une entrée valide pour B [20] on partage et remplace le bloc B avec des nouveaux sous-blocs de B t.q. deux états $s', s'' \in B$ appartiendront aux même sous-bloc de B si $\lambda(s', a) = \lambda(s'', a)$ et $\delta(s', a)$, $\delta(s'', a)$ appartiennent aux même bloc de π. Si la partition finale contient seulement des blocs de cardinal 1 alors la machine M possède une séquence de distinction adaptative.

(b) Pour construire une séquence de distinction adaptative de M on procède suivant les étapes :

   i. Dans une première étape on raffine la partition et on construit un arbre de raffinement [21] ST. Celui-ci possède une racine et les nœuds internes sont étiquetés avec des sous-ensembles de S (la racine est étiquetée avec $\{S\}$) t.q. l'étiquette d'un nœud interne est l'union de toutes les étiquettes

19. On suppose qu'on a construit préalablement une séquence de séparation pour chaque paire d'états
20. $\exists s', s'' \in B$ t.q. $\lambda(s', a) \neq \lambda(s'', a)$ ou $\delta(s', a)$, $\delta(s'', a)$ appartiennent aux blocs différents de π
21. *Splitting tree* en anglais

des nœuds successeurs. Les arêtes sont étiquetées avec des symboles de O et les nœuds internes ont aussi attachées des séquences de I.
L'algorithme proposé initialise ST avec la racine, étiquetée avec S et initialise aussi la partition π avec $\{S\}$. Ensuite on raffine cette partition (jusqu'à ce-que on obtient une partition discrète) en utilisant les entrées valides de I (et les éventuelles séquences de I attachées aux nœuds). On classifie les entrées valides en trois types (en tenant compte des blocs courants de la partition) et on augmente ST en tenant compte de ce fait.

ii. À partir de l'arbre de raffinement ST (dont les feuilles forment une partition discrète de S) on construit une séquence de distinction adaptative. On maintient deux ensembles (dénotés I et C) d'états initiaux et courants. Initialement $I = C = S$ et tant que $|I| > 1$ on cherche le nœud v de ST de la plus grande profondeur t.q. son étiquette contient I, on applique à I la séquence de I attachée au nœud v et on réactualise les ensembles I et C.

Les algorithmes pour vérifier l'existence des séquences de distinction pour les machines de Mealy déterministe et pour leurs construction on été proposées dans [LY94, LY96]. Pour les machines de Mealy non-déterministes et probabilistes on trouve des algorithmes de construction dans [ACY95].

2. des *séquences* UIO[22].

   Une séquence UIO représente un cas plus général que celui des séquences de distinction (en effet, une séquence de distinction est un séquence UIO pour tous les états de sa machine). Cependant sa construction demeure exponentielle car, comment on montre dans [LY94], vérifier si un état quelconque possède une séquence UIO équivaut au problème d'accessibilité dans un graphe de taille exponentielle : si $(I, O, S, \lambda, \delta)$ est une machine de Mealy alors on construit le graphe orienté $G = (V, A)$, avec les nœuds $V = \{(p, P) \mid p \in S \wedge P \subseteq S\}$ et les arêtes $A = \{(p, P) \xrightarrow{a} (q, Q) \mid (p, P), (q, Q) \in V \wedge a \in I \wedge q = \delta(p, a) \wedge Q = \{\delta(r, a) \mid r \in P \wedge \lambda(r, a) = \lambda(p, a)\}\}$. Maintenant on peut montrer que l'état s a une séquence UIO si et seulement si il existe dans G un chemin de $(s, S \setminus \{s\})$ à un nœud quelconque $(p, \emptyset)$.

Pour les machines ou les entrées SET et STATUS manquent conjointement, mais on dispose d'une séquence de distinction (pré-établie ou adaptative) SD ou bien des séquence

22. Une séquence UIO pour un état s est une séquence d'entrées qui permet, en fonction de sorties observées, de décider si l'état ou elle a été appliquée est s ou n'est pas s

UIO pour tous les états, il faut modifier légèrement les tours Eulériens de transitions (car l'état final de la machine après l'application d'une des séquences ci-dessus pourrait être différent de l'état où la séquence a été appliquée). Plus précisément, on ajoute des transitions *factices* ([Hol91]) : pour toute transition $(p, a/o, q)$ on ajoute la transition factice $(p, a \cdot \text{UIO}(q)/o \cdot \lambda(q, \text{UIO}(q)), \delta(q, \text{UIO}(q)))$ (ou $(p, a \cdot \text{SD}/o \cdot \lambda(q, \text{SD}), \delta(q, \text{SD}))$). Ces transitions factices sont ajoutées après l'éventuelle transformation du graphe initial dans un graphe symétrique. Ensuite, on calcule les tours Eulériens de transitions prenant en compte seulement les transitions factices (ceci est une instance du problème du postier rural chinois).

Séquences caractéristiques

Un autre algorithme (fondé sur les *séquences caractéristiques* et la relation de similarité) a été proposé dans [LY96]. Cet algorithme a une complexité polynômiale.

Une *famille de séparation* pour une machine de Mealy est un collection d'ensembles de séquences, un ensemble pour chaque état, t.q. pour deux états distincts, leurs ensembles contiennent deux séquences (une dans chaque ensemble) avec un préfixe commun (qui est une séquence de séparation pour les deux états). Si les ensembles qui constituent une famille de séparation sont identiques alors leurs éléments s'appellent *séquences caractéristiques* et on l'appelle *ensemble de séparation*.

L'algorithme de construction ([LY96]) d'une famille (un ensemble) de séparation a une complexité temps (le pire cas) polynômiale : on maintient une partition de l'ensemble des états et on la raffine en utilisant les séquences de séparation. L'algorithme s'arrête quand la partition devient discrète.

En utilisant les familles de séparation on peut définir la relation de similarité entre les machines de Mealy : deux machines sont similaires si à chaque état de la deuxième machine correspond un état dans la première tel qu'ils sont similaires (les séquences de sorties obtenues en appliquant dans les deux états les séquences de l'ensemble de séparation du premier état coïncident).

L'algorithme mentionné ci-dessus s'appuie sur l'existence de la propriété RESET (fiable, dans le sens suivant : appliquée dans n'importe quel état on arrive dans le même état, notons-le s_r, *chaque* fois quand on l'applique). Soit $(Z_s)_{s \in S}$ une famille de séparation pour la machine M. On procède par étapes :

1. On construit, par un parcours en largeur d'abord et en partant de s_r, l'arbre BFS qui recouvre M.

2. On construit des sous-séquences pour vérifier la similarité : pour chaque état s de

M et pour chacune de ses séquences caractéristiques x :

(a) on applique RESET,

(b) on applique la séquence d'entrées p_s (de s_r à s dans BFS),

(c) on applique x et on vérifie les sorties.

Si les séquences de sorties coïncident pour chaque état et pour chacune de leurs séquences caractéristiques alors on peut confirmer la similarité.

3. On construit maintenant les séquences correspondant aux transitions qui ne paraissent pas dans l'arbre BFS et ainsi on peut vérifier l'isomorphisme. Soit $(s, a/o, t)$ une telle transition. Pour chaque séquence caractéristique $x \in Z_t$:

   (a) on applique RESET,

   (b) on applique la séquence d'entrées p_s (de s_r à s dans BFS),

   (c) on applique $a \cdot x$

   Si les séquences de sorties coïncident pour chaque transition et pour chaque séquence caractéristique de l'état cible on peut confirmer l'isomorphisme.

Si on dispose d'une séquence de distinction pré-établie x_0 l'algorithme présenté ci-dessus pourrait être simplifié ([LY96]) :

1. Soit $S = \{s_1, \ldots, s_n\}$. On construit la famille de séparation $(Z_i)_{i \in 1..n}$, avec $Z_i = \{x_0\}$, pour tout $i \in 1$ .. n.

2. On construit une séquence d'entrées pour vérifier la similarité : $x_0 \cdot \tau(t_1, s_2) \cdot x_0 \cdot \tau(t_2, s_3) \cdot \ldots \cdot \tau(t_n, s_1) \cdot x_0$, avec $t_i \doteqdot \delta(s_i, x_0)$ et $\tau(s_i, s_j) \in I^+$ une séquence de transfert entre les états s_i et s_j : $\delta(s_i, \tau(s_i, s_j)) = s_j$. Les séquences de transfert entre tous les états existent car M est supposée être fortement connexe (elles sont choisies en fonction du critère du chemin de longueur minime). En supposant que la machine M est dans l'état s_1 alors en appliquant la séquence ci-dessus on devrait obtenir n séquences de sorties distinctes (à la suite de l'application des sous-séquences $\tau(t_i, s_{i+1 mod\ n})$, $i \in 1$ .. n), sinon la similarité n'est pas vérifiée.

3. Ensuite, pour vérifier l'isomorphisme, pour chaque transition $(s_i, a/o, s_j)$ et en supposant que la machine est dans l'état s_k on construit la séquence $y = \tau(t_k, s_{i_1}) \cdot x_0 \cdot \tau(t_{i-1}, s_i) \cdot a \cdot x_0$.

Si on dispose d'une séquence de distinction adaptative T alors on modifie légèrement l'algorithme précédent (en prenant comme famille de séparation $Z_i = \{x_i\}$, $i \in 1 .. n$, où x_i et la séquence d'entrées obtenue dans l'arbre T sur le chemin de racine vers le nœud feuille étiqueté avec l'état s_i).

Conclusion

Les méthodes proposées sont utiles pour la génération de séquences de test de conformité pour les protocoles sans données (ou avec la partie donnée très simple qui n'influence pas le flot de contrôle du protocole). Si on les utilise pour dériver des séquences de test à partir des spécifications de protocoles avec un flot de données assez complexe on pourrait obtenir des séquences de test qui ne sont pas exécutables (c.à-d. non seulement ces séquences ne nous servent pas à réfuter une implémentation non-conforme à la spécification mais elles ne correspondent à aucune séquence dans le modèle de la spécification).

3.4.2 Théories formelles du test de conformité

Les théories formelles du test ont été développées à partir de la relation d'équivalence de test[23] de [DNH84, Hen88] dans le cadre des algèbres de processus. Pour définir cette relation on considère que les processus sont soumis à des expérimentations. Le processus en cause interagit avec l'entité expérimentatrice (éventuellement un autre processus), le système expérimental ainsi obtenu évoluant[24] à partir de la configuration initiale (les états initiaux de processus et de l'entité expérimentatrice) soit à l'infini, soit vers de configurations de blocage (d'où le système expérimental ne peut guère évoluer). On introduit aussi l'ensemble d'états de succès de l'entité expérimentatrice. Ceux-ci sont nécessaires pour établir les résultats des expérimentations. Un calcul se finit avec succès si le système expérimental passe par un état de succès et s'il ne contient aucun état de succès il se finit avec échec. Du fait que le système expérimental peut être non-déterministe on peut avoir plusieurs calculs à partir de la configuration initiale. Deux processus sont *test-équivalents*[25] si pour toute entité expérimentatrice, les deux ensembles de calculs C_1, C_2 obtenus à la suite des interactions des deux processus avec l'entité expérimentatrice ne peuvent être distingués dans le sens suivant : C_1 contient un calcul fini par

23. *testing equivalence* en anglais

24. Cette évolution est appelée *calcul* dans [Hen88]

25. Dans la littérature ([Tre96]) ce type de définition est appelé *définition par extension* (*extensional definition* en anglais), dans l'opposition avec les *définitions spécifiques* (*intensional definition* en anglais) qui ne nécessitent que les processus en cause et pas une entité expérimentatrice.

succès ssi[26] C_2 contient un calcul fini par succès et C_1 ne contient aucun calcul fini par échec ssi[27] C_2 ne contient aucun calcul fini par échec. La relation d'équivalence de test (et plus précisément les relations de préordre sous-jacentes : *may* et *must*) a permis le développement de nombreuses relations de conformité (ou d'implémentation) (cf. ci-dessous).

L'activité de test de conformité est aujourd'hui l'objet d'une norme ISO ([Ray87, fS92b]) qui régit (entre autres) :

1. le domaine d'application : il s'agit de protocoles de communication (organisés dans couches, chaque protocole offrant un point d'accès supérieur et inférieur),
2. la signification de la conformité d'une IUT avec sa spécification.
3. les méthodes de test de conformité (figure 3.2[p.62]) : elles décrivent l'architecture de test. On distingue quatre méthodes :
   (a) la méthode *locale* prévoit que le système sous test SUT[28] est constitué seulement par UT, LT, TCP et IUT. L'IUT peut être contrôlée et observée directement (via les points de contrôle et d'observation PCO).
   (b) la méthode *distribuée* prévoit que SUT est constitué par UT, LT, TCP, IUT mais aussi le fournisseur SERVICE car dans ce cas le point d'accès inférieur de l'IUT n'est pas directement accessible.
   (c) la méthode *coordonnée* est presque similaire à celle distribuée à l'exception du fait que le point d'accès supérieur de l'IUT est optionnel. Il y a aussi une différence entre les rôles de testeurs, le LT étant coordonné par l'UT.
   (d) la méthode *à distance* est similaire à la méthode distribuée mais elle ne prévoit pas de testeur UT.

   Les méthodes présentées dans la figure 3.2[p.62] sont des méthodes à une seule couche[29]. Dans les cas où l'IUT est composée par plusieurs couches ou elle est isolée par plusieurs couches on obtient les méthodes à plusieurs couches[30] et les méthodes embarquées[31].

26. l'implication dans un sens donne la relation de préordre *may*
27. l'implication dans un sens donne la relation de préordre *must*
28. On désigne par ce terme tous les entités qui participent au processus de test : les testeurs supérieur UT et inférieur LT, l'IUT, les procédures de coordination des testeurs TCP, le fournisseur de services SERVICE qui permet la communication (éventuelle) entre l'IUT et le LT
29. *Single Layer* en anglais
30. *Multi Layer* en anglais
31. *Embedded* en anglais

4. Le développement de tests de conformité pour les protocoles de test de conformité, décomposé en plusieurs étapes :

   (a) le processus d'implémentation : en partant d'une spécification d'un protocole on obtient l'implémentation IUT.

   (b) le processus de génération de test : en partant de la même spécification de protocole et en tenant compte aussi de l'architecture de test on obtient une suite abstraite[32] de tests de conformité.

   (c) le processus d'implémentation de tests : en prenant en compte l'IUT et la suite abstraite de tests on obtient une suite exécutable de tests de conformité.

   (d) le processus d'exécution de tests : on applique la suite exécutable de tests de conformité à l'IUT en obtenant un verdict.

5. la notation de test : on utilise le langage TTCN[33].

La formalisation de cette norme a été l'enjeu de plusieurs recherches, dont on peut mentionner ici [Bri88, Tre92, Pha94a, Tre96]. Les recherches mentionnées ci-dessus ont eu pour but d'établir quels sont les modèles formels qu'on utilise (y compris les éventuelles restrictions qu'on appelle *hypothèses*), de définir formellement des relations de conformité entre ces modèles et d'offrir des algorithmes pour la génération de tests de conformité.

Les théories formelles du test de conformité utilisent comme modèle la classe des STEs (ou une sous-classe de celle-ci, la classe des IOSTEs) et elles considèrent que les implémentations et les spécifications peuvent être modélisées par des STEs (IOSTEs). Cette hypothèse faite sur les spécifications et les implémentations s'appelle *l'hypothèse de test.* D'ailleurs on a vu que les STEs sont utilisés pour donner une sémantique opérationnelle aux systèmes (protocoles) et ceci, mais aussi le fait qu'ils permettent le développement des définitions spécifiques de relations de conformité sont parmi les raisons pour lesquelles ils ont été choisis au lieu des algèbres de processus.

On utilise les IOSTEs pour une meilleure modélisation des communications asymétriques, c.à-d. les communications dont on peut distinguer les entrées des sorties. Dans ce cas les sorties du testeur se synchronisent avec les entrées de l'implémentation et vice-versa. On suppose ainsi qu'une implémentation ne peut pas bloquer les entrées mais aussi que ses sorties ne peuvent pas être bloquées par le testeur.

32. à cause du fait qu'on ne prend pas en compte l'IUT
33. Tree and Tabular Combined Notation

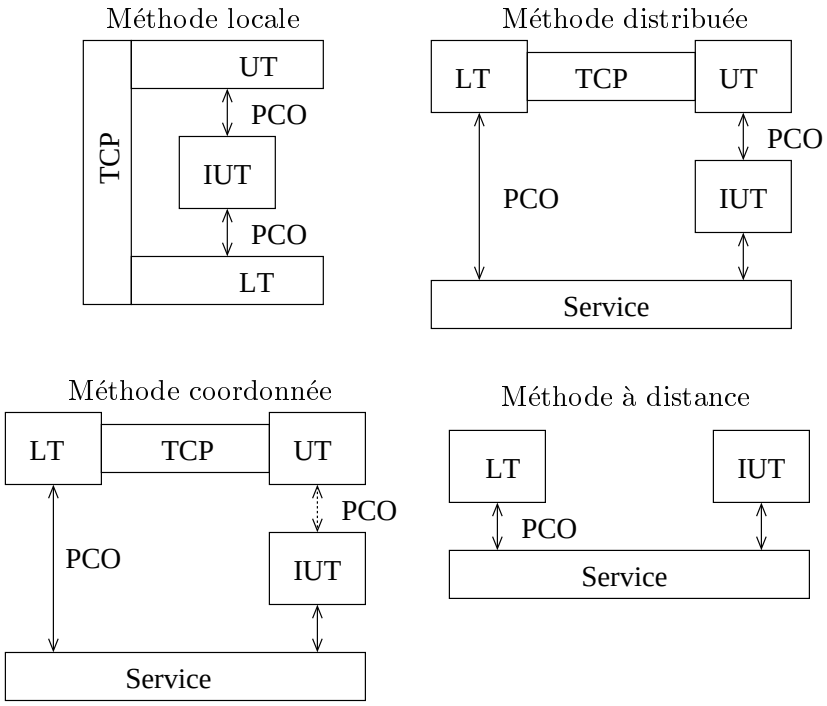

FIG. 3.2 – *Méthodes de test de conformité*

Une relation de conformité (ou d'implémentation) exprime la correction d'une implémentation par rapport à une spécification. Une définition par extension d'une relation de conformité `imp` se fait par rapport aux observations effectuées par un ensemble d'*observateurs* sur l'implémentation et la spécification ([Tre96]) : si $\mathcal{U}$ est l'ensemble d'observateurs alors l'implémentation I implémente correctement S (I `imp` S) si toute observation faite sur I peut être retrouvée dans une observation (faite par le même observateur) sur S : $(\forall u \in \mathcal{U})(obs(u,i) * obs(u,s))$. Dans une algèbre de processus les observations *obs* sont obtenues en utilisant un opérateur de composition (synchrone ou asynchrone) de processus. Dans la suite on se restreint aux définitions spécifiques des relations de conformité.

Les principales relations de conformité qu'on peut trouver dans la littérature sont les suivantes :

Définition 3.13 (Relations de conformité ([Pha94a, Tre96]))
Soit I une implémentation de[34] la spécification S.
Les relations de conformité dans le cas des communications symétriques sont :

1. La relation de *préordre de traces* $\leq_{\mathrm{tr}} \subseteq \mathcal{STE}_\tau(\Sigma) \times \mathcal{STE}_\tau(\Sigma)$ est définie t.q. $I \leq_{\mathrm{tr}} S$ si $trace(I.q_0) \subseteq trace(S.q_0)$.

2. La relation de *préordre de test* $\leq_{\mathrm{te}} \subseteq \mathcal{STE}_\tau(\Sigma) \times \mathcal{STE}_\tau(\Sigma)$ est définie t.q. $I \leq_{\mathrm{te}} S$ si $(\forall \sigma \in \Sigma^*)(\forall A \subseteq L)(I.q_0$ `after` σ `refuses` $A \Rightarrow S.q_0$ `after` σ `refuses` $A)$.

3. La relation CONF ([Bri88]) CONF $\subseteq \mathcal{STE}_\tau(\Sigma) \times \mathcal{STE}_\tau(\Sigma)$ est la restriction de $\leq_{\mathrm{te}}$ aux traces de S : I CONF S si $(\forall \sigma \in trace(S.q_0))(\forall A \subseteq L)(I.q_0$ `after` σ `refuses` $A \Rightarrow$ $S.q_0$ `after` σ `refuses` $A)$.

4. On suppose maintenant que les observateurs peuvent détecter aussi que le processus observé ne peut pas effectuer les actions de Σ qu'on lui propose. Pour obtenir une définition spécifique de la relation de *préordre de refus* on étend d'abord la définition de τ-STEs avec des *transitions de refus*. Celles-ci sont des transitions de type $p \xrightarrow{A} p$, où $A \subseteq \Sigma$ dénote toutes les actions qui peuvent être refusées dans l'état p ($(\forall a \in A \cup \{\tau\})(\neg p \xrightarrow{a})$). Les *traces de refus* de p sont les traces obtenues en considérant aussi les transitions de refus : $ftrace(p) \doteqdot \{\phi \in (\Sigma \cup 2^\Sigma)^* \mid p \stackrel{\phi}{\Longrightarrow}\}$.
   La relation de *préordre de refus* $\leq_{\mathrm{rf}} \subseteq \mathcal{STE}_\tau(\Sigma) \times \mathcal{STE}_\tau(\Sigma)$ est définie t.q. $I \leq_{\mathrm{rf}} S$ si $ftrace(I.q_0) \subseteq ftrace(S.q_0)$.

34. On suppose que S et I ont le même ensemble d'étiquettes : $\Sigma \cup \{\tau\}$ dans le cas des communications symétriques et $\Sigma_I \cup \Sigma_O \cup \{\tau\}$ dans le cas des communications asymétriques

Dans le cas des communications asymétriques l'implémentation et les observateurs sont des IOSTEs tandis que la spécification est un τ-STE ([Tre96]). Dans ces conditions on définit les relations de conformité suivantes :

1. Pour définir la relation de *test à entrée-sortie*[35] $\leq_{\text{iot}} \subseteq \mathcal{IOSTE}(\Sigma_I, \Sigma_O) \times \mathcal{STE}_\tau(\Sigma_I \cup \Sigma_O)$ il faut définir d'abord le concept de *quiescence* (pour les états et les traces).

   Un état p est *de repos*[36] ($\delta(p)$) si $(\forall a \in \Sigma_O \cup \{\tau\})(\neg p \xrightarrow{a})$ et une *trace de repos* σ est une trace qui mène vers un état de repos : $(\exists p' \in p \texttt{ after } \sigma)(\delta(p'))$. Si $qtrace(p)$ est l'ensemble de traces de repos de p alors on peut définir la relation $\leq_{\text{iot}}$ t.q. $I \leq_{\text{iot}} S$ si $trace(I.q_0) \subseteq trace(S.q_0) \wedge qtrace(I.q_0) \subseteq qtrace(S.q_0)$.

   Si $out(p) \doteqdot \{o \in \Sigma_O \mid p \xrightarrow{o}\} \cup \{\delta \mid \delta(p)\}$ on peut fournir une définition équivalente ([Tre96]) : $I \leq_{\text{iot}} S$ si $(\forall \sigma \in (\Sigma_I \cup \Sigma_O)^*)(out(I.q_0 \texttt{ after } \sigma) \subseteq out(S.q_0 \texttt{ after } \sigma))$.

2. De manière similaire aux relations $\leq_{\text{te}}$ et CONF on définit la relation IOCONF $\subseteq \mathcal{IOSTE}(\Sigma_I, \Sigma_O) \times \mathcal{STE}_\tau(\Sigma_I \cup \Sigma_O)$ t.q. elle est la restriction de $\leq_{\text{iot}}$ aux traces de S : I IOCONF S si $(\forall \sigma \in trace(S.q_0))(out(I.q_0 \texttt{ after } \sigma) \subseteq out(S.q_0 \texttt{ after } \sigma))$.

3. On voudrait définir maintenant une relation similaire à la relation de préordre de refus. Pour cela on introduit les *traces de suspension* qui, pour un état p, sont ses traces de refus dans le cas où seulement les actions de Σ_O peuvent être refusées : $strace(p) \doteqdot ftrace(p) \cap (\Sigma_I \cup \Sigma_O \cup \{\Sigma_O\})^*$. Si on remplace Σ_O par δ[37] on obtient $strace(p) \subseteq \Sigma_\delta^*$, où $\Sigma_\delta \doteqdot \Sigma_I \cup \Sigma_O \cup \{\delta\}$.

   On définit la relation $\leq_{\text{ior}} \subseteq \mathcal{IOSTE}(\Sigma_I, \Sigma_O) \times \mathcal{STE}_\tau(\Sigma_I \cup \Sigma_O)$ t.q. $I \leq_{\text{ior}} S$ si $strace(I.q_0) \subseteq strace(S.q_0)$ ou, de manière équivalente, en utilisant les ensembles out, si $(\forall \sigma \in \Sigma_\delta^*)(out(I.q_0 \texttt{ after } \sigma) \subseteq out(S.q_0 \texttt{ after } \sigma))$.

4. De manière similaire aux relations $\leq_{\text{te}}$, CONF et $\leq_{\text{iot}}$, IOCONF on restreint la relation $\leq_{\text{ior}}$ aux traces de suspension de S et on obtient la relation IOCO $\subseteq \mathcal{IOSTE}(\Sigma_I, \Sigma_O) \times \mathcal{STE}_\tau(\Sigma_I \cup \Sigma_O)$: I IOCO S si $(\forall \sigma \in strace(S.q_0))(out(I.q_0 \texttt{ after } \sigma) \subseteq out(S.q_0 \texttt{ after } \sigma))$.

Dans [Pha94a] on définit plusieurs relations de conformité (celles-ci utilisent une définition plus simple des ensembles out : $out(p) \doteqdot \{o \in \Sigma_O \mid p \xrightarrow{o}\}$) :

1. La relation $R_1 \subseteq \mathcal{IOSTE}(\Sigma_I, \Sigma_O) \times \mathcal{IOSTE}(\Sigma_I, \Sigma_O)$ est définie t.q. $I \; R_1 \; S$ si $(\forall \sigma \in trace(S.q_0))(\sigma \in trace(I.q_0) \Rightarrow out(I.q_0 \texttt{ after } \sigma) \subseteq out(S.q_0 \texttt{ after } \sigma))$.

35. *input-output testing* en anglais
36. *quiescent state* en anglais
37. δ est la notation pour les actions qui indiquent les états de repos

2. La relation $R_2 \subseteq \mathcal{IOSTE}(\Sigma_I, \Sigma_O) \times \mathcal{IOSTE}(\Sigma_I, \Sigma_O)$ raffine la relation R_1 en prenant en compte l'absence des actions de sortie : $I\ R_2\ S$ si $(\forall \sigma \in trace(S.q_0))(\sigma \in trace(I.q_0) \Rightarrow (out(I.q_0\ \texttt{after}\ \sigma) \subseteq out(S.q_0\ \texttt{after}\ \sigma) \wedge out(I.q_0\ \texttt{after}\ \sigma) = \emptyset \iff out(S.q_0\ \texttt{after}\ \sigma) = \emptyset))$. Cette relation est similaire à la relation IOCO.

3. La relation $R_3 \subseteq \mathcal{IOSTE}(\Sigma_I, \Sigma_O) \times \mathcal{IOSTE}(\Sigma_I, \Sigma_O)$ est définie t.q. $I\ R_3\ S$ si $trace(S.q_0) \subseteq trace(I.q_0)$.

4. La relation $R_4 \subseteq \mathcal{IOSTE}(\Sigma_I, \Sigma_O) \times \mathcal{IOSTE}(\Sigma_I, \Sigma_O)$ prend en compte l'absence des actions de sortie : $I\ R_4\ S$ si $trace(S.q_0) \subseteq trace(I.q_0) \wedge (\forall \sigma \in trace(S.q_0))$ $(out(I.q_0\ \texttt{after}\ \sigma) = \emptyset \iff out(S.q_0\ \texttt{after}\ \sigma) = \emptyset)$. Les relations R_3 et R_4 définissent comme conforme une implémentation dont le comportement inclut celui de la spécification.

5. La relation $R_5 \subseteq \mathcal{IOSTE}(\Sigma_I, \Sigma_O) \times \mathcal{IOSTE}(\Sigma_I, \Sigma_O)$ est l'intersection des relations R_2 et R_4 : $I\ R_5\ S$ si $(\forall \sigma \in trace(S.q_0))(\sigma \in trace(I.q_0) \wedge (out(I.q_0\ \texttt{after}\ \sigma) = out(S.q_0\ \texttt{after}\ \sigma)))$.

6. La relation d'implémentation finie $R_\Upsilon \subseteq \mathcal{IOSTE}(\Sigma_I, \Sigma_O) \times \mathcal{IOSTE}(\Sigma_I, \Sigma_O)$, où $\Upsilon \subseteq \{\Sigma_I \cup \Sigma_O\}^*$, est définie t.q. : $I\ R_\Upsilon\ S$ si $(\forall \sigma \in \Upsilon)(trace(S.q_0) \cap trace(I.q_0) \Rightarrow (out(I.q_0\ \texttt{after}\ \sigma) = out(S.q_0\ \texttt{after}\ \sigma)))$.

■

Pour les relations de conformité des communications symétriques/asymétriques on peut établir les résultats suivants ([Tre96]) :

Proposition 3.1 (Communications symétriques)
$\leq_{\text{rf}} \subset \leq_{\text{te}} = \leq_{\text{tr}} \cap \text{CONF}$. ■
Preuve
Dans [Tre96].

□

Proposition 3.2 (Communications asymétriques)
$\leq_{\text{ior}} \subset \leq_{\text{iot}} \subset \text{IOCONF}$ et $\leq_{\text{ior}} \subset \text{IOCO} \subset \text{IOCONF}$. ■
Preuve
Dans [Tre96] : on utilise les définitions qui utilisent les ensembles *out* et on observe qu'on peut remplacer les quatre définitions avec une seule définition paramétrée par l'ensemble $\mathcal{F}$: $I\ \text{IOCO}_{\mathcal{F}}\ S$ si $(\forall \sigma \in \mathcal{F})(out(I.q_0\ \texttt{after}\ \sigma) \subseteq out(S.q_0\ \texttt{after}\ \sigma))$ où $\mathcal{F} \in \{(\Sigma_I \cup \Sigma_O)^*, (\Sigma_I \cup \Sigma_O \cup \{\delta\})^*, trace(S.q_0), strace(S.q_0)\}$.

□

Pour les relations mentionnées ci-dessus on a conçu (voir ci-dessous) des algorithmes qui, pour une spécification quelconque et pour une relation de conformité donnée, calculent des cas de test (avec un comportement fini, c.à-d. il n'ont pas de trace de longueur infinie) qui forment une suite de test (éventuellement infinie) *valide*[38] et *exhaustive*. Si *apply* est une fonction qui rend le verdict (`pass`, `fail`, `inconclusive`) correspondant à l'application[39] du cas de test t à une implémentation quelconque I on peut définir la relation *passes* t.q. I *passes* $t \Leftrightarrow$ *apply*$(t, I) \neq$ `fail`. On étend cette relation a une suite de cas de test Π : I *passes* $\Pi \Leftrightarrow (\forall t)(I$ *passes* $t)$. La suite Π est *valide* si $(\forall I)(I$ `imp` $S \Rightarrow I$ *passes* $\Pi)$ et *exhaustive* si $(\forall I)(I$ `imp` $S \Leftarrow I$ *passes* $\Pi)$.

Pour la relation CONF un algorithme non-déterministe et récursif a été proposé dans [Bri88, Tre92]. Cet algorithme fournit un ensemble d'observateurs (une suite de test valide et exhaustive) pour cette relation, ensemble appelé *testeur canonique* (et qui est modélisé par un STE). Le testeur canonique a plusieurs propriétés importantes (comme par exemple le fait que deux testeurs canoniques construits par l'algorithme pour une même spécification sont équivalents de point de vue test).

Pour les relations de conformité des communications asymétriques (dont on considère la relation générale IOCO$_{\mathcal{F}}$) dans [Tre96] on propose un algorithme non-déterministe et récursif. On utilise, au lieu de IOSTEs, des automates de suspension (ceux-ci sont des STEs déterministes, obtenus à partir des IOSTEs originaux auxquels on ajoute explicitement les transitions étiquetées avec δ). Si p est un IOSTE et Γ_p est l'automate de suspension correspondant alors pour toute trace σ les ensembles *out* coïncident : $p.q_0$ `after` $\sigma = \Gamma_p.q_0$ `after` σ. Cette propriété permet l'utilisation des automates de suspension (plus précisément leur traces) pour obtenir des cas de test : si Γ est un automate de suspension et $\mathcal{F}$ l'ensemble de traces de la relation de conformité alors on dispose de trois choix non-déterministes :

1. Soit on arrête l'algorithme (et ainsi la construction du cas de test) en arrivant dans l'état `pass`.

2. Soit on choisit de fournir une entrée a à l'implémentation et dans ce cas le cas de

38. *sound* en anglais

39. Cela revient à l'existence d'une trace dans le STE (IOSTE) obtenu par la composition (synchrone ou asynchrone) entre les STEs (IOSTEs) qui modélisent l'implémentation et le cas de test, trace qui conduit vers un état spécial du cas de test, état qui signifie que la trace correspondante de l'implémentation se trouve (ou pas) dans la spécification. Le verdict associé à l'application du cas de test est `fail` s'il existe une trace qui ne se trouve pas parmi celles de la spécification. Le verdict `pass` est donné dans le cas contraire et celui d'`inconclusive` est utilisé dans certains outils de test, notamment TGV, principalement à cause du fait qu'ils génèrent des cas de test seulement pour une partie de la spécification et on ne s'intéresse pas dans les traces qui n'appartient pas à cette partie de la spécification

test est obtenu par la juxtaposition de la transition étiquetée par a et un sous-cas de test obtenu par l'appel récursif de cette procédure avec comme paramètres l'automate Γ' t.q. $\Gamma \xrightarrow{a} \Gamma'$ et l'ensemble de traces $\mathcal{F}'$ constitué par toutes les traces σ t.q. $a \cdot \sigma \in \mathcal{F}$. Une condition préalable pour appliquer cette règle est que l'ensemble $\mathcal{F}' \neq \emptyset$.

3. Soit on examine chaque sortie x de l'implémentation :

   (a) on arrive l'état `fail` par x si la sortie x n'est pas prévu par la spécification et on pourrait arriver dans l'état courant par une trace de $\mathcal{F}$ (il s'agit ici de $\mathcal{F}$ initial),

   (b) on arrive l'état `pass` par x si la sortie x n'est pas prévu par la spécification mais à la différence de cas précédent on ne peut pas arriver dans l'état courant par une trace de $\mathcal{F}$ (initial),

   (c) la sortie x est prévue dans la spécification et cette partie du cas de test est obtenue par la juxtaposition de la transition étiquetée par x et un sous-cas de test obtenu par l'appel récursif de cette procédure avec comme paramètres l'automate Γ' t.q. $\Gamma \xrightarrow{x} \Gamma'$ et l'ensemble de traces $\mathcal{F}'$ constitué par toutes les traces σ t.q. $x \cdot \sigma \in \mathcal{F}$. On n'impose pas la condition $\mathcal{F}' \neq \emptyset$.

L'outil TORX - [TB99, FGM00] utilise cet algorithme pour dériver des cas de test.

Les relations de conformité $R_{1,2,3,4,5}$ sont utilisées dans l'outil TVEDA ([Pha94a]). L'algorithme de dérivation des cas de test construit un testeur canonique pour chaque relation. Pour la relation R_1 un testeur *canonique asynchrone* est une image miroir de l'automate des traces du IOSTE qui modélise la spécification. L'automate des traces est obtenu à partir de IOSTE de la spécification par une construction similaire à la détermination d'un automate non-déterministe ([HU79]). On lui ajoute un état spécial (`fail`), on inverse la signification des actions (les actions d'entrée deviennent les actions de sortie correspondantes et vice-versa), et on ajoute les transitions étiquetées par une action d'entrée ? a à partir de tout état s vers l'état `fail` si dans l'automate de traces initial il n'y avait aucune transition étiquetée ! a sortante de s.

Quoique, initialement, l'outil AUTOLINK ([SEG+98]) n'avait pas une relation de conformité explicitement attachée, on a reconstruit ultérieurement ([Gog01]) une relation de conformité et on a établi qu'elle se situe entre UIO et la relation IOCONF (comme pouvoir de détection).

Les testeurs canoniques générés par les algorithmes ci-dessus sont des suites complètes (valides et exhaustives) de cas de test et prennent en compte toute la spécification. Parfois on s'intéresse seulement à une partie de la spécification et donc on voudrait générer seulement une partie d'une suite complète de cas de test. Pour résoudre ce problème les chercheurs ont utilisé une technique issue du domaine de la vérification par modèles[40]. Il s'agit de l'approche fondé sur la théorie des automates : pour décider si IMP $\models \phi$ on vérifie si $L(A_{\text{IMP}} \times A_{\neg\phi}) = \emptyset$ (et construire éventuellement un contre-exemple qui est une trace de $L(A_{\text{IMP}} \times A_{\neg\phi})$). Pour capturer les traces de la spécification qui nous intéressent on utilise les objectifs de test. Un objectif de test peut être vu comme une propriété dans le cas de la vérification. Si on cherche à obtenir un contre-exemple pour la négation de cette propriété on trouve une trace de la spécification qui doit appartenir aussi à l'implémentation. Ceci est, plus ou moins, le principe général de tous les outils de test fondés sur les techniques de la vérification formelle ([CSE96, FJJV97, FJJV96, EFM97, GH99]). Dans la section 3.4.3$_{[p.68]}$ on décrit le principe de TGV ([FJJV97, FJJV96]).

3.4.3 Test de conformité fondé sur la vérification - TGV

L'outil de génération de cas de test TGV ([FJJV97, FJJV96, JM97, JM99]) suit la méthodologie décrite dans la norme [fS92b] (et aussi celle employée par les experts du test). Cette méthodologie est basée sur les étapes suivantes :

1. identification d'une architecture de test, qui est une description de l'environnement dans lequel l'implémentation est testée,
2. écriture, en langage naturel, d'objectifs de test, qui permettent de tester un aspect particulier de l'implémentation,
3. dérivation manuelle des tests (sous forme TTCN [fS92a]).

Dans la suite on présente les principaux concepts de TGV :

1. les modèles,
2. la conformité d'une implémentation à une spécification.
3. l'architecture de test,
4. la notion de cohérence entre la spécification et l'objectif de test,
5. les cas (arbres) de test (décorés par des verdicts et par des temporisateurs[41]).

40. *model-checking* en anglais
41. *timers* en anglais

Les modèles

Les modèles utilisés pour la spécification, l'implémentation, l'objectif de test et le cas de test sont des IOSTEs. De plus, les IOSTEs de l'implémentation et du cas de test satisfont la condition de contrôlabilité. On suppose, pour simplifier, que l'implémentation communique directement avec l'environnement de test (le *testeur*). La communication est asymétrique : si on contrôle les sorties du testeur, on ne peut qu'observer, en retour, les sorties de l'implémentation. Les entrées de l'implémentation correspondent aux sorties du testeur et sont appelées *actions contrôlables*. Inversement, les sorties de l'implémentation correspondent aux entrées du testeur et sont appelées *actions observables*. Pour observer la présence ou l'absence d'interblocage ou de boucles, on associe à chaque action observable, un temporisateur qui est géré par le testeur.

La spécification $S \in \mathcal{IOSTE}(\Sigma_O, \Sigma_I)$ est modélisée par le IOSTE déterministe $(Q_{\mathrm{S}}, \Sigma, \{\xrightarrow{a}_{\mathrm{S}} |\ a \in \Sigma\}, q_0^{\mathrm{S}})$, où $\Sigma \doteqdot \Sigma_I \cup \Sigma_O$ et $\Sigma \ni \delta$, une action spéciale qui dénote les blocages de sortie (comme ci-dessus).

Ces spécifications sont obtenues à la suite de quelques modifications imposées aux spécifications initiaux : il s'agit d'abord d'une opération d'inversion entre les alphabets Σ_I et Σ_O t.q. les spécifications passent de $\mathcal{IOSTE}(\Sigma_I, \Sigma_O)$ à $\mathcal{IOSTE}(\Sigma_O, \Sigma_I)$, ensuite d'un remplacement par τ des actions qui ne sont pas observables par l'environnement suivi par une τ-réduction qui élimine les transitions étiquetées par τ. On ajoute ensuite les transition étiquetées par δ (pour les blocages de sortie).

L'IUT $I \in \mathcal{IOSTE}(\Sigma_O, \Sigma_I)$ est modélisée par le IOSTE $(Q_{\mathrm{I}}, \Sigma, \{\xrightarrow{a}_{\mathrm{I}} |\ a \in \Sigma\}, q_0^{\mathrm{I}})$.

Les objectifs de test sont des IOSTEs (de $\mathcal{IOSTE}(\Sigma_I, \Sigma_O)$) déterministes, acycliques et qui satisfont la condition de contrôlabilité. Ils ont un ensemble d'états accepteurs, noté `Accept`. Un état accepteur n'a pas de successeur. On note $TP = (Q_{\mathrm{TP}}, \Sigma, \{\xrightarrow{a}_{\mathrm{TP}} |\ a \in \Sigma\}, q_0^{\mathrm{TP}})$, le IOSTE associé à l'objectif de test. L'objectif de test est défini par le couple formé de TP et de `Accept`, où `Accept` $\subseteq Q_{\mathrm{TP}}$.

La conformité d'une implémentation à une spécification

La relation de conformité choisie est la relation IOCO (de la définition 3.13[p.63]).

Architecture de test

L'architecture de test est celle proposée par la méthode locale : un testeur interagit avec l'IUT via des PCOs. L'architecture de test influence la génération de test et peut

conduire à l'entrelacement de séquences d'actions observables, observées sur des PCOs distincts. L'architecture de test est prise en compte en transformant la spécification. D'abord, pour éviter la concurrence entre les entrées et les sorties on considère que l'environnement agit de manière raisonnable[42] : chaque fois que l'environnement envoie un message à l'IUT il attend jusqu'à ce que il reçoit tous les sorties prévues dans la spécification. Ensuite toute séquence de sorties dans une même transition est transformée dans les losanges qui modélisent la concurrence.

La cohérence entre la spécification et l'objectif de test

Un objectif de test est un critère pour sélectionner un comportement de la spécification, en vue de générer des cas de test. Il faut donc définir une relation de cohérence entre l'objectif de test et la spécification. Elle permettra de dériver des cas de test valides (dans le sens où un verdict `Fail` correspond à une IUT non conforme).

La relation de cohérence est définie de telle sorte que :

1. l'ensemble des comportements, défini par l'objectif de test, doit être inclus d'une certaine manière dans l'ensemble des comportements de la spécification,

2. de tout état de la spécification en relation avec un état d'acceptation, il existe une séquence conduisant à l'état initial de la spécification.

La définition de la relation de cohérence ressemble à celle d'une relation de simulation :

Définition 3.14 (Relation de cohérence)
Soient Q_{TP} l'ensemble des états de l'objectif de test et Q_{S} l'ensemble des états de la spécification. Une relation $R \subseteq Q_{\mathrm{TP}} \times Q_{\mathrm{S}}$ est une *relation de cohérence* si $R \subseteq \mathcal{F}(R)$ où :

$$
\begin{array}{lll}
\mathcal{F}(R) = \{(p^{\mathrm{TP}}, p^{\mathrm{S}}) \mid & & \\
p^{\mathrm{TP}} \xrightarrow{\alpha}_{\mathrm{TP}} q^{\mathrm{TP}} & \Longrightarrow & \exists q^{\mathrm{S}}, q_1^{\mathrm{S}}, \exists \sigma \in (\Sigma \setminus \{\alpha\})^* \cdot p^{\mathrm{S}} \xrightarrow{\sigma}_{\mathrm{S}} q_1^{\mathrm{S}} \xrightarrow{\alpha}_{\mathrm{S}} q^{\mathrm{S}} \wedge \\
 & & (q^{\mathrm{TP}}, q^{\mathrm{S}}) \in R \wedge (p^{\mathrm{TP}}, q_1^{\mathrm{S}}) \in R \wedge \\
p^{\mathrm{TP}} \in \mathtt{Accept} & \Longrightarrow & \exists \sigma \in \Sigma^* \cdot p^{\mathrm{S}} \xrightarrow{\sigma}_{\mathrm{S}} q_0^{\mathrm{S}}\}
\end{array}
$$

L'objectif de test est cohérent vis-à-vis de la spécification s'il existe une relation $R \subseteq \mathcal{F}(R)$, vérifiant $(q_0^{\mathrm{TP}}, q_0^{\mathrm{S}}) \in R$. ∎

42. *Reasonable Environment Hypothesis* en anglais

Les cas de test

Un arbre de test est un IOSTE déterministe, acyclique et qui satisfait la condition de contrôlabilité. Il est construit à partir de la spécification et de l'objectif de test, si la cohérence est vérifiée. L'arbre de test est construit par étapes :

1. La construction du produit synchrone,
2. La vérification de la condition de contrôlabilité et la synthèse de l'arbre de test, à partir du produit synchrone,
3. La décoration de certaines transitions par des verdicts et des temporisateurs.

Le produit synchrone entre l'objectif de test et la spécification est un IOSTE construit conformément à l'algorithme suivant :

Algorithme 1 (Le produit synchrone [FJJV96])

`Entrée :`
L'objectif de test TP et la spécification S.

`Sortie :`
Le produit synchrone $P = (Q_{\mathrm{P}}, \Sigma, \{\xrightarrow{a}_{\mathrm{P}} | \ a \in \Sigma\}, q_0^{\mathrm{P}})$.

`Méthode :`
Les ensembles $Q_{\mathrm{P}} \subseteq Q_{\mathrm{TP}} \times Q_{\mathrm{S}}$ et $\{\xrightarrow{a}_{\mathrm{P}} | \ a \in \Sigma\}$ se construisent conformément aux règles suivantes :

$$\frac{}{q_0^{\mathrm{P}} = (q_0^{\mathrm{TP}}, q_0^{\mathrm{S}}) \in Q_{\mathrm{P}}} \tag{3.22}$$

$$\frac{(p^{\mathrm{TP}}, p^{\mathrm{S}}) \in Q_{\mathrm{P}} \quad p^{\mathrm{TP}} \xrightarrow{\alpha}_{\mathrm{TP}} q^{\mathrm{TP}} \quad p^{\mathrm{S}} \xrightarrow{\alpha}_{\mathrm{S}} q^{\mathrm{S}}}{(q^{\mathrm{TP}}, q^{\mathrm{S}}) \in Q_{\mathrm{P}} \quad (p^{\mathrm{TP}}, p^{\mathrm{S}}) \xrightarrow{\alpha}_{\mathrm{P}} (q^{\mathrm{TP}}, q^{\mathrm{S}})} \tag{3.23}$$

$$\frac{(p^{\mathrm{TP}}, p^{\mathrm{S}}) \in Q_{\mathrm{P}} \quad \neg(p^{\mathrm{TP}} \xrightarrow{\alpha}_{\mathrm{TP}}) \quad p^{\mathrm{S}} \xrightarrow{\alpha}_{\mathrm{S}} q^{\mathrm{S}} \quad (p^{\mathrm{TP}}, p^{\mathrm{S}}) \notin \mathtt{Accept} \times \{q_0^{\mathrm{S}}\}}{(p^{\mathrm{TP}}, q^{\mathrm{S}}) \in Q_{\mathrm{P}} \quad (p^{\mathrm{TP}}, p^{\mathrm{S}}) \xrightarrow{\alpha}_{\mathrm{P}} (p^{\mathrm{TP}}, q^{\mathrm{S}})} \tag{3.24}$$

▪

La synthèse de l'arbre de test se fait à partir des feuilles $(q^{\mathrm{TP}}, q_0^{\mathrm{S}})$, avec $q^{\mathrm{TP}} \in \mathtt{Accept}$ par un algorithme linéaire, qui parcourt en profondeur d'abord le produit synchrone P entre l'objectif de test et la spécification et qui vérifie la cohérence pendant le parcours :

Algorithme 2 (Synthèse arbre de test [FJJV96])

`Entrée :`

Le produit synchrone P entre l'objectif de test et la spécification.

`Sortie :`

L'arbre de test, qui est un sous-graphe orienté, acyclique de P.

`Méthode :`

On définit syntaxiquement les DAGs[43] comme $n :: 0 \mid 1 \mid \alpha; n \mid n + n$. On utilise aussi le prédicat $\vdash^v (p^{\mathrm{TP}}, p^{\mathrm{S}}) : n$, où $v \in \{1, 2, 3\}$ avec la signification suivante : le nœud associé avec $(p^{\mathrm{TP}}, p^{\mathrm{S}})$ est n et il appartient au préambule, au corp de l'arbre de test ou au postambule si v est 1 ou 2 ou 3. On utilise aussi la notation $Node((p^{\mathrm{TP}}, p^{\mathrm{S}}))$ pour indiquer le nœud couramment associé avec $(p^{\mathrm{TP}}, p^{\mathrm{S}})$. Initialement $Node((p^{\mathrm{TP}}, p^{\mathrm{S}})) = 0$. L'opérateur *Comb* est utilisé dans la synthèse pour s'assurer de la cohérence :

$$Comb(m, \alpha; n) := \begin{cases} \alpha; n & \text{si } \alpha \in \Sigma_O,\ m = \sum\limits_{\substack{i \in I \\ \alpha_i \in \Sigma_I}} \alpha_i; n_i \\ m & \text{si } m = \alpha'; n' \text{ et } \alpha' \in \Sigma_O \\ m + \alpha; n & \text{sinon} \end{cases}$$

Ensuite on synthèse l'arbre de test en appliquant les règles suivantes ($3.25_{[\text{p.72}]}$ régit la synthèse du préambule, $3.26_{[\text{p.72}]}$, $3.27_{[\text{p.72}]}$, $3.28_{[\text{p.72}]}$ régissent la synthèse du corp de l'arbre de test et $3.29_{[\text{p.72}]}$, $3.30_{[\text{p.73}]}$ régissent la synthèse du postambule) :

$$\frac{\vdash^v (q_0^{\mathrm{TP}}, q^{\mathrm{S}}) : n \quad v \in \{1, 2\} \quad (q_0^{\mathrm{TP}}, p^{\mathrm{S}}) \xrightarrow{\alpha}_{\mathrm{P}} (q_0^{\mathrm{TP}}, q^{\mathrm{S}})}{\vdash^1 (q_0^{\mathrm{TP}}, p^{\mathrm{S}}) : Comb(Node((q_0^{\mathrm{TP}}, p^{\mathrm{S}})), \alpha; n)} \tag{3.25}$$

$$\frac{\vdash^2 (q^{\mathrm{TP}}, q^{\mathrm{S}}) : n \quad (q_0^{\mathrm{TP}}, p^{\mathrm{S}}) \xrightarrow{\alpha}_{\mathrm{P}} (q^{\mathrm{TP}}, q^{\mathrm{S}}) \quad q^{\mathrm{TP}} \neq q_0^{\mathrm{TP}}}{\vdash^2 (q_0^{\mathrm{TP}}, p^{\mathrm{S}}) : Comb(Node((q_0^{\mathrm{TP}}, p^{\mathrm{S}})), \alpha; n)} \tag{3.26}$$

$$\frac{\vdash^2 (q^{\mathrm{TP}}, q^{\mathrm{S}}) : n \quad (p^{\mathrm{TP}}, p^{\mathrm{S}}) \xrightarrow{\alpha}_{\mathrm{P}} (q^{\mathrm{TP}}, q^{\mathrm{S}}) \quad p^{\mathrm{TP}} \neq q_0^{\mathrm{TP}}}{\vdash^2 (p^{\mathrm{TP}}, p^{\mathrm{S}}) : Comb(Node((p^{\mathrm{TP}}, p^{\mathrm{S}})), \alpha; n)} \tag{3.27}$$

$$\frac{\vdash^3 (q^{\mathrm{TP}}, q^{\mathrm{S}}) : n \quad (p^{\mathrm{TP}}, p^{\mathrm{S}}) \xrightarrow{\alpha}_{\mathrm{P}} (q^{\mathrm{TP}}, q^{\mathrm{S}}) \quad p^{\mathrm{TP}} \notin \texttt{Accept} \quad q^{\mathrm{TP}} \in \texttt{Accept}}{\vdash^2 (p^{\mathrm{TP}}, p^{\mathrm{S}}) : Comb(Node((p^{\mathrm{TP}}, p^{\mathrm{S}})), \alpha; n)} \tag{3.28}$$

$$\frac{p^{\mathrm{TP}} \in \texttt{Accept}}{\vdash^3 (p^{\mathrm{TP}}, q_0^{\mathrm{S}}) : 1} \tag{3.29}$$

43. *Direct Acyclic Graphs* en anglais

$$\frac{\vdash^3 (p^{\mathrm{TP}}, q^{\mathrm{S}}) : n \quad p^{\mathrm{TP}} \in \texttt{Accept} \quad (p^{\mathrm{TP}}, p^{\mathrm{S}}) \xrightarrow{\alpha}_{\mathrm{P}} (p^{\mathrm{TP}}, q^{\mathrm{S}})}{\vdash^3 (p^{\mathrm{TP}}, p^{\mathrm{S}}) : \mathit{Comb}(\mathit{Node}((p^{\mathrm{TP}}, p^{\mathrm{S}})), \alpha; n)} \tag{3.30}$$

▪

La décoration de certaines transitions par des verdicts et des temporisateurs se fait en utilisant les deux algorithmes suivants :

Algorithme 3 (Les verdicts [FJJV96])

`Entrée :`

L'arbre de test fourni par l'algorithme $2_{[\mathrm{p.72}]}$.

`Sortie :`

L'arbre de test avec des verdicts associés aux certaines transitions.

`Méthode :`

La fonction partielle *verdict* associe des verdicts aux transitions de l'arbre de test. Ceci est fait en utilisant les règles $3.31_{[\mathrm{p.73}]}$, $3.32_{[\mathrm{p.73}]}$, $3.33_{[\mathrm{p.74}]}$ et $3.34_{[\mathrm{p.74}]}$.

Le prédicat $\vdash^v$ est modifié t.q. v peut prendre aussi les valeurs 4 et 5 avec la signification suivante : $\vdash^4 (p^{\mathrm{TP}}, p^{\mathrm{S}}) : 1$ signifie que le nœud associé avec $(p^{\mathrm{TP}}, p^{\mathrm{S}})$ est 1 et qu'il est l'état final d'une transition étiquetée par le verdict `Inconclusive`. On ajoute un nouveau état $\perp$ à Q_{P} t.q. $\vdash^5 \perp : 1$.

La signification des verdicts est la suivante :

1. Le verdict `Pass` est associé à une transition si l'état final de la transition appartient au postambule et la spécification est dans l'état initial :

$$\frac{\vdash^v (p^{\mathrm{TP}}, p^{\mathrm{S}}) : n \quad v \in \{2, 3\} \quad \vdash^3 (q^{\mathrm{TP}}, q_0^{\mathrm{S}}) : 1 \quad (p^{\mathrm{TP}}, p^{\mathrm{S}}) \xrightarrow{\alpha}_{\mathrm{P}} (q^{\mathrm{TP}}, q_0^{\mathrm{S}})}{\mathit{verdict}(n, \alpha, 1) = \texttt{Pass}} \tag{3.31}$$

2. Le verdict `(Pass)` est associé à une transition dont l'état initial appartient au corps de l'arbre de test et l'état final au postambule :

$$\frac{\vdash^2 (p^{\mathrm{TP}}, p^{\mathrm{S}}) : m \quad \vdash^3 (q^{\mathrm{TP}}, q^{\mathrm{S}}) : n \quad (p^{\mathrm{TP}}, p^{\mathrm{S}}) \xrightarrow{\alpha}_{\mathrm{P}} (q^{\mathrm{TP}}, q^{\mathrm{S}})}{\mathit{verdict}(m, \alpha, n) = \texttt{(Pass)}} \tag{3.32}$$

3. Le verdict `Inconclusive` signifie qu'une entrée du testeur, correspondant à une sortie de la spécification, n'est pas décrite dans l'arbre de test ou ne conduit pas à un verdict `(Pass)`. On rajoute, dans l'arbre de test, une nouvelle transition,

entrante de l'état $\perp$, étiquetée par l'entrée correspondante, et on lui associe le verdict `Inconclusive` :

$$\frac{\vdash^{v} (p^{\mathrm{TP}}, p^{\mathrm{S}}) : n \quad \not\vdash^{u} (q^{\mathrm{TP}}, q^{\mathrm{S}}) \quad u, v \in \{1,2,3\} \quad (p^{\mathrm{TP}}, p^{\mathrm{S}}) \xrightarrow{i}_{\mathrm{P}} (q^{\mathrm{TP}}, q^{\mathrm{S}}) \quad i \in \Sigma_I}{\vdash^{4} (q^{\mathrm{TP}}, q^{\mathrm{S}}) : 1 \quad (n, i, 1) \in \mathrm{DAG} \quad \mathit{verdict}(n, i, 1) = \texttt{Inconclusive}} \tag{3.33}$$

4. Le verdict `Fail` correspond à toute transition étiquetée par une entrée du testeur, qui ne correspond pas à une sortie de la spécification. Cette transition correspond à la transition «Otherwise Fail» de TTCN :

$$\frac{\vdash^{v} (p^{\mathrm{TP}}, p^{\mathrm{S}}) : n \quad v \in \{1,2,3\} \quad i \in \Sigma_I \quad (\neg((p^{\mathrm{TP}}, p^{\mathrm{S}}) \xrightarrow{i}_{\mathrm{P}}) \vee (\xrightarrow{i}_{\mathrm{S}} = \emptyset \wedge \xrightarrow{i}_{\mathrm{I}} \neq \emptyset))}{\vdash^{5} \perp : 1 \quad (n, i, 1) \in \mathrm{DAG} \quad \mathit{verdict}(n, i, 1) = \texttt{Fail}} \tag{3.34}$$

▪

Algorithme 4 (La gestion des temporisateurs [FJJV96])
Les temporisateurs doivent assurer la finitude de l'application d'un arbre de test sur une implémentation. Tout blocage ou boucle divergente (τ^{ω}) doit être détecté : lorsque le testeur émet une sortie, il attend en retour un certain nombre d'entrées. A l'inverse, si les délais des temporisateurs sont suffisants, chaque expiration d'un temporisateur doit correspondre à un blocage ou à une boucle divergente. A chaque entrée i est associé un temporisateur t_i. Trois opérations sont possibles sur les temporisateurs : `Start`(t_i), `Cancel`(t_i) et `Timeout`(t_i). Les opérations sur les temporisateurs décorent certaines transitions étiquetées :

1. `Start`(t_i) est réalisé dans la transition qui atteint un état où i est possible,

2. `Cancel`(t_i) est réalisé à la réception de i ou quand celle-ci n'est plus possible (à cause d'un choix par exemple),

3. `Timeout`(t_i) représente l'observation d'un blocage sur l'attente de i. Une transition, entrante de $\perp$, étiquetée par `Timeout`(t_i) est rajoutée en alternative à l'état où une transition étiquetée par i est possible.

▪

Améliorations

L'outil TGV (cf. aux algorithmes $1_{[p.71]}$, $2_{[p.72]}$, $3_{[p.73]}$ et $4_{[p.74]}$) présenté ci-dessus a été amélioré par la mise en œuvre des principes de la génération à la volée ([FJJM92]) appliquées à l'abstraction et à la détermination ([JM97]) du graphe d'état. Plus précisément on construit seulement la partie du graphe d'état nécessaire à la production du cas de test.

Dans [JM99] on propose trois algorithmes à la volée, fondés sur l'algorithme de Tarjan de décomposition d'un graphe orienté en ses composantes fortement connexes maximales. Le premier algorithme, à la différence des algorithmes précédents, construit un ensemble de cas de test (qui est stocké comme un sous-graphe de toutes les séquences menant vers les états `Accept` de l'objectif de test). Le deuxième algorithme proposé construit un cas de test à partir du sous-graphe fourni par le premier algorithme. Le troisième algorithme est utilise par le premier en vue d'anticiper les opérations du second. Les algorithmes proposés construisent des cas de test avec un nombre minimum d'états inconclusifs ([Mor00]).

Ces algorithmes, avec les algorithmes de génération d'objectifs de test à partir d'une spécification (proposés dans l'outil TVEDA - [Pha94b]), ont été englobés dans l'outil TESTCOMPOSER de TTT ([KJG99]).

3.4.4 Machines à état fini étendues

Dans le cas de test de conformité à partir des machines à état fini étendues (EFSMs[44]) le principal problème qui se pose ([BDA]) est le fait qu'une trace de la spécification pourrait ne pas correspondre à aucune trace du modèle.

Les principaux recherches sont issues du domaine de test de conformité pour les automates ou pour les IOSTEs.

Automates

Dans le premier cas on a essayé de coupler les méthodes utilisées pour le test de flot de contrôle (les méthodes de test pour les machines de Mealy) avec les méthodes pour le test de flot de données ([RW85]). Dans [BDA] on présente quelques-unes de ces méthodes :

1. La méthode proposée dans [HU00] étend les critères classiques de couverture du flot de données ([RW85]) aux processus SDL. On construit un modèle du graphe de

44. *Extended Finite State Machines* en anglais

flot (le graphe de flot des messages étendu) qui n'utilise pas d'entrelacements et qui met en évidence les dépendances de flot de contrôle et celles de flot de données, intra et inter-processus (on considère aussi qu'il existe une dépendance de données entre l'émission d'un message et la réception du même message). Ensuite pour chaque dépendance on construit une séquence de test.

Dans [HLSU02] on étend la méthode précédente, en utilisant des techniques symboliques de vérification par modèles ([BCM+92]) guidées par des critères de couverture de test ([RW85]) pour générer des séquences de test. Plus précisément, pour chaque critère de couverture de test (couverture de tous les états, de tous les transitions, de toutes les définitions de variables, de toutes les utilisations de variables, de toutes les entrées et de toutes les sorties) on construit une formule dans la logique temporelle CTL ([CES86]) et on utilise un outil de vérification par modèles pour générer des témoins[45] ou des contre-exemples pour ces formules et donc des séquences de test.

2. Les autres méthodes y mentionnées [CZ93, HLJ95, RDT95] utilisent plus ou moins le même principe : l'utilisation des dépendances de flot de données comme critère de couverture de test et l'utilisation des méthodes de test pour les machines de Mealy pour générer des séquences prenant en compte les dépendances de données.

Dans [LY96] on propose une méthode qui permet d'obtenir directement la partie accessible de la FSM minimisée équivalente avec une EFSM donnée (et cela sans passer par la construction de la FSM-modèle de la EFSM et ensuite par la minimisation et par l'analyse d'accessibilité de celle-ci) et d'appliquer ensuite les méthodes de test pour les machines de Mealy.

Le principe utilisé dans la construction de la FSM minimisée et accessible est de partitionner, pour chaque location de la EFSM, le domaine des valeurs de variables en fonction des gardes des transitions sortantes de cette location (et on obtient ainsi des états dans la FSM). On raffine ensuite les transitions non-stables (les transitions dont le domaine de l'état source n'est pas inclus dans l'image inverse du domaine de l'état cible) en partitionnant le domaine de l'état source. On obtient ainsi éventuellement des nouvelles transitions non-stables et on reprend l'algorithme.

On peut intercaler aussi une analyse d'accessibilité pendant l'algorithme ci-dessus. Cette phase permet d'obtenir directement la FSM minimisée et accessible.

45. *witnesses* en anglais

IOSTEs

Dans [Pha94a] on introduit l'hypothèse d'uniformité étendue forte.

Un automate étendu est considéré une structure imposée à un IOSTE. Plus précisément une transition étendue est un sous-ensemble non-vide de l'ensemble de transitions de IOSTE et un automate étendu est un ensemble fini des transitions étendues.

Si S est un IOSTE et t une transition étendue on définit une *instanciation* de t comme une séquence de transitions consécutives appartenant à t. Une instanciation $\imath$ est *maximale* si elle contient seulement des transitions différentes et il n'existe pas une autre instanciation $\jmath$ de t avec toutes les transitions différentes et $\imath$ une sous-séquence propre de $\jmath$.

Si X est un automate étendu sur S et pour chaque transition étendue t on a une instanciation maximale $\imath_t$ alors $\bigcup_{t \in X} \{\pi_t \sigma_t\}$ est un ensemble de séquences *couvrant* les transitions étendues de X. Ci-dessus $\pi_t \in trace(S.q_0)$, l'état initial de $\imath_t$ appartient à $S.q_0$ `after` π_t et σ_t est obtenue par la concaténation des étiquettes des transitions de $\imath_t$.

On peut définir maintenant, pour S, X et Υ, un ensemble de séquences couvrant les transitions étendues de X, l'*hypothèse d'uniformité étendue forte* sur un IOSTE qui modélise l'implémentation I : si $R_\Upsilon(I, S)$ alors I est conforme à S ($R_5(I, S)$).

Dans [RBJ00] on propose un modèle d'EFSMs fondé sur le modèle de IOSTEs de [LT89]. Ce modèle (IOSTS[46]) est utilisé pour modéliser la spécification, l'implémentation, l'objectif de test et le cas de test. La technique proposée suit les principes de TGV : génération à partir de la spécification et d'un objectif de test (via le produit synchrone) d'un cas de test dont certains états correspondent aux verdicts `pass`, `fail` et `inconclusive`. On introduit aussi une autre opération de composition entre les IOSTSs. Celle-ci est utilisée pour la composition de l'IOSTS du cas de test avec le IOSTS de l'implémentation. Une implémentation est rejetée si dans le IOSTS ainsi obtenu il existe une trace vers un état `fail`. La méthode proposée permet la construction des cas de test valides (par rapport à la spécification et à l'objectif de test).

Néanmoins, les problèmes qui se posent sont la détermination du cas de test obtenu et l'élimination des actions internes (pour lesquels on propose quelques méthodes heuristiques).

46. *Input/Output Symbolic Transition Systems* en anglais

Autres méthodes

Dans [OL99], pour générer des tests (plus précisément des données de test) à partir des spécifications[47], on les utilise pour produire des prédicats (pré-conditions, invariants et post-conditions) et ensuite, en utilisant certaines techniques[48] pour les satisfaire, on arrive à générer des données de test.

Une technique de génération des tests (données et séquences de test) à partir des spécifications est la technique fondée sur l'*analyse des mutations* ([DO91]). Le but de cette méthode est de générer de tests qui font la distinction entre les comportements des deux EFSMs (la spécification initiale et une mutation de celle-ci) qui présentent des différences mineurs. La méthode consiste dans :

1. la *génération des mutations* en utilisant des opérateurs de mutation. On peut obtenir ainsi des mutants équivalents (qui ont le même comportement, exprimé par l'ensemble de traces, avec celui de la spécification initiale) ou des mutants qui peuvent être tués (ils ne possèdent au moins une trace qui se trouve parmi celles de la spécification initiale).

2. On construit ensuite un ensemble de tests (traces) qui est adéquat par rapport à la spécification initiale et l'ensemble des opérateurs de mutation : chaque mutant obtenu par l'application d'un opérateur de mutation peut être tué par au moins une trace de l'ensemble de test. Dans [DO91] on fournit une procédure de décision de l'adéquation[49] et une procédure qui génère des tests adéquats.

Dans [ABD02] on utilise l'analyse des mutations avec les techniques de vérification par modèles pour générer de tests à partir d'EFSMs (décrites dans le langage de l'outil SMV - [BCM+92]). Pour chaque spécification mutante S' on construit une formule $\varphi_{S'}$ de logique temporelle t.q. la spécification initiale S ne satisfait pas $\varphi_{S'}$ ssi la spécification mutante S' peut être tuée. La construction de $\varphi_{S'}$ est faite t.q. les traces qui peuvent tuer S' sont les contre-exemples qui montrent que S ne satisfait pas $\varphi_{S'}$.

3.5 Objectifs de test abstraits

Un objectif de test (OT) est un STE qui décrit un modèle d'interactions entre le testeur et l'IST. Il est décrit du point du vue spécification, c.à-d. les entrées du testeur

47. similaires aux spécifications EFSMs
48. comme la méthode de partitionnement du domaine d'entrées dans catégories
49. accepteur

sont les entrées pour l'IST et ses sorties sont les sorties de l'IST.

On introduit dans la suite les OTs abstraits. Ceux-ci sont des abstractions des OT simples : au lieu d'utiliser des constantes numériques pour les paramètres de signaux on utilise les éléments d'un treillis complet $(L, \sqsubseteq, \sqcup, \sqcap, \bot, \top)$. On a aussi besoin d'une fonction de concrétisation $\phi : L \to 2^{\mathbf{Z}}$ en vue de pouvoir exprimer des relations comme par exemple $v \in \ell$ ou $\ell_1 \cap \ell_2$ (avec $\ell, \ell_1, \ell_2 \in L$).

Soient S et C avec les significations ci-dessus. L'univers des *commandes abstraites* $\Upsilon(L)$ est l'ensemble :

$$\begin{array}{l}\Upsilon^e(L) \cup \Upsilon^s(L), \text{ avec} \\ \Upsilon^e(L) = \{c \ ? \ s(\ell) \mid \ell \in L \wedge c \in C \wedge s \in S\} \\ \Upsilon^s(L) = \{c \ ! \ s(\ell') \mid \ell' \in L \cup \{\oslash\} \wedge c \in C \wedge s \in S\}\end{array}$$

Définition 3.15 (Objectif de test abstrait)

Un *objectif de test abstrait* TP_L (conçu pour la spécification SP, l'ensemble de points de contrôle et d'observation PCO[50] et le treillis support L) est le STE $(Q_{\mathrm{tp}}, \Upsilon(L), \{\xrightarrow{a}_{\mathrm{tp}} \mid a \in \Upsilon(L)\}, q_0^{\mathrm{tp}})$ t.q. :

1. Parmi les états de Q_{tp} il existe deux $q_{\mathrm{a}}^{\mathrm{tp}}$, $q_{\mathrm{r}}^{\mathrm{tp}}$, sans successeurs, avec la signification ACCEPT et REJECT. Ces états seront utiles dans la construction des cas de test.

2. L'ensemble $\Upsilon(L)$ modélise d'une manière similaire à TTCN ([fS92a]) des contraintes locales imposées sur les paramètres des signaux.

3. Dans un état quelconque le testeur peut exécuter

   (a) soit une seule action d'entrée :
   $$(\forall q)(\forall a)(\forall a') \quad (q \in Q \wedge a \in \Upsilon^e(L) \wedge a' \in \Upsilon(L) \setminus \{a\} \wedge Post_a(q) \neq \emptyset \Rightarrow Post_{a'}(q) = \emptyset \wedge |\, Post_a(q)\,| = 1).$$

   (b) soit seulement des actions de sortie :
   $$(\forall q)(\forall a)(\forall a') \quad (q \in Q \wedge a \in \Upsilon^s(L) \wedge a' \in \Upsilon^e(L) \wedge Post_a(q) \neq \emptyset \Rightarrow Post_{a'}(q) = \emptyset).$$

4. L'OT abstrait est déterministe dans le sens :
   $$(\forall (q, c, s, \ell, \ell')) \quad (q \in Q \wedge Post_{c\ !\ s(\ell)}(q) \neq \emptyset \wedge Post_{c\ !\ s(\ell')}(q) \neq \emptyset \Rightarrow \phi(\ell) \cap \phi(\ell') = \emptyset).$$

50. on suppose, pour simplicité, que PCO $= C^{\mathrm{ext}}$

5. Le testeur ne peut pas arriver dans les états $q_{\mathrm{a}}^{\mathrm{tp}}$ et $q_{\mathrm{r}}^{\mathrm{tp}}$ en exécutant une action d'entrée :

   $(\forall q)(\forall a)(q \in Q \wedge a \in \Upsilon^e(L) \Rightarrow Post_a(q) \cap \{q_{\mathrm{a}}^{\mathrm{tp}}, q_{\mathrm{r}}^{\mathrm{tp}}\} = \emptyset)$.

6. Les actions du testeur se retrouvent dans celles de la spécification : pour toute action ($c?s(\cdots)$ ou $c!s(\cdots)$) de TP_L il existe au moins une action similaire dans SP ($c?s(\cdots)$ ou $c!s(\cdots)$).

▪

Exemple 3.3 (OT abstrait) *Dans la figure* 3.3[p.80] *on a représenté un* OT *abstrait avec le treillis support* INT. *Les états R et A sont les états* ACCEPT *et* REJECT.

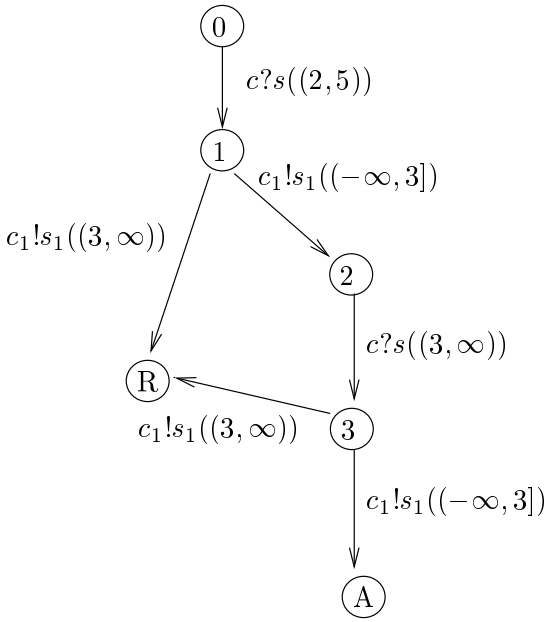

FIG. 3.3 – *Objectif de test abstrait*

Observation 3.2 *Les conditions imposées sur les* OT*s abstraits assurent la compatibilité avec l'outil* TGV *(plus précisément avec les* OT*s conçus pour cet outil). À partir des* OT*s abstraits on peut dériver (dans certaines conditions) des* OT*s simples (pour* TGV*).*

*On a voulu aussi que les interactions entre le testeur et l'*IST *suivent un certain modèle : le testeur commence par envoyer un message vers l'*IST*, il attend tout message venant de tout* PCO*, il fournit un verdict ou il reprend le cycle.*

Dans la suite on utilise une vision du test de conformité légèrement différente de celle qu'on trouve dans [FJJV97, FJJV96] (où les sortie du testeur sont les entrées de la spécification et vice-versa) : nos OTs représentent un processus observateur, interne, de la spécification (ou de l'IUT) et non pas un processus extérieur à celle-ci. Les entrées contrôlables sont les entrées de la spécification, celles qui correspondent aux sorties que le testeur a l'intention de les envoyer vers l'IUT pendant l'exécution des cas de test. Les entrées contrôlables dépendent entièrement du testeur. Elles sont un paramètre de la génération des cas de test : on considère une entrée externe de la spécification si elle fait partie de l'ensemble des entrées contrôlables. Il faut aussi que cet ensemble couvre les entrées de l'OT.

Définition 3.16 (Entrées contrôlables)
L'ensemble des *entrées* (de la spécification) *contrôlables* (par le testeur) est l'ensemble $\Sigma_{\mathrm{f}} \subseteq \{c\ ?\ s(\ell) \mid c\ ?\ s(\ell) \in \Upsilon^e(L) \wedge c \in C^{\mathrm{ext}}\}$, t.q. :

1. Σ_{f} *couvre* l'OT :
$$(\forall\ c\ ?\ s(\ell_1))\quad (\ c\ ?\ s(\ell_1) \in \Upsilon^e(L) \Rightarrow (\exists\ c\ ?\ s(\ell_2))(\ c\ ?\ s(\ell_2) \in \Sigma_{\mathrm{f}} \wedge \phi(\ell_1) \subseteq \phi(\ell_2)\ )\ ).$$
2. $c\ ?\ s(\ell) \in \Sigma_{\mathrm{f}} \Rightarrow (\forall \ell')(\ell' \neq \ell \Rightarrow c\ ?\ s(\ell') \notin \Sigma_{\mathrm{f}})$.

▪

On peut calculer un ensemble d'entrées contrôlable directement à partir des entrées de l'OT :
$$\Sigma_{\mathrm{f}} = \{\ c\ ?\ s\left(\sqcup\{\ell \mid c\ ?\ s(\ell) \in \Upsilon^e(L) \wedge (\ \stackrel{c\ ?\ s(\ell)}{\longrightarrow}_{\mathrm{tp}} \neq \emptyset)\}\right)\ \}.$$

Exemple 3.4 *Pour l'*OT *de la figure* 3.3[p.80] *un ensemble d'entrées contrôlable est* $\{c\ ?\ s(2, \infty)\}$.

La dérivation automatique des tests de conformité se fait par l'exploration d'un type de produit synchrone entre le modèle de la spécification, l'OT abstrait et en considérant l'ensemble d'entrées contrôlables Σ_{f}.

Définition 3.17 (Produit synchrone)
Le *produit synchrone* $\prod(\widehat{SP}, TP, \Sigma_{\mathrm{f}})$ entre le modèle

$\widehat{SP} = (\widehat{Q}, \widehat{\Sigma}, \{\xrightarrow{a}| \ a \in \widehat{\Sigma}\}, \hat{q}_0)$, l'OT $TP_L = (Q_{\rm tp}, \Upsilon(L), \{\xrightarrow{a}_{\rm tp}| \ a \in \Upsilon(L)\}, q_0^{\rm tp})$ et paramétré par $\Sigma_{\rm f}$ est le STE : $(Q_\pi, \widehat{\Sigma}, \{\xrightarrow{a}_\pi| \ a \in \widehat{\Sigma}\}, q_0^\pi)$, avec $Q_\pi \subseteq \widehat{Q} \times Q_{\rm tp}$ et $\{\xrightarrow{a}_\pi| \ a \in \widehat{\Sigma}\}$ sont les plus petits ensembles obtenus par l'application des règles suivantes :

$$\frac{-}{q_0^\pi = (\hat{q}_0, q_0^{\rm tp}) \in Q_\pi} \tag{3.35}$$

$$\frac{(\hat{q}, q_{\rm tp}) \in Q_\pi \quad \hat{q} \xrightarrow{\tau} \hat{q}' \quad q_{\rm tp} \notin \{q_a^{\rm tp}, q_r^{\rm tp}\}}{(\hat{q}', q_{\rm tp}) \in Q_\pi \quad (\hat{q}, q_{\rm tp}) \xrightarrow{\tau}_\pi (\hat{q}', q_{\rm tp})} \tag{3.36}$$

$$\frac{(\hat{q}, q_{\rm tp}) \in Q_\pi \quad \hat{q} \xrightarrow{c?s(v)} \hat{q}' \quad q_{\rm tp} \xrightarrow{c?s(\ell)}_{\rm tp} q'_{\rm tp} \quad v \in \phi(\ell)}{(\hat{q}', q'_{\rm tp}) \in Q_\pi \quad (\hat{q}, q_{\rm tp}) \xrightarrow{c?s(v)}_\pi (\hat{q}', q'_{\rm tp})} \tag{3.37}$$

$$\frac{(\hat{q}, q_{\rm tp}) \in Q_\pi \quad \hat{q} \xrightarrow{c?s(v)} \hat{q}' \quad c?s(\ell) \in \Sigma_{\rm f} \quad v \in \phi(\ell)}{(\hat{q}', q_{\rm tp}) \in Q_\pi \quad (\hat{q}, q_{\rm tp}) \xrightarrow{c?s(v)}_\pi (\hat{q}', q_{\rm tp})} \tag{3.38}$$

$$\frac{(\hat{q}, q_{\rm tp}) \in Q_\pi \quad \hat{q} \xrightarrow{c!s(v)} \hat{q}' \quad q_{\rm tp} \xrightarrow{c!s(\ell)}_{\rm tp} q'_{\rm tp} \quad v \in \phi(\ell) \vee \ell = \oslash}{(\hat{q}', q'_{\rm tp}) \in Q_\pi \quad (\hat{q}, q_{\rm tp}) \xrightarrow{c!s(v)}_\pi (\hat{q}', q'_{\rm tp})} \tag{3.39}$$

$$\frac{(\hat{q}, q_{\rm tp}) \in Q_\pi \quad \hat{q} \xrightarrow{c!s(v)} \hat{q}'}{(\hat{q}', q'_{\rm tp}) \in Q_\pi \quad (\hat{q}, q_{\rm tp}) \xrightarrow{c!s(v)}_\pi (\hat{q}', q'_{\rm tp})} \tag{3.40}$$

▪

Exemple 3.5 *On donne des exemples pour les définitions ci-dessus dans la figure* 3.4[p.83]. *La spécification est composée de deux processus qui communiquent via le canal interne* $cl \in C^{\rm int}$. *Les files externes de la spécification sont* $ci, co \in C^{\rm ext}$. *L'ensemble d'entrées contrôlables est* $\Sigma_{\rm f} = \{ci?sr([1, 10])\}$. *Le treillis support est* INT *et la fonction de concrétisation est* $\phi : {\rm INT} \to 2^{\mathbf{Z}}$, *définie t.q.* $\phi([a, b]) = \{a, a + 1, \ldots, b\}$. *Cet exemple est utilisé ultérieurement pour montrer les changements de la spécification induits par les algorithmes des chapitres* 4[p.85] *et* 5[p.93].

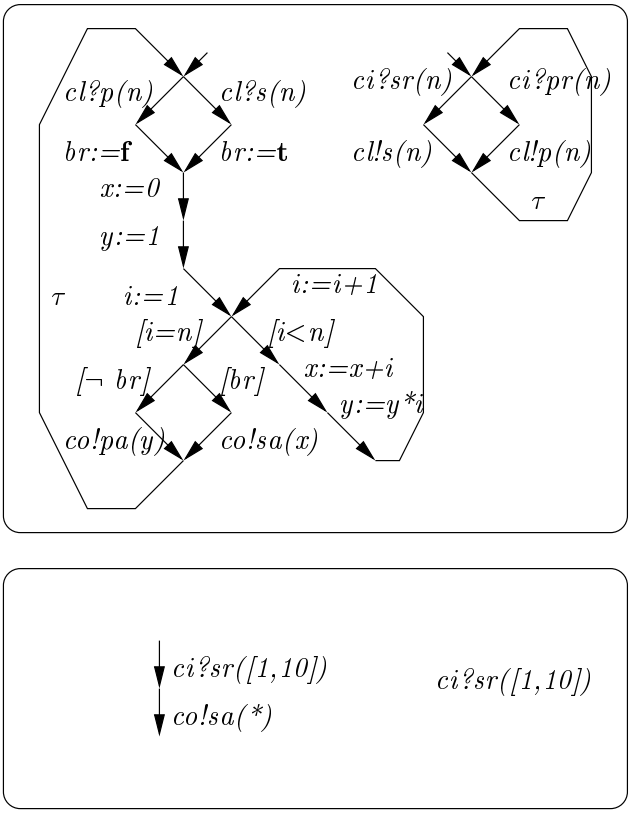

FIG. 3.4 – *Exemple*

3.6 Conclusions

Dans la section 3.5[p.78] on a présenté les objectifs de test abstraits. Ils seront utilisés dans la suite de manière similaire aux critères de tranchage[51]. Ils peuvent être utilisés comme une représentation pour un ensemble d'objectifs de test simples. On a défini leur syntaxe et une opération de composition entre un objectif de test abstrait et une spécification, modélisée par un STE[52].

51. *Slicing criteria* en anglais

52. Une opération de composition directe, au niveau de AEC, comme dans [RBJ00], aurait produit un AEC non-déterministe pour lequel attribuer des verdicts aux certains de ses états aurait été impossible

Chapitre 4

Analyse du contrôle

L'analyse de l'utilité du contrôle restreint (si c'est possible) une spécification, en préservant au moins, le comportement observable (tel qu'il est sélectionné en utilisant un OT et TGV) de celle-ci.

Étant donnés une spécification et un OT, l'objectif de cette analyse est de trouver une approximation sûre de la partie de la spécification qui intervient dans les interactions entre le testeur et l'IST.

Cela pourrait conduire à restreindre la spécification à l'ensemble d'états et de transitions qui pourraient être atteints, étant données un ensemble d'entrées contrôlables.

Une entrée est utile si elle est une entrée contrôlable par le testeur ou si on peut déterminer qu'elle pourrait intervenir dans une exécution de l'implémentation. De manière similaire, une sortie est utile si on peut déterminer qu'elle pourrait intervenir dans une exécution de l'implémentation.

L'analyse de l'utilité du contrôle est fondée sur la technique de tranchage[1] qui est fortement utilisée dans diverses domaines (ex. pour déboguer les programmes). Le tranchage dans le contexte des programmes concurrents a été utilisé pour la première fois par [Che93], puis pour des processus légers[2] par [Kri98] et pour PROMELA ([Hol97]) par [MT98]. Dans [DHZ99] il y a une étude du tranchage des programmes séquentiels en ce qui concerne une formule de LTL et dans [HCD+99] le tranchage est utilisé pour vérifier et extraire des automates à état fini à partir des programmes JAVA. Dans [CFR+99] il y a une application du tranchage dans la vérification des programmes VHDL. Dans le domaine de test, l'idée de limiter la spécification à ce qui est utile selon un objectif de test a été introduite dans [Che96].

1. *slicing* en anglais
2. *threaded programs* en anglais

Une analyse du contrôle qui utilise une abstraction sur les files d'attente et qui simplifie les AECs en fonction du cas de test abstrait choisi a été développée dans [Pac01].

Soient $SP = (S, C, P)$ une spécification, TP un objectif de test et PCO $= C^{\text{ext}}$ l'ensemble de points de contrôle et d'observation.

On calcule une approximation conservatrice de la spécification en utilisant les entrées contrôlables. Chaque processus est restreint aux états et transitions qui pourraient être accessibles en tenant compte des restrictions concernant les entrées contrôlées par le testeur. Intuitivement, cette analyse construit, pour chaque processus, le plus grand sous-processus, après l'élimination des entrées externes, non-contrôlées par testeur (qui ne sont pas couvertes par Σ_{f}) et ensuite les entrées internes qui ne sont pas couvertes par des sorties internes.

Définition 4.1 (Utilité du contrôle)

La *partie utile* de la spécification SP en rapport avec Σ_{f} est une nouvelle spécification : $SP \setminus_{\text{f}} \Sigma_{\text{f}} = (S, C, \{p \setminus_{\text{f}} \Sigma_{\text{f}} \mid p \in P\})$, où $p \setminus_{\text{f}} \Sigma_{\text{f}}$ est le processus $(X_p, (Q'_p, \Sigma_p, \{ \xrightarrow[\setminus_{\text{f}}]{a} p \mid a \in \Sigma_p\}, q_0^p))$ avec $Q'_p \subseteq Q_p$ et $\xrightarrow[\setminus_{\text{f}}]{a} p \subseteq \xrightarrow{a} p, \forall a \in \Sigma_p$ les plus petits ensembles obtenus par l'application des règles suivantes[3] :

$$\frac{}{q_0^p \in Q'_p} \tag{4.1}$$

$$\frac{q_p \in Q'_p \quad q_p \xrightarrow{[b]\ x:=e}_p q'_p}{q'_p \in Q'_p \quad q_p \xrightarrow[\setminus_{\text{f}}]{[b]\ x:=e} p\ q'_p} \tag{4.2}$$

$$\frac{q_p \in Q'_p \quad q_p \xrightarrow{[b]\ c!s(e)}_p q'_p}{q'_p \in Q'_p \quad q_p \xrightarrow[\setminus_{\text{f}}]{[b]\ c!s(e)} p\ q'_p} \tag{4.3}$$

$$\frac{q_p \in Q'_p \quad q_p \xrightarrow{[b]\ c?s(x)}_p q'_p \quad \exists c?s(\cdots) \in \Sigma_{\text{f}}}{q'_p \in Q'_p \quad q_p \xrightarrow[\setminus_{\text{f}}]{[b]\ c?s(x)} p\ q'_p} \tag{4.4}$$

$$\frac{q_p \in Q'_p \quad q_p \xrightarrow{[b]\ c?s(x)}_p q'_p \quad c \in C^{\text{int}} \quad \exists r.\ q_r \xrightarrow[\setminus_{\text{f}}]{[b']\ c!s(e)} r\ q'_r}{q'_p \in Q'_p \quad q_p \xrightarrow[\setminus_{\text{f}}]{[b]\ c?s(x)} p\ q'_p} \tag{4.5}$$

▪

3. La règle 4.5[p.86] concerne la propagation entrées/sorties inter-processus.

Algorithme 5 (Utilité du contrôle) L'algorithme qui implémente les équations $4.1_{[p.86]} \div 4.5_{[p.86]}$ est montré aux tables $4.1_{[p.88]}$, $4.2_{[p.90]}$, $4.3_{[p.91]}$, $4.4_{[p.91]}$.

Il est fondé sur une analyse d'accessibilité locale. On maintient les ensembles d'états et transitions accessibles et on les augmente jusqu'on atteint les plus petit point fixe. Ceci est fait par un parcours en profondeur d'abord de chaque AEC en utilisant le tableau *Vis* qui marque les états et les transitions visitées, la pile *Stk* (qui contient les états à visiter) et l'ensemble *Wait* (qui sert à stocker les transitions étiquetées $[b]\ c?s(x)$ qui interviennent dans la règle $4.5_{[p.86]}$).

Le parcours s'effectue de manière cyclique et différente pour les transitions intervenant dans les règles $4.2_{[p.86]}$, $4.3_{[p.86]}$, $4.4_{[p.86]}$ (procédure *ExploreRule234*) et celles intervenant dans la règle $4.5_{[p.86]}$ (procédure *ExploreRule5*) : la visite des transitions intervenant dans la règle $4.5_{[p.86]}$ est différée jusqu'à ce que toutes les autres transitions aient été visitées (la pile *Stk* est vide). On procède alors par la visite des transitions de l'ensemble *Wait* (pour lesquelles on a pu trouver une sortie correspondante dans un autre processus).

On note par SPEC l'ensemble de toutes les spécifications et par FEED l'ensemble de tous les ensembles d'entrées contrôlables. Les lignes numérotées font référence aux règles correspondantes. Le tableau *match* enregistre l'état du prédicat $\exists r.\ q_r \xrightarrow[\backslash_{\mathrm{f}}]{[b']\ c!s(e)} q'_r$ de la règle $4.5_{[p.86]}$.

La complexité de l'algorithme est polynômiale dans la taille de la spécification (c'est celle du parcours en profondeur d'abord).

▪

Exemple 4.1 *Si l'algorithme* $5_{[p.87]}$ *est appliqué pour la spécification et l'ensemble d'entrées contrôlables de la figure* $3.4_{[p.83]}$ *alors on obtient la spécification montrée à la figure* $4.1_{[p.89]}$.

L'entrée externe $ci?pr(n)$ *n'est pas couverte par les entrées contrôlables et donc son élimination conduit à l'élimination de* $cl!p(n)$ *et donc* $cl?p(n)$ *n'est pas couverte par une entrée interne et donc elle est éliminée aussi avec* $br := \mathrm{f}$.

Lemme 4.1

Soient SP une spécification, Σ_{f} l'ensemble d'entrées contrôlables et $SP \setminus_{\mathrm{f}} \Sigma_{\mathrm{f}}$ la partie utile de la spécification SP en rapport avec Σ_{f}.

Alors $\widehat{SP \setminus_{\mathrm{f}} \Sigma_{\mathrm{f}}}$ est un sous-graphe de $\widehat{SP}$. ▪

Preuve

Les deux STEs $\widehat{SP}$ et $\widehat{SP \setminus_{\mathrm{f}} \Sigma_{\mathrm{f}}}$ ont le même ensemble d'étiquettes et on prouve l'inclusion des états et l'inclusion des relations (pour chaque étiquette). Ceci est fait en

```
procedure SliceControl ((S, C, P) : SPEC; Σ_f : FEED; var (S, C, P \_f Σ_f) : SPEC )
  var
    Stk : pile de ⋃_{p∈P} (Q_p × {p});
    Wait : ensemble de ⋃_{p∈P} (Q_p × Σ_p × {p});
    Vis : ⋃_{p∈P} (Q_p × Σ_p) → bool;
    match : C × S × P → bool;
    proc ∈ P;
    q ∈ ⋃_{p∈P} Q_p;
    a ∈ ⋃_{p∈P} Σ_p;
    c ∈ C;
    s ∈ S;
  procedure
    Init;
    ExploreRule234;
    ExploreRule5;
  procedure MAJ(proc ∈ P; q : ⋃_{p∈P} Q_p; a : ⋃_{p∈P} Σ_p)
      begin
        Q_{proc\_fΣ_f} := Q_{proc\_fΣ_f} ∪ Post_a(q) ;
        -a->_{\_f proc} := -a->_{\_f proc} ∪{(q, q') | q' ∈ Post_a(q)};
      endproc MAJ;
  begin
    Stk := ∅;
    Wait := ∅;
    Init();
    repeat
      ExploreRule234();
      ExploreRule5();
    until (Stk = ∅);
  end.
```

TAB. 4.1 – *Calcul de l'utilité du code*

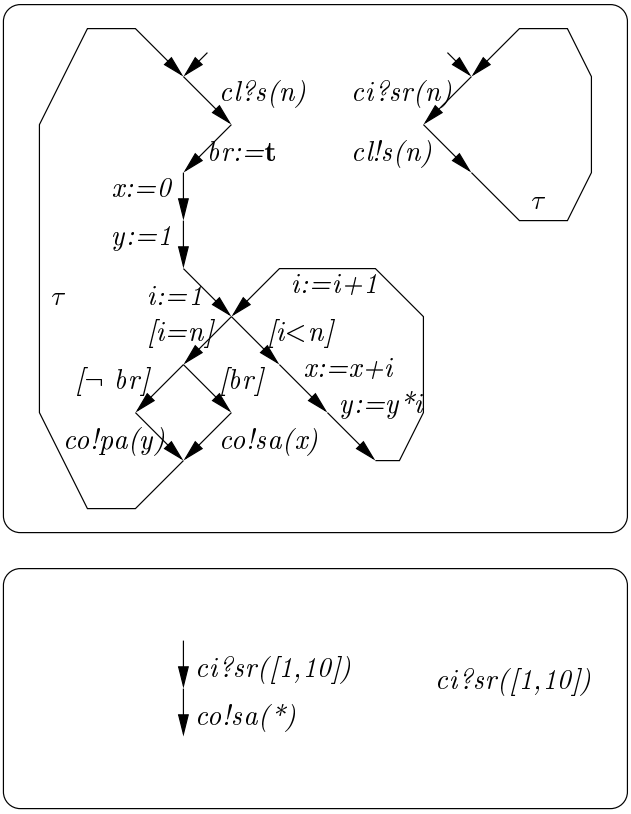

FIG. 4.1 – *Analyse de l'utilité du contrôle*

```
                procedure Init
                  begin
                    foreach (proc ∈ P)
                      X_{proc\f Σf} := X_proc ;
                      Σ_{proc\f Σf} := Σ_proc ;
(4.1[p.86])           q_0^{proc\f Σf} := q_0^proc;
                      Q_{proc\f Σf} := {q_0^{proc\f Σf}};
                      Stk.Push(Q_{proc\f Σf} × {proc});
                      foreach ((c, s) ∈ C × S)
                        match(c, s, proc) := false;
                      foreach (q ∈ Q_proc ∧ a ∈ Σ_proc)
                        Vis(q, a) := false;
                    endfor
                  endproc Init ;
```

TAB. 4.2 – *Calcul de l'utilité du code : procédure Init*

montrant par induction que chaque état accessible dans $\widehat{SP \setminus_{\mathrm{f}} \Sigma_{\mathrm{f}}}$ fait partie des états de $\widehat{SP}$ et que tous ses états successeurs dans $\widehat{SP \setminus_{\mathrm{f}} \Sigma_{\mathrm{f}}}$ font aussi partie des états de $\widehat{SP}$.

La preuve complète se trouve dans l'annexe $B_{[\mathrm{p.175}]}$.

□

Lemme 4.2

Si S, S' sont les modèles sémantiques des deux spécifications SP et SP' et S' est un sous-graphe de S alors $\prod(S', TP, \Sigma_{\mathrm{f}})$ est aussi un sous-graphe de $\prod(S, TP, \Sigma_{\mathrm{f}})$. ▪

Preuve

Le raisonnement est similaire à celui du lemme $4.1_{[\mathrm{p.87}]}$: on prouve l'identité des états initiaux et ensuite l'inclusion des ensembles $Post$.

De plus on tient compte du fait que S' est un sous-graphe de S et donc si les prémisses des règles $3.35_{[\mathrm{p.82}]} \div 3.40_{[\mathrm{p.82}]}$ sont vraies dans $\prod(S', TP, \Sigma_{\mathrm{f}})$ elles seront aussi vraies pour $\prod(S, TP, \Sigma_{\mathrm{f}})$.

□

Corollaire 4.1

$\prod(\widehat{SP \setminus_{\mathrm{f}} \Sigma_{\mathrm{f}}}, TP, \Sigma_{\mathrm{f}})$ est un sous-graphe de $\prod(\widehat{SP}, TP, \Sigma_{\mathrm{f}})$. ▪

Théorème 4.1 (Correction de l'analyse de l'utilité du contrôle)

```
                procedure ExploreRule234
                  begin
                    while (Stk ≠ ∅)
                      (q, proc) := Stk.Pop();
                      foreach (q --a-->_proc q' t.q. ¬ Vis(q, a))
(4.2[p.86])             if ((⟨a⟩a la forme ⟨[b] x := e⟩)∨
(4.3[p.86])                 (⟨a⟩a la forme ⟨[b] c!s(e)⟩)∨
(4.4[p.86])                 (⟨a⟩a la forme ⟨[b] c?s(x)⟩∧
                              (∃I)(I ∈ INT ∧ c?s(I) ∈ Σ_f)))
                            Vis(q, a) := true;
                            MAJ(proc, q, a);
                            Stk.Push((q', proc));
                            if (⟨a⟩ a la forme ⟨[b] c!s(e)⟩)
                              match(c, s, proc) := true;
(4.5[p.86])                 else if (⟨a⟩a la forme ⟨[b] c?s(x)⟩ ∧ c ∈ C^int)
                              Wait := Wait ∪ {(q, a, proc)};
                            else
                              Vis(q, a) := true;
                            endif
                          endfor
                        endwhile
                      endproc ExploreRule234 ;
```

TAB. 4.3 – *Calcul de l'utilité du code : procédure ExploreRule234*

```
                procedure ExploreRule5
                  begin
(4.5[p.86])         while ((∃(q, [b] c?s(x), proc) ∈ Wait) ∧ (∃proc' ∈ P \ {proc})(match(c, s, proc')))
                      Wait := Wait \ {(q, [b] c?s(x), proc)};
                      Vis(q, [b] c?s(x)) := true;
                      MAJ(proc, q, [b] c?s(x));
                      foreach (q' ∈ Post_[b] c?s(x)(q))
                        Stk.Push((q', proc));
                    endwhile
                  endproc ExploreRule5 ;
```

TAB. 4.4 – *Calcul de l'utilité du code : procédure ExploreRule5*

Soient SP une spécification, TP un objectif de test et Σ_{f} l'ensemble d'entrées contrôlables (qui couvrent TP). Les produits synchrones entre les modèles de SP et $SP \setminus_{\mathrm{f}} \Sigma_{\mathrm{f}}$ avec l'objectif de test TP sont fortement bisimilaires :

$$\prod(\widehat{SP}, TP, \Sigma_{\mathrm{f}}) \sim \prod(\widehat{SP \setminus_{\mathrm{f}} \Sigma_{\mathrm{f}}}, TP, \Sigma_{\mathrm{f}}) \ .$$

▪

Preuve

On montre que l'égalité entre les états est aussi une bisimulation et donc c'est la bisimulation forte.

Le fait que $\prod(\widehat{SP}, TP, \Sigma_{\mathrm{f}})$ simule $\prod(\widehat{SP \setminus_{\mathrm{f}} \Sigma_{\mathrm{f}}}, TP, \Sigma_{\mathrm{f}})$ résulte du corollaire $4.1_{[\mathrm{p.90}]}$.

La simulation inverse résulte en appliquant une induction structurelle en fonction de l'étiquette intervenante dans la définition de la relation de bisimulation employée et des règles $3.35_{[\mathrm{p.82}]} \div 3.40_{[\mathrm{p.82}]}$. Le principe général appliqué ici est le suivant : pour chaque transition de $\prod(\widehat{SP}, TP, \Sigma_{\mathrm{f}})$ on remonte en arrière vers (la) les règles $3.35_{[\mathrm{p.82}]} \div 3.40_{[\mathrm{p.82}]}$ qui l'ont générée. En essayant d'appliquer les mêmes règles pour $\prod(\widehat{SP \setminus_{\mathrm{f}} \Sigma_{\mathrm{f}}}, TP, \Sigma_{\mathrm{f}})$ et ainsi obtenir une transition dans $\prod(\widehat{SP \setminus_{\mathrm{f}} \Sigma_{\mathrm{f}}}, TP, \Sigma_{\mathrm{f}})$ qui simule la transition de $\prod(\widehat{SP}, TP, \Sigma_{\mathrm{f}})$, on remonte vers les règles $3.11_{[\mathrm{p.49}]} \div 3.16_{[\mathrm{p.49}]}$ qui ont généré les transitions de $\widehat{SP}$. On montre que les transitions de SP qui apparaissent dans les prémisses de ces règles sont gardées dans $SP \setminus_{\mathrm{f}} \Sigma_{\mathrm{f}}$ et donc en appliquant les règles $3.11_{[\mathrm{p.49}]} \div 3.16_{[\mathrm{p.49}]}$ cette fois pour $SP \setminus_{\mathrm{f}} \Sigma_{\mathrm{f}}$ on obtient une transition dans $\widehat{SP \setminus_{\mathrm{f}} \Sigma_{\mathrm{f}}}$ qui permet, via les règles $3.35_{[\mathrm{p.82}]} \div 3.40_{[\mathrm{p.82}]}$ d'obtenir une transition dans $\prod(\widehat{SP \setminus_{\mathrm{f}} \Sigma_{\mathrm{f}}}, TP, \Sigma_{\mathrm{f}})$ qui simule la transition de $\prod(\widehat{SP}, TP, \Sigma_{\mathrm{f}})$.

La preuve complète se trouve dans l'annexe $B_{[\mathrm{p.177}]}$.

□

Chapitre 5

Analyse des données

Ce chapitre est dédié aux analyses de flot de données. Celles-ci appartiennent au cadre général de l'analyse de programmes ([NNH99]) : en outre l'analyse de flot de données ([Hec77, MR90]), on peut y inclure l'interprétation abstraite ([CH78, CC92b]), l'analyse des contraintes ([Apt99]) et celle des types. Le résultat de ces analyses est utilisé ensuite dans l'optimisation de code ([ASU89, Muc97]).

Les analyses de flot de données permettent le calcul, au moment de la compilation, de quelques types d'invariants de programmes. Il s'agit des invariants ([MRB95, SS98]) construits à partir des notions comme la définition d'une variable (une variable est *définie* dans une instruction du programme soit si elle est lue soit si elle apparaît dans la partie gauche d'une affectation) ou l'utilisation d'une variable (une variable est *utilisée* dans une instruction du programme soit si elle est écrite soit si elle apparaît dans la partie droite d'une affectation) ou des invariants concernant les valeurs de variables (propagation des constantes ou des intervalles entiers)[1].

On s'intéresse aux invariants qui permettent de restreindre le nombre d'états du modèle (et comme ceux-ci sont un produit cartésien entre les états de contrôle du programme et les ensembles des valeurs des contextes de variables/files, on restreint les contextes de variables[2]). Pour restreindre le domaine de définition d'un contexte de variables (et ainsi restreindre la taille des états du modèle) on utilise l'analyse des variables actives ([Hec77]) ou l'analyse des variables utiles ([Wei84, Tip94]). Pour restreindre l'ensemble des valeurs des contextes de variables (et ainsi réduire le nombre d'états et de transitions du modèle) on utilise la propagation des constantes ([Kil73]) ou celle des

1. Il s'agit ici des invariants simples, des invariants plus sophistiqués et précis concernant les valeurs des variables, comme les intervalles linéaires ou affins pouvant être générés seulement en utilisant les techniques de l'interprétation abstraite

2. utilisés dans la génération du modèle d'une spécification, définition 3.11$_{[p.48]}$

intervalles entiers ([CC92b]) et ensuite on applique des techniques comme par exemple l'élimination du code mort.

Les analyses de flot de données ont, essentiellement une nature *équationnelle*. Un problème de flot de données consiste en un *graphe de flot*, un *treillis complet* et un *espace de fonctions* monotones (ou continues). Le système équationnel se construit à partir du graphe de flot (qui est soit le graphe de dépendances de contrôle[3] [MR90, Muc97] soit le graphe inverse de celui-ci, en fonction du type de problème, en avant ou en arrière). Une équation décrit la dépendance entre l'information d'un état et l'information des états prédécesseurs ou successeurs.

Ce qu'on a décrit ci-dessus est valable dans le cadre des analyses de flot intra-processus (ou les analyses de flot pour un seul AEC). Dans le cas d'un réseau de AECs l'analyse de flot est inter-processus et on doit ainsi définir le concept de successeur/prédécesseur d'un état dans un autre processus. Dans le cas des actions de communication inter-processus on a considéré que les actions d'entrée et les actions de sortie avec la même signature (canal de communication, signal et paramètres) peuvent se synchroniser (quelque soit leur place dans les processus différents). Donc si on a deux transitions étiquetées avec deux actions de communication correspondantes on considère que l'information de l'état source de la transition étiquetée avec l'action de sortie peut arriver dans l'état destination de la transition étiquetée avec l'action d'entrée (un concept similaire est celui de précédence immédiate dans le graphe de contrôle des programmes JAVA [DC97]).

À partir du graphe de flot, de la *règle de confluence*[4] et de la valeur initiale des invariants[5] on construit un *opérateur de flot* (qui est monotone ou continue, en fonction de l'espace des fonctions) et dont le plus petit (grand) point fixe est la solution du problème et donc l'invariant cherché.

Dans la littérature il existe plusieurs techniques de calcul du point fixe. Si le treillis est de la hauteur finie alors on peut appliquer un approche itératif classique ([Hec77, Muc97]), en utilisant une liste de travail, initialisée par un parcours en profondeur d'abord du graphe de flot et qui, pendant l'algorithme, contient les états de contrôle dont l'invariant attaché a changé suite à une évaluation d'un prédécesseur/successeur. Un algorithme similaire ([NNH99]) calcule l'invariant attaché directement à une compo-

3. Dans les analyses de flot de données *classiques* on peut avoir des graphes de dépendances de contrôle du programme (intra-procédurals) ou des graphes de dépendances de contrôle du système (inter-procédurals)

4. qui établit comment on combine l'information venant des successeurs/prédécesseurs

5. qui est donnée par les contraintes symboliques (constantes ou intervalles) attachées aux entrées de l'objectif de test

sante fortement connexe maximale. Si le treillis est de la hauteur infinie on utilise des techniques d'approximation approchée du point fixe ([CH78]) tirées de l'interprétation abstraite (l'élargissement et le rétrécissement [CC92b]) et aussi une itération de chaque sous-composante fortement connexe ([Bou92])

Le chapitre commence (section 5.1[p.97]) par une présentation des problèmes de flot intra-processus (pour un seul AEC). Outre la définition d'un problème de flot intra-processus on présente aussi les algorithmes qui permettent le calcul de la solution (ou une approximation de celle-ci) dans le cas des treillis de la hauteur finie/infinie.

On présente quatre analyses de flot de données[6] :

1. L'*analyse d'activité des variables* ([Hec77]) calcule, dans un état, les variables qui seront utilisées sur un chemin partant de cet état et sans qu'une définition de la variable existe sur ce chemin.

2. L'*analyse d'utilité des variables* est similaire à l'analyse d'activité mais elle prend en compte aussi l'objectif de test.

   On utilise la technique de *tranchage*[7] ([Wei84, Tip94]).

   On analyse l'utilité des variables mais aussi celle des sorties ou entrées. Intuitivement, une variable est utile dans un état si sa valeur, par un enchaînement des dépendances des données, est utile dans un certain endroit de la spécification[8] (choisi par utilisateur). Une entrée est utile si elle est une entrée contrôlable par le testeur ou si on peut déterminer qu'elle pourrait intervenir dans une exécution de l'implémentation. De manière similaire, une sortie est utile si on peut déterminer qu'elle pourrait intervenir dans une exécution de l'implémentation.

3. La *propagation des constantes* ([Kil73, WZ91]) calcule, pour chaque variable et dans chaque état de contrôle, une approximation de l'ensemble des valeurs possibles de la variable. L'approximation consiste dans trois valeurs : une constante k (ce qui signifie que la variable a la valeur constante k dans l'état respectif), soit une valeur qui dénote l'ensemble de tout les valeurs (car l'algorithme dans ce cas

6. Ces analyses ont un intérêt dans la mesure où, dans le cas de la génération automatique de cas de test, l'OT restreint la spécification : la partie contrôle de l'OT permet, par l'intermédiaire des analyses d'activité et d'utilité, de réduire la spécification et les contraintes attachées aux paramètres de signaux sont propagées dans la spécification en utilisant les techniques de la propagation des constantes et celle des intervalles

7. *slicing* en anglais

8. critère de tranchage

a détecté au moins deux valeurs pour cette variable), soit une valeur qui dénote le fait que la variable n'a pas été définie.

4. La *propagation des intervalles entiers* ([CC92a]) calcule, pour chaque variable et dans chaque état de contrôle, un intervalle entier qui contient l'ensemble des valeurs possibles de la variable.

Dans la section 5.2[p.109] on présente une extension des problèmes de flot de données intra-processus aux problèmes de flot de données inter-processus (réseaux de AECs qui communiquent asynchronement via des files d'attente).

Dans la section 5.3[p.117] on présente les applications des analyses de flot de données ci-dessus :

1. Pour l'analyse d'activité il s'agit ([BFG99a, BFG00a]) d'une transformation de la spécification originale, en éliminant complètement les variables inactives dans chaque état et, dans chaque transition, une réinitialisation de chaque variable inactive dans l'état suivant. La spécification transformée est fortement bisimilaire à la spécification originale.

2. Pour l'analyse de l'utilité il s'agit ([BFG00b, BFGed]) d'une abstraction de la spécification initiale. On abstrait chaque action d'affectation d'une variable inutile, les paramètres inutiles des signaux des actions d'entrée et ceux inutiles des des actions de sortie. La spécification transformée est fortement bisimilaire à la spécification originale.

3. Les résultats de la propagation des constantes et ceux de la propagation des intervalles entiers seront utilisés ([BFG00b, BFGed]) pour éliminer les transitions dont la garde s'évalue à faux et donc qui ne seront jamais franchissables dans le modèle. La spécification transformée est fortement bisimilaire à la spécification originale.

4. La propagation des constantes est appliquée aussi dans le recouvrement des variables temporisées ([BLM01]). La spécification transformée est fortement bisimilaire à la spécification originale.

Des invariants plus précis que ceux fournis par la propagation des constantes/intervalles entiers pourraient être obtenus en utilisant le cadre offert par les problèmes de flot de données avec le treillis de polyèdres convexes ([Ker94]) ou le treillis de congruences linéaires ([Gra90]). On peut envisager aussi l'utilisation d'autres techniques, comme les techniques de calcul des invariants tirées des optimisations par parallélisation du code séquentiel (en effectuant une analyse des dépendances de données dans les nid-boucles ([PW86, GKT91, CDRV96])) ou celles qui remplacent certaines transitions par

des meta-transitions ([Boi98, Ann01]) ou encore des techniques qui permettent le calcul d'invariants non-linéaires ([BBF+00]).

5.1 Problèmes de flot de données intra-processus

Définition 5.1 (Problème de flot de données intra-processus)
Un *problème de flot de données intra-processus* (PFD) est un tuple $(S, L, F, T, \uplus, \delta, X_0, \mathcal{E})$, où :

1. S est un STE : $(Q, \Sigma, \{\xrightarrow{a} \mid a \in \Sigma\}, q_0)$.
2. L'*espace des propriétés* L est un treillis complet : $(L, \sqsubseteq, \sqcup, \sqcap, \bot, \top)$.
3. L'*espace des fonctions* monotones[9] (ou continues[10]) est $F \subseteq L \to L$ t.q. :
   (a) $1_L, \bar{\top}, \bar{\bot} \in F$,
   (b) F est fermé par les opérateurs $\circ$ (composition des fonctions), $\sqcap$ et $\sqcup$ (où $f \sqcap g, f \sqcup g : L \to L$, $(f \sqcap g)(x) = f(x) \sqcap g(x)$ et $(f \sqcup g)(x) = f(x) \sqcup g(x)$).
4. Les *fonction de transfert* sont $T : \Sigma \to F$. Si Σ contient l'action τ alors on prend $T(\tau) = 1_L$.
5. L'*opérateur de combinaison* est $\uplus \in \{\sqcap, \sqcup\}$.
6. $\delta \in \{\to, \leftarrow\}$ est la *direction de flot* : en avant ($\to$) où en arrière ($\leftarrow$).
7. $X_0 : Q \to L$ est une *approximation initiale* de la solution (habituellement $X_0 = \bar{\top}$ si $\uplus = \sqcap$ et $X_0 = \bar{\bot}$ si $\uplus = \sqcup$ ou n'importe quelle approximation qui satisfait les conditions de l'observation 2.2[p.33].
8. L'*opérateur de flot* est $\mathcal{E} : (Q \to L) \to (Q \to L)$ où, pour tout $q \in Q$:

$$\mathcal{E}(X)(q) = X_0(q) \uplus \biguplus_{a \in \Sigma} \biguplus_{q' \in \begin{cases} Post_a(q) & \text{,si} \quad \delta = \leftarrow \\ Pre_a(q) & \text{,si} \quad \delta = \rightarrow \end{cases}} T(a)(X(q'))$$

On s'intéresse à la *solution* du PFD qui est $lfp(\mathcal{E})$ ou $gfp(\mathcal{E})$ (en fonction de l'opérateur de combinaison $\sqcup$ respectivement $\sqcap$). ▪

9. toutes les fonctions sont monotones
10. idem 9 mais par rapport à la continuité

Notation 5.1 *Pour identifier les éléments d'un* PFD *(ex. P) on utilise le préfixe : P. (ex. P.$\mathcal{E}$).*

5.1.1 Solutions

Pour trouver la solution du PFD ci-dessus on s'appuie sur les théorèmes 2.1$_{[p.32]}$, 2.3$_{[p.34]}$ et 2.5$_{[p.35]}$. Si l'opérateur de flot $\mathcal{E}$ est monotone et l'espace des propriétés L est de la hauteur finie alors on utilise le théorème 2.1$_{[p.32]}$ (le deuxième cas) pour trouver la solution du PFD. Si L n'est pas de la hauteur finie mais $\mathcal{E}$ est continue on trouve une approximation de la solution en utilisant les théorèmes 2.3$_{[p.34]}$ et 2.5$_{[p.35]}$ ([Bou93]).

On utilise plusieurs stratégies d'itération :

1. La première stratégie (WLIST-STR) utilise une file, initialisée par un parcours en profondeur du graphe du STE avec les nœuds visités dans le préordre inverse (si la direction de flot est en arrière) ou dans le postordre inverse (si la direction de flot est en avant). Ensuite l'ordre d'itération est donné par la file même.

   Cette stratégie s'applique aux problèmes de flot avec la hauteur du treillis finie.

2. La deuxième stratégie d'itération (SCC-STR) s'appuie sur l'algorithme de Tarjan ([Muc97]) de décomposition du graphe du STE dans des composantes fortement connexes.

   Pour chaque composante fortement connexe on entretient une file dont l'initialisation et l'itération se fait de manière identique avec la file de la première stratégie.

   L'ordre d'itération des composantes fortement connexes est l'ordre dans laquelle elles sont fournies et stockées dans la pile des composantes fortement connexes par l'algorithme de Tarjan (si la direction de flot est en avant) ou l'ordre inverse (si la direction de flot est en arrière).

   Cette stratégie s'applique aussi aux problèmes de flot avec la hauteur du treillis finie.

3. La troisième stratégie (SCSC-STR) utilise un algorithme de [Bou92] de décomposition du graphe du STE dans des sous-composantes fortement connexes. Cet algorithme fournit aussi un ensemble admissible de points d'élargissement (tout circuit non trivial passe par au moins un point de l'ensemble). Les points d'élargissement sont les racines des sous-composantes fortement connexes.

   On utilise l'algorithme pour obtenir un ensemble admissible de points d'élargissement et ensuite la stratégie est presque identique avec la deuxième stratégie

avec l'exception du fait que dans une composante fortement connexe on utilise les itérations chaotiques croissantes approchées supérieurement avec la stratégie d'itération donnée par la file dont on stocke la composante et avec comme l'ensemble de points d'élargissement l'ensemble obtenu précédemment.

À la fin on effectue une itération utilisant un opérateur de rétrécissement.

Cette stratégie s'applique aux problèmes de flot en avant avec la hauteur du treillis non finie.

Algorithme 6 (Stratégie WLIST-STR**)**

`Entrée :`

P, un problème de flot de données monotone, avec $P.L$ un treillis complet.

`Sortie :`

Le point fixe $X : P.S.Q \to P.L$ de $\mathcal{E}$.

`Méthode :`

La procédure $Iterative - Solver$ (table $A.5_{[p.167]}$) implémente cette stratégie.

On utilise une file (wl), initialisée dans la procédure $InitializeWorklist$ par un parcours en profondeur du graphe du STE avec les nœuds visités dans le préordre inverse (si la direction de flot est en arrière) ou dans le postordre inverse (si la direction de flot est en avant). Les lignes $1, 2, 5$ correspondent à l'initialisation dans le préordre inverse et $3, 4, 5$ correspondent à l'initialisation dans le postordre inverse. On utilise une pile (Stk) pour stocker les états à visiter et l'ensemble Vis pour stocker les états visités.

Cet algorithme est *classique* (il est proposé dans [Hec77, MR90, Muc97]).

L'initialisation de la liste de travail dans un ordre inverse est proposée dans [Muc97]. Un parcours des états de $P.S$ dans le préordre inverse garantit que tous les état successeurs d'un état sont traités avant celui-ci. Un parcours des états de $P.S$ dans le postordre inverse garantit que tous les état prédécesseurs d'un état sont traités avant celui-ci.

Ensuite l'ordre d'itération est donnée par la file même et la procédure $IterateWorklist$ effectue l'itération jusqu'à la stabilisation.

▪

Observation 5.1

Dans l'algorithme de la table $A.5_{[p.167]}$ on initialise la file dans un ordre inverse à celle d'un parcours en profondeur dans préordre ou postordre.

Observons-nous qu'un parcours en préordre n'équivaut pas à un parcours en postordre inverse (et aussi un parcours en postordre n'équivaut pas à un parcours en préordre inverse). On voit dans la figure $5.1_{[p.100]}$ que un parcours en préordre est $1, 2, 4, 7, 5, 8, 3, 6$

et un parcours en postordre est $7, 4, 8, 5, 2, 6, 3, 1$ et inverse $1, 3, 6, 2, 5, 8, 4, 7$ et dans le cas du préordre le nœud 5 est traité avant le nœud 3 ce qui n'est pas le cas du préordre inverse. ▪

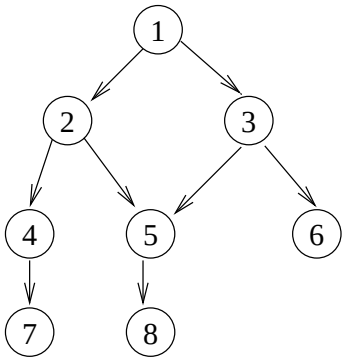

FIG. 5.1 – *Le postordre inverse*

Théorème 5.1 (Stratégie WLIST-STR)

Soit P un PFD. L'algorithme *Iterative* − *Solver* (table $A.5_{[p.167]}$) converge et fournit $lfp(P.\mathcal{E})$ ou $gfp(P.\mathcal{E})$ (en fonction de $P.\uplus$). La complexité temps de l'algorithme est $\mathcal{O}(n \cdot h \cdot d + m)$, où n est le nombre d'états[11], m est la taille de la relation de transition[12], h est la hauteur du treillis[13] et d le degré de branchement[14]. ▪

Preuve

En ce qui concerne la complexité temps on obtient le facteur $n \cdot h$ à cause du fait que pour chaque état c'est possible que la valeur $X(q)$ se stabilise après h pas. Le facteur d existe à cause du fait que $P.\uplus$ s'applique au plus d fois pour un état.

Le terme $n + m$ représente la complexité du parcours en profondeur de la phase de l'initialisation.

On obtient donc la complexité $\mathcal{O}(n \cdot h \cdot d + n + m)$ qui est donc la complexité $\mathcal{O}(n \cdot h \cdot d + m)$.

□

11. $|P.S.Q|$
12. $\sum_{a \in \Sigma} |\{\xrightarrow{a}| \ a \in \Sigma\}|$
13. $h(P.L)$
14. en fonction de $P.\delta$ il est soit $\Delta^{\text{out}}(P.S) \doteqdot \max_{q \in P.S.Q} |P.S.Post(q)|$, soit $\Delta^{\text{in}}(P.S) \doteqdot \max_{q \in P.S.Q} |P.S.Pre(q)|$

Algorithme 7 (Stratégie SCC-STR)

`Entrée :`

P, un problème de flot de données monotone, avec $P.L$ un treillis complet.

`Sortie :`

Le point fixe $X : P.S.Q \to P.L$ de $\mathcal{E}$.

`Méthode :`

La procédure $SCC - Solver$ (tables $A.1_{[p.163]}$, $A.2_{[p.164]}$, $A.3_{[p.165]}$, $A.4_{[p.166]}$) implémente cette stratégie.

Cette stratégie diffère de la stratégie WLIST-STR par le fait qu'au lieu d'utiliser une seule liste de travail on utilise plusieurs listes (files). Chaque liste correspond à une composante fortement connexe maximale. La pile *SCCs* stocke les listes de travail.

La procédure *InitializeWorklist* (table $A.2_{[p.164]}$) construit les composantes fortement connexes maximales et les insère dans *SCCs*. Pour partitionner l'ensemble $P.S.Q$ dans des composantes fortement connexes maximales on utilise l'algorithme de Tarjan : la procédure $Strong - CC$ de table $A.3_{[p.165]}$ est une version de [NSS94] de cet algorithme. On utilise les structures de données suivantes : le tableau *Dfn* sert à stocker l'indice dans l'ordre de parcours en profondeur (la valeur -1 signifie que l'état en question n'est pas encore visité) et le tableau *LL* stocke la valeur de l'indice minimal dans la composante fortement connexe maximale à laquelle l'état courant appartient (cette indice est l'indice de l'état *tête* de la composante : $LL(x) = \min\{Pre(y) \mid y \in P.S.Q \wedge \exists\ x \xrightarrow{*} y$ chemin de $P.S\}$). Le tableau des valeurs booléennes *InComponent* indique si un état fait partie (ou non) d'une composante fortement connexe maximale. Une composante fortement connexe maximale est trouvée quand, pour un état x, les valeurs de *Dfn* et *LL* coïncident. Dans ce cas on dépile les états de la composante (ils ont tous la valeur de *Dfn* plus petite que $Dfn(x)$), on les marque comme faisant partie d'une composante fortement connexe maximale, on les introduit dans la file *scc* et à la fin on introduit cette file dans la pile *SCCs*. On peut observer que chaque file est initialisée soit dans le préordre inverse (les lignes $3, 5, 6$ de la table $A.3_{[p.165]}$) soit dans le postordre inverse (les lignes $4, 5, 6$ de la table $A.3_{[p.165]}$). On observe aussi que l'ordre des listes de travail dans *SCCs* doit etre changé (et cela est fait dans la phase d'itération de la table $A.4_{[p.166]}$) si le problème P est en arrière (car l'algorithme présenté fournit les composantes dans un ordre approprié pour un problème avec la direction de flot en avant).

La phase d'itération (table $A.4_{[p.166]}$) consiste en une itération pour chaque liste de travail (cette itération est identique avec celle de la seule file de la stratégie WLIST-STR : on utilise la même procédure *IterateWorklist* de la table $A.5_{[p.167]}$ mais si dans un état l'information change on introduit dans la pile seulement ses prédécesseurs/successeurs qui font partie de la même composante fortement connexe). Il suffit une seule itération

per composante car le graphe des composantes fortement connexes maximales[15] est acyclique et l'ordre dans lequel les listes de travail sont stockées dans *SCCs* est exactement le préordre ou le postordre inverse du graphe des composantes.

▪

Théorème 5.2 (Stratégie SCC-STR**)**

Soit P un PFD. L'algorithme $SCC - Solver$ (tables $A.1_{[p.163]}$, $A.2_{[p.164]}$, $A.3_{[p.165]}$, $A.4_{[p.166]}$) converge et fournit $lfp(P.\mathcal{E})$ ou $gfp(P.\mathcal{E})$ (en fonction de $P.\uplus$). La complexité temps de l'algorithme est égale à celle de l'algorithme $Iterative - Solver$. ▪

Observation 5.2

Quoique les deux complexités théoriques sont égales, en pratique l'algorithme $SCC - Solver$ peut s'avérer plus efficace à cause du fait qu'au lieu de stabiliser une composante fortement connexe plus grande (comme en effet c'est le cas de l'algorithme $Iterative - Solver$) on stabilise plusieurs composantes fortement connexes (de taille plus petite). ▪

Algorithme 8 (Stratégie SCSC-STR**)**

`Entrée :`

P, un problème de flot de données monotone, avec $P.L$ un treillis complet.

`Sortie :`

$X : P.S.Q \to P.L$, une approximation conservatrice du point fixe de $\mathcal{E}$.

`Méthode :`

La procédure $SCSC - Solver$ (tables $A.6_{[p.168]}$, $A.7_{[p.169]}$, $A.8_{[p.170]}$, $A.9_{[p.170]}$, $A.10_{[p.171]}$, $A.11_{[p.172]}$, $A.12_{[p.172]}$) implémente cette stratégie.

Cette stratégie diffère de la stratégie SCC-STR par le fait qu'elle s'applique aux problèmes de flot de données avec la hauteur du treillis infinie.

On utilise les opérations d'élargissement et de rétrécissement et cela impose que, dans la phase d'initialisation des files, on calcule un ensemble admissible de points d'élargissement (cf. [Bou92]). Ce calcul consiste dans une décomposition du graphe du STE dans des sous-composantes fortement connexes et les points d'élargissement seront les têtes des sous-composantes. Les procédures *Initialize* (table $A.12_{[p.172]}$), $Strong - SCC$ (table $A.7_{[p.169]}$) et *Split* (table $A.8_{[p.170]}$) implémentent ce calcul. Outre les structures de données utilisées aussi dans la stratégie SCC-STR (*LL*, *Dfn*, *InComponent*, *SCCs*) et qui

15. obtenu par un remplacement de chaque composante par un nœud et on y ajoute des arêtes entre les nœuds s'il y a des arêtes entre les états des composantes correspondantes

ont la même signification on utilise le tableau *Elarg* (pour marquer les états d'élargissement). La procédure $Strong - SCC$ présente quelques différences à l'égard de la procédure correspondante $Strong - CC$ (table $A.3_{[p.165]}$). Celles-ci sont marquées par $*$:

1. Le paramètre booléen *truecall* permet de faire une différence entre les appels initiaux de $Strong - SCC$ par les procédures *Initialise* ou *Split*. Dans le premier cas (*truecall* est vrai) la procédure $Strong - SCC$ a un comportement similaire à la procédure $Strong - CC$ en ce qui concerne le stockage des composantes fortement connexes maximales dans *SCCs*. Dans le deuxième cas (*truecall* est faux) on se trouve dans la phase de calcul des sous-composantes fortement connexes dont on ne les stocke pas (on stocke seulement les états têtes des sous-composantes).

2. La variable *loop* sert à faire une distinction entre les composantes fortement connexes triviales (constituées par un seul état et sans transitions) et celles qui contiennent au moins une boucle (y compris celles composées par un seul état x et une transition $x \longrightarrow x$[16]).

   Dans le deuxième cas (*loop* est vraie) on a découvert un point d'élargissement, on l'élimine de sa (sous-) composante, on réinitialise les données concernant les autres états de la (sous-) composante et on applique récursivement la procédure $Strong - SCC$ (pour tous les états successeurs de la tête de composante). Ces actions sont effectuées dans la procédure *Split*.

La phase d'itération est similaire à celle de la stratégie SCC-STR mais on stabilise une composante fortement connexe en utilisant les itérations chaotiques croissantes approchées supérieurement (définition $2.11_{[p.35]}$). Cette phase se termine avec une itération de rétrécissement.

■

Théorème 5.3 (Stratégie SCSC-STR)
Soit P un PFD. L'algorithme $SCSC - Solver$ (tables $A.6_{[p.168]}$, $A.7_{[p.169]}$, $A.8_{[p.170]}$, $A.9_{[p.170]}$, $A.10_{[p.171]}$, $A.11_{[p.172]}$, $A.12_{[p.172]}$) converge et fournit une approximation supérieure de $lfp(\mathcal{E})$. La complexité temps de l'algorithme est quadratique dans le nombre des états et de transitions. ■

Preuve
Dans [Bou92].

□

16. Dans ce cas x est un point d'élargissement car il existe le circuit $x \longrightarrow x$ qui passe par x

<table>
<tr><th>PFD</th><th>Activité</th><th>Utilité</th><th>Constantes</th><th>Intervalles</th></tr>
<tr><td>S</td><td colspan="4">STE sous-jacent de la spécification $(X, Q, \Sigma, \{\xrightarrow{a} \mid a \in \Sigma\}, q_0)$</td></tr>
<tr><td rowspan="2">L</td><td colspan="2" rowspan="2">2^X</td><td colspan="2">$X \to \mathcal{V}$
$\mathcal{V}$ est le treillis</td></tr>
<tr><td>CONST</td><td>INT</td></tr>
<tr><td rowspan="2">F</td><td colspan="2" rowspan="2">$\{\lambda Z.A \cup (Z \setminus B) \mid A, B \subseteq X\}$</td><td colspan="2">$F = \{\lambda f.[v/x]f \mid v \in \mathcal{V} \wedge x \in X \wedge f \in L\}$
$\mathcal{V}$ est le treillis</td></tr>
<tr><td>CONST</td><td>INT</td></tr>
<tr><td rowspan="2">T</td><td colspan="4">voir les sections</td></tr>
<tr><td>5.1.2[p.105]</td><td>5.1.3[p.105]</td><td>5.1.4[p.106]</td><td>5.1.5[p.108]</td></tr>
<tr><td>$\uplus$</td><td colspan="2">$\cup$</td><td colspan="2">$\sqcup_L$</td></tr>
<tr><td>δ</td><td colspan="2">$\leftarrow$</td><td colspan="2">$\rightarrow$</td></tr>
<tr><td>X_0</td><td>$\bar{\emptyset}$</td><td>Rlv_0</td><td>$Const_0(q_0) = \bar{\top}_L$
$Const_0(q') = \bar{\bot}_L,\ q' \neq q_0$</td><td>$Int_0(q_0) = \overline{[-\infty, +\infty]}$
$Int_0(q') = \bar{\bot}_L,\ q' \neq q_0$</td></tr>
<tr><td>$\mathcal{E}$</td><td>$Live$</td><td>Rlv</td><td>$Const$</td><td>Int</td></tr>
</table>

TAB. 5.1 – *Problèmes de flot de données intra-processus*

5.1.2 Activité des variables

L'objet de cette analyse est le calcul, pour chaque état, de l'ensemble des variables qui seront utilisées sur un chemin partant de cet état et sans qu'une définition de la même variable existe sur ce chemin.

Analyse de l'activité des variables est l'instance LIVE du problème de flot de données intra-processus, dont les éléments sont montrés à la table 5.1[p.104].

Les fonctions de transfert $T : \Sigma \to F$ sont t.q. :

1. $T([b]\ y := e) = \lambda Z.(Use(b) \cup Use(e) \cup (Z \setminus \{y\}))$.
2. $T([b]\ c?s(y)) = \lambda Z.(Use(b) \cup (Z \setminus \{y\}))$.
3. $T([b]\ c!s(e)) = \lambda Z.(Use(b) \cup Use(e) \cup Z)$.

Ci-dessus la fonction $Use : AE_X \cup BE_X \to 2^X$ rend les variables de X qui apparaissent dans une expression arithmétique ou logique.

Lemme 5.1

L'espace des fonctions a les propriétés demandées par la définition 5.1[p.97]. ▪

Preuve

La preuve se trouve dans l'annexe B[p.182].

□

Lemme 5.2

LIVE est un PFD monotone. ▪

Preuve

L'espace des fonctions est $F = \{\lambda Z.A \cup (Z \setminus B) \mid A, B \subseteq X\}$. Soient $f \in F$, $f(Z) = A \cup (Z \setminus B)$ et $Z_1 \subseteq Z_2$. Si $Z_1 \subseteq Z_2$ alors $Z_1 \setminus B \subseteq Z_2 \setminus B$ et on obtient $f(Z_1) \subseteq f(Z_2)$.

□

5.1.3 Utilité des données

Cette analyse est similaire à l'analyse d'activité mais dans ce cas le point de départ de l'analyse est constitué par le critère de tranchage : une variable est utile dans un état si sa valeur, par un enchaînement des dépendances des données (qui ne contient aucune définition de la variable), est utilisée dans une sortie observable par le testeur.

Analyse de l'utilité des données est l'instance RLV du problème de flot de données intra-processus, dont les éléments sont montrés à la table 5.1[p.104].

Les fonctions de transfert $T : \Sigma \to F$ sont définies comme suit :
$T(\sigma) = \lambda Z.(\, Use(\sigma, Z) \cup (Z \setminus Def(\sigma)))$, où $Def : \Sigma \to 2^X$ et $Use : \Sigma \times 2^X \to 2^X$ sont

$$Def(\sigma) = \begin{cases} \{y\} & , \ \sigma = [b]\ y := e \vee \sigma = [b]\ c?s(y) \\ \emptyset & , \ \text{sinon} \end{cases} \quad \text{et}$$

$$Use(\sigma, Z) = \begin{cases} vars(b) \cup vars(e) & , \ \sigma = [b]\ y := e \wedge y \in Z \\ vars(b) & , \ \text{sinon} \end{cases}$$

et la fonction $vars : AE_X \cup BE_X \to 2^X$ rend les variables de X qui apparaissent dans une expression arithmétique ou logique.

Observation 5.3

En ce qui concerne les actions du type $[b]\ c!s(e)$ c'est évident que $Def([b]\ c!s(e)) = \emptyset$. On a mis $Use([b]\ c!s(e)) = vars(b)$ seulement parce que l'ensemble $vars(e)$ pourrait être pris en calcul dans $Rlv_0(q)$, où q est un état t.q. $Post_{[b]\ c!s(e)}(q) \neq \emptyset$. ▪

Lemme 5.3

L'espace des fonctions a les propriétés demandées par la définition 5.1[p.97]. ▪

Preuve

Preuve similaire au lemme 5.1[p.105].

□

Lemme 5.4

RLV est un PFD monotone. ▪

Preuve

Preuve similaire au lemme 5.2[p.105].

□

5.1.4 Propagation des constantes

L'objet de cette analyse est le calcul d'invariants du type : la variable x a la valeur 5 dans l'état q, ou la variable y a n'a pas la valeur constante dans l'état p ou la variable z a n'est pas définie dans l'état p ([Kil73]).

Les résultats de cette analyse pourront être utilisés pour réduire la spécification (élimination des transitions dont la garde s'évalue au faux).

La propagation des constantes est l'instance CONST du problème de flot de données intra-processus, dont les éléments sont montrés à la table 5.1[p.104].

Les fonctions de transfert $T : \Sigma \to F$ sont t.q. :

1. $T([b]\ y := e) = \lambda f.[Eval_f(e)/y]f$,

2. $T([b]\ c?s(y)) = \lambda f.[Env(c,s)/y]f$,

3. $T([b]\ c!s(e)) = 1_L$,

où la fonction $Eval_f : AE_X \to$ CONST est définie comme suit :

$Eval_f(k) = k, \forall k \in D \setminus \{\mathbf{t}, \mathbf{f}\}$

$Eval_f(x) = f(x), \forall x \in X$

$$Eval_f(x+y) = \begin{cases} \top & , \ f(x) = \top \vee f(y) = \top \\ \bot & , \ (f(x) = \bot \wedge f(y) \neq \top) \vee \\ & \quad (f(y) = \bot \wedge f(x) \neq \top) \\ c_1 + c_2 & , \ f(x) = c_1 \wedge f(x) = c_2 \end{cases}$$

$Eval_f(x-y)$ similaire au cas $x+y$

et la fonction $Env : C \times S \to$ CONST est une approximation des valeurs fournies par l'environnement au processus.

Lemme 5.5

L'espace des fonctions a les propriétés demandées par la définition 5.1[p.97]. ▪

Preuve

La preuve se trouve dans l'annexe B[p.183].

□

Lemme 5.6

CONST est un PFD monotone. ▪

Preuve

L'espace des fonctions est $F = \{\lambda f.[v/x]f \mid v \in \text{CONST} \wedge x \in X \wedge f \in L\}$. Soient $f_1, f_2 \in L$ t.q. $f_1 \sqsubseteq f_2$. Alors $(\forall y \in X \setminus \{x\})(f_1(y) \sqsubseteq f_2(y))$ et donc $(\forall y \in X \setminus \{x\})([v/x]f_1(y) \sqsubseteq [v/x]f_2(y))$. En plus $v = [v/x]f_1(x) = [v/x]f_1(x)$ et donc $(\forall y \in X)([v/x]f_1(y) \sqsubseteq [v/x]f_2(y))$. On obtient $[v/x]f_1 \sqsubseteq [v/x]f_2$.

□

Le lemme suivant montre que la propagation des constantes est une analyse de flot de données correcte (les invariants calculés sont des approximations supérieures des ensembles de valeurs des variables au moment de l'exécution) :

Lemme 5.7 (Correction de la propagation des constantes)

Soient $\widehat{SP} = (\widehat{Q}, \widehat{\Sigma}, \{\xrightarrow{a} \mid a \in \widehat{\Sigma}\}, \hat{q}_0)$ et $Y = lfp(Const)$ la solution du problème CONST. Soit $\phi :$ CONST $\to 2^{\mathbf{Z}}$ la fonction définie t.q. $\phi(\top) = \mathbf{Z}$, $\phi(\bot) = \emptyset$ et $\phi(x) = \{x\}$, pour $x \in$ CONST$\setminus\{\bot, \top\}$. Le résultat suivant est vrai : $(\forall\ (q, \sigma, \rho))(\forall x)((q, \sigma, \rho) \in \widehat{Q} \wedge x \in X \Rightarrow \sigma(x) \in \phi(Y(q)(x)))$. ▪

Preuve

On montre par induction : le résultat est vrai dans $\hat{q}_0$ et s'il est vrai dans un état alors il est vrai dans tous les états successeurs. La preuve complète se trouve dans l'annexe $B_{[p.183]}$.

□

5.1.5 Propagation des intervalles entiers

L'objet de cette analyse est le calcul d'invariants du type : la variable x prend des valeurs dans l'intervalle $[a, b]$ dans l'état p ([CC92b]). On s'appuie aussi sur les résultats de l'arithmétique des intervalles ([Kea96, RR88]).

Les résultats de cette analyse pourront être utilisés pour réduire la spécification (élimination des transitions dont la garde s'évalue au faux).

Propagation des intervalles entiers est l'instance INT du problème de flot de données intra-processus, dont les éléments sont montrés à la table $5.1_{[p.104]}$.

Les fonctions de transfert $T : \Sigma \to F$ sont t.q. :

1. $T([b]\ y := e) = \lambda f.[Eval_f(e)/y]f$,
2. $T([b]\ c?s(y)) = \lambda f.[Env(c, s)/y]f$,
3. $T([b]\ c!s(e)) = 1_L$,

où la fonction $Eval_f : AE_X \to \text{INT}$ est définie comme suit :

$Eval_f(k) = [k, k], \forall k \in D \setminus \{\mathbf{t}, \mathbf{f}\}$

$Eval_f(x) = f(x), \forall x \in X$

$$Eval_f(x + y) = \begin{cases} [-\infty, +\infty] & , \quad f(x) = [-\infty, +\infty] \vee f(y) = [-\infty, +\infty] \\ \bot & , \quad (f(x) = \bot \wedge f(y) \neq [-\infty, +\infty]) \vee \\ & \quad\ (f(y) = \bot \wedge f(x) \neq [-\infty, +\infty]) \\ c_1 + c_2 & , \quad f(x) = c_1 \wedge f(x) = c_2 \end{cases}$$

$Eval_f(x - y)$ similaire au cas $x + y$

et la fonction $Env : C \times S \to \text{INT}$ est une approximation des valeurs fournies par l'environnement au processus.

Ci-dessus, si $\text{INT} \ni c_i = [a_i, b_i]$, $i \in \{1, 2\}$ alors $c_1 \pm c_2 = [a_1 \pm a_2, b_1 \pm b_2]$.

Lemme 5.8

L'espace des fonctions a les propriétés demandées par la définition $5.1_{[p.97]}$. ▪

Preuve

Preuve similaire au lemme $5.5_{[p.107]}$.

□

Lemme 5.9

INT est un PFD continue. ▪

Preuve

L'espace des fonctions est $F = \{\lambda f.[v/x]f \mid v \in \text{INT} \wedge x \in X \wedge f \in L\}$. Soient $(f_n)_{n\in\mathbf{N}} \subseteq L$. Il faut montrer que $[v/x] \sqcup_{n\in\mathbf{N}} f_n = \sqcup_{n\in\mathbf{N}}[v/x]f_n$.

$[v/x] \sqcup_{n\in\mathbf{N}} f_n(y) = \begin{cases} \sqcup_{n\in\mathbf{N}} f_n(y) & , \quad y \neq x \\ v & , \quad y = x \end{cases}$.

Mais $\sqcup_{n\in\mathbf{N}}[v/x]f_n(y) = \sqcup_{n\in\mathbf{N}} \begin{cases} f_n(y) & , \quad y \neq x \\ v & , \quad y = x \end{cases} = \begin{cases} \sqcup_{n\in\mathbf{N}} f_n(y) & , \quad y \neq x \\ v & , \quad y = x \end{cases}$.

Donc $[v/x] \sqcup_{n\in\mathbf{N}} f_n = \sqcup_{n\in\mathbf{N}}[v/x]f_n$.

□

Lemme 5.10

Soient $\widehat{SP} = (\widehat{Q}, \widehat{\Sigma}, \{\xrightarrow{a} \mid a \in \widehat{\Sigma}\}, \hat{q}_0)$ et Y la solution du problème INT. Soit ϕ : INT $\to 2^{\mathbf{Z}}$ la fonction définie t.q. $\phi(\top) = \mathbf{Z}$, $\phi(\bot) = \emptyset$ et $\phi((a,b)) = \{a, a+1, \ldots, b\}$. Le résultat suivant est vrai : $(\forall\ (q,\sigma,\rho))(\forall x)((q,\sigma,\rho) \in \widehat{Q} \wedge x \in X \Rightarrow \sigma(x) \in \phi(Y(q)(x)))$. ▪

Preuve

Preuve similaire à la lemme 5.7[p.107]. □

5.2 Problèmes de flot de données inter-processus

L'objet d'un problème de flot de données inter-processus est l'analyse des flot de données pour une composition asynchrone d'automates étendus communicants[17].

On les décrit d'abord de manière générale (comme une classe abstraite de problèmes) et ensuite on obtient des instances en particularisant certains éléments de leur définition (ex. en prenant L comme 2^X ou $X \to$ CONST).

On a deux types de problèmes de flot de données inter-processus :

1. Le problème global peut être décomposé en sous-problèmes plus petits, un pour chaque automate étendu communicant (et cela parce que le flot de données global peut être décomposé en flot locaux correspondant à chaque AEC). Dans ce cas la solution est obtenue en résolvant chaque sous-problème (dans n'importe quel ordre).

17. Dans notre cas on considère aussi que les files d'attente sont sans perte de message

Plus formellement, si le problème globale PFD peut être décrit comme une famille de problèmes $(\text{PFD}_i)_{i\in I}$ (où chaque problème PFD_i est défini comme dans la définition 5.1$_{[p.97]}$) alors résoudre PFD revient à résoudre chaque PFD_i.

L'analyse d'activité des variables appartient à cette catégorie.

2. Le problème associé à la composition asynchrone ne peut pas être décomposé en sous-problèmes et cela est dû au fait qu'il existe de dépendances (ou interférences) entre les flot de données locaux. Comme on a prévu que les ensembles de variables pour chaque AEC sont disjoints deux à deux il résulte que les seules dépendances sont dues au transfert de données entre les AECs.

   La propagation des contraintes (constantes ou intervalles) ou l'analyse d'utilité des données appartiennent à cette catégorie.

   Pour résoudre ce type de problèmes on a plusieurs choix :

   (a) On compose de manière asynchrone les automates (une solution coûteuse à cause du nombre exponentiel d'états du produit asynchrone).

   Plus formellement : soit $\text{PFD}_i = (S_i, L, F, T_i, \uplus, \delta, X_0^i, \mathcal{E}^i)$, $i \in I$ les problèmes de flot de données associés à chaque AEC.

   Soit S_r le produit asynchrone $|||_{i\in I} S_i$ (auquel, dans le cas de la propagation des constantes/intervalles on ajoute $|C^{\text{int}} \times S|$ nouvelles variables[18]).

   Soit PFD un problème de flot de données défini comme dans la définition 5.1$_{[p.97]}$: $(S_r, L_r, F_r, T_r, \uplus, \delta, X_0^r, \mathcal{E}^r)$ t.q. $L_r = L' \times \prod_{i\in I} L$[19] et $F_r \subseteq L_r \to L_r$ (avec les restrictions imposées dans la définition 5.1$_{[p.97]}$ sur l'espace de fonctions), $T_r : \Sigma_r \to F_r$ (qui prennent en compte aussi les nouvelles variables : à l'étiquette $[b]\ c?s(y)$ ou $[b]\ c!s(e)$, $c \in C^{\text{int}}$ correspond la même fonction de transfert que pour l'étiquette $[b]\ y := v_{c,s}$, respectivement l'étiquette $[b]\ v_{c,s} := e$, où $v_{c,s}$ est la nouvelle variable correspondant à la file interne c et au signal s), $X_0^r : Q_r \to L_r$ et $\mathcal{E}^r : (Q_r \to L_r) \to (Q_r \to L_r)$.

   On résout PFD en utilisant les techniques indiquées dans la section 5.1$_{[p.97]}$.

   (b) On construit l'union des AECs de la spécification. On obtient un problème dont la solution, pour les analyses qu'on considèrent, coïncide (et dans le même temps est moins coûteuse) avec la solution précédente.

18. On abstrait les contextes de files $C^{\text{int}} \to (S \times D)^*$ par $C^{\text{int}} \times S \to \mathcal{V}$, où $\mathcal{V}$ est le treillis CONST ou INT

19. où L' est le treillis complet correspondant aux nouvelles variables

Toutefois, avec ce type de problème on ne peut pas obtenir des invariants de la forme : dans l'état de contrôle global $(q_1, \ldots, q_n)$ la valeur v du paramètre de signal s dans la file interne c est constante (ou $v \in [a, b]$, $a, b \in \mathbf{Z}$). Les invariants qu'on peut obtenir sont de type : la valeur v du paramètre de signal s dans la file interne c est constante (ou appartient à un intervalle entier).

Le fait que les solutions coïncident s'explique, de manière informelle, par le fait qu'on utilise un opérateur de confluence qui collecte l'information (au lieu de la raffiner) et donc dans l'AEC obtenu par le produit asynchrone d'AECs initiaux, on peut simplifier les losanges obtenus par l'entrelacement. Notamment, on considère seulement les n chemins *purs* (ceux qui sont aussi des chemins d'AECs composants) obtenus par la projection de n'importe quel chemin du losange.

Dans le cas de l'analyse de l'utilité on observe d'abord que, si dans un chemin quelconque $q_1 \xrightarrow{v:=\ldots} q_2 \longrightarrow \ldots q_3 \xrightarrow{x:=v} q_4$, t.q. x, v sont des variables et entre q_2 et q_3, dans le chemin, il n'y a pas d'autre définition de v, on remplace par τ l'étiquette de n'importe quelle transition (sauf la première et la dernière) et si la variable x est utile dans l'état q_4 alors v est utile dans q_2 (même après le remplacement). On peut étendre ce raisonnement dans le cas d'un losange $\blacklozenge$ dans l'AEC produit asynchrone[20] : $\bar{q}_1 \xrightarrow{v:=\ldots}_i \bar{q}_2 \blacklozenge \bar{q}_3 \xrightarrow{x:=v}_i \bar{q}_4$ (on considère aussi que dans le losange il existe un chemin entre $\bar{q}_2$ et $\bar{q}_3$ t.q. par sa projection on obtient le chemin γ mentionné ci-dessus : $q_1 \xrightarrow{v:=\ldots}_i q_2 \longrightarrow_i \ldots q_3 \xrightarrow{x:=v}_i q_4$). L'information concernant la variable v, dans l'état $\bar{q}_2$, collectée sur tout chemin du losange qui est obtenu à partir de γ est incluse (même égale[21]) dans celle collectée sur γ.

Il nous reste le cas de chemin $\eta\ \bar{q}_1 \xrightarrow{c!s(v)}_i \bar{q}_2 \longrightarrow \ldots \bar{q}_3 \xrightarrow{c?s(x)}_j \bar{q}_4$, où x, v sont des variables, $c \in C^{\text{int}}$, $i \neq j$ et entre $\bar{q}_2$ et $\bar{q}_3$ dans η il n'y a pas de transition étiquetée avec $c!s(\ldots)$. Si x est utile dans $\bar{q}_4$ alors $v_{c,s}$ est utile dans $\bar{q}_3$ et ensuite dans $\bar{q}_2$ et donc v est utile dans $\bar{q}_1$. Mais dans l'AEC union on a introduit la transition $q_1 \xrightarrow{(0,x:=v)} q_4$ spécialement pour ce cas. Donc on peut conclure que v est utile dans un état de contrôle global $\bar{q}$ ssi elle est utile dans l'état q (ou v, q appartiennent au même AEC).

Le raisonnement précédent peut être appliqué dans le cas de la propaga-

20. par $\bar{q}$ on note un état de contrôle global dont la projection sur un AEC initial est l'état de contrôle q

21. On tient compte aussi du fait que les AECs initiaux ne partagent pas leurs variables et donc on ne peut pas remplacer une transition de γ par une transition $\ldots \xrightarrow{v:=\ldots}_j \ldots$, $i \neq j$

tion des constantes/intervalles. Le cas de la propagation de l'information par l'intermédiaire du losange ⧫ est simple : la propagation de l'information composée équivaut à une «composition» de la propagation par l'intermédiaire de chemins purs. Dans le cas du chemin η on peut utiliser le raccourci de la transition $q_1 \overset{(0,x:=v)}{\longrightarrow} q_4$ et donc on arrive à la même conclusion : la projection de l'information calculée sur une variable v dans un état $\bar{q}$ est identique avec celle de l'état q (q, v appartiennent au même AEC).

Dans la suite on considère seulement cette formulation d'un problème de flot de données inter-processus (notre approche est similaire à celle de [Dwy95] pour les analyses de flôt de données des programmes ADA).

Définition 5.2 (Problème de flot de données inter-processus)
Soient $(S_i)_{i\in 1..n}$ des STEs avec les PFD intra-processus associés :
$(S_i, L_i, F_i, T_i, \uplus, \delta, X_0^i, \mathcal{E}_i)_{i\in 1..n}$.
Un *problème de flot de données inter-processus* associé à la composition asynchrone des STEs est le problème de flot de données intra-processus $(S, L, F, T, \uplus, \delta, X_0, \mathcal{E})$, où :

1. S est le τ-STE obtenu par l'union de $(S_i)_{i\in 1..n} : (Q, \Sigma, \{\overset{a}{\longrightarrow} | \ a \in \Sigma\}, q_0) = \bigcup_{i=1}^{n} S_i$.
2. L'*espace des propriétés* L est le treillis produit cartésien de treillis : $\prod_{i=1}^{n} L_i$.
3. L'*espace des fonctions* est $F \subseteq L \to L$ t.q. les conditions pour F de la définition 5.1$_{[p.97]}$ sont accomplies.
4. Les *fonctions de transfert* $T : \Sigma \to F$ accomplissent les conditions :
   (a) $T(\tau, \cdot) = 1_L$,
   (b) $T(\sigma, i)(\ell_1, \ldots, \ell_i, \ldots, \ell_n) = (\ell_1, \ldots, T_i(\sigma)(\ell_i), \ldots, \ell_n)$, $i \in 1$ .. n,
   (c) $T(\sigma_i \parallel \sigma_j, 0)(\ell_1, \ldots, \ell_i, \ldots, \ell_j, \ldots, \ell_n) = (\ell_1, \ldots, f_{ji}(\sigma)(\ell_i), \ldots, \ell_j, \ldots, \ell_n)$, où $f_{ji} : \Sigma \to (L_j \to L_i)$ et $\sigma \doteqdot (\sigma_i \parallel \sigma_j, 0)$.
5. L'*opérateur de combinaison* est $\uplus \in \{\sqcap, \sqcup\}$.
6. $\delta \in \{\to, \leftarrow\}$ est la *direction de flot*.
7. $X_0 : Q \to L$ est une *approximation initiale* de la solution qui satisfait les conditions de l'observation 2.2$_{[p.33]}$: $X_0(q_0) = (X_0^i(q_0^i))_{i\in 1..n}$ et $X_0(q) = (\cdot, \ldots, X_0^i(q), \ldots, \cdot)$ si

$q \in Q_i$. Ci-dessus $\cdot$ signifie n'importe quelle valeur de L_i (d'ailleurs cette valeur ne sera pas changée dans le point fixe de l'opérateur de flot car dans un processus seulement ses variables peuvent être écrites et, comme on verra ultérieurement que les treillis sont liés aux variables, il résulte qu'un processus peut modifier seulement les éléments du son treillis).

8. L'*opérateur de flot* est $\mathcal{E} : (Q \to L) \to (Q \to L)$ où, pour $q \in Q$:

$$\mathcal{E}(X)(q) = \quad X_0(q) \uplus \biguplus_{a \in \Sigma} \quad \biguplus_{q' \in \begin{cases} Post_a(q) & ,\text{si} \quad \delta = \leftarrow \\ Pre_a(q) & ,\text{si} \quad \delta = \rightarrow \end{cases}} T(a)(X(q')) \quad .$$

On s'intéresse à la *solution* du PFD qui est $lfp(\mathcal{E})$ ou $gfp(\mathcal{E})$ (en fonction de l'opérateur de combinaison $\sqcup$ respectivement $\sqcap$). ▪

Dans la suite on considère SP la spécification $(S, C, (A_i)_{i \in 1..n})$, où les AECs sont $A_i = (X_i, S_i)$, $i \in 1 \,..\, n$ avec les STEs sous-jacents $S_i = (Q_i, \Sigma(X_i), \{\xrightarrow{a}_i | \; a \in \Sigma(X_i)\}, q_0^i)$, $i \in 1 \,..\, n$.

Soit SP' l'union des AECs $\bigcup_{i=1}^{n} A_i$: $SP' = (X, S')$, où $X = \bigcup_{i=1}^{n} X_i$ et

$S' = \bigcup_{i=1}^{n} S_i \doteqdot (Q, \Sigma, \{\xrightarrow{a} | \; a \in \Sigma\}, q_0)$ est le STE sous-jacent de SP'.

5.2.1 Activité des variables

Soient $\text{LIVE}_i = (S_i, 2^{X_i}, F_i, T_i, \cup, \leftarrow, \bar{\emptyset}, Live_i)$ les PFDs associés aux AECs $A_i, i \in 1 \,..\, n$.

Comme dans notre modèle les processus n'ont pas de mémoire partagée une analyse globale (inter-processus) d'activité des variables revient à une analyse locale (intra-processus) pour chaque processus.

Ceci revient à trouver les solutions des problèmes $(\text{LIVE}_i)_{i \in 1..n}$ (qui sont $lfp(Live_i)$).

5.2.2 Utilité des données

Soient $\text{RLV}_i = (S_i, 2^{X_i}, F_i, T_i, \cup, \leftarrow, Rlv_0^i, Rlv_i)$ les PFDs associés aux AECs $A_i, i \in 1 \,..\, n$.

L'analyse de l'utilité des données inter-processus est l'instance IP-RLV du problème de flot de données inter-processus, dont les éléments sont montrés à la table 5.2[p.114].

Les fonctions de transfert $T : \Sigma \to F$ sont définies comme suit :

PFD	Utilité	Constantes	Intervalles
S	S', le STE sous-jacent de la spécification SP'		
L	2^X $X \doteqdot \bigcup_{i=1}^{n} X_i$	$\prod_{i=1}^{n} (X_i \to \mathcal{V})$ ou[a] $X \to \mathcal{V}$ $\mathcal{V}$ est le treillis	
		CONST	INT
F	$\{\lambda Z.A \cup (Z \setminus B) \mid A, B \subseteq X\}$	$F = \{\lambda f.[v/x]f \mid v \in \mathcal{V} \wedge x \in X \wedge f \in L\}$ $\mathcal{V}$ est le treillis	
		CONST	INT
T	voir les sections		
	5.2.2[p.113]	5.2.3[p.115]	5.2.4[p.117]
$\uplus$	$\cup$	$\sqcup_L$	
δ	$\leftarrow$	$\rightarrow$	
X_0	Rlv_0	$Const_0(q_0) = \bar{\top}_L$ $Const_0(q') = \bar{\bot}_L,\ q' \neq q_0$	$Int_0(q_0) = \bar{\top}_L$ $Int_0(q') = \bar{\bot}_L,\ q' \neq q_0$
$\mathcal{E}$	Rlv	$Const$	Int

TAB. 5.2 – *Problèmes de flot de données inter-processus*

[a] les deux ensembles sont isomorphes

$T(\sigma,i) = \lambda Z.(Use(\sigma,i,Z) \cup (Z \setminus Def(\sigma,i,Z)))$, où $Def : \Sigma \times 2^X \to 2^X$ et $Use : \Sigma \times 2^X \to 2^X$ sont

$$Def(\sigma,i,Z) = \begin{cases} \{y\} & , \ (\sigma = [b]\ y := e \vee \sigma = [b]\ c?s(y)) \wedge i \neq 0 \\ Z & , \ \text{sinon} \end{cases}$$

$$Use(\sigma,\cdot,Z) = \begin{cases} vars(b) \cup vars(e) & , \ \sigma = [b]\ y := e \wedge y \in Z \\ vars(b) & , \ \text{sinon} \end{cases}$$

et la fonction $vars : AE_X \cup BE_X \to 2^X$ rend les variables de X qui apparaissent dans une expression arithmétique ou logique.

Observation 5.4

Cette observation complète l'observation 5.3[p.106] dans le cas inter-processus. Dans le cas d'une action $[b]\ c!s(e)$ on fait distinction entre les deux cas : $c \in C^{\text{int}}$ ou $c \in C^{\text{ext}}$.

Dans le premier cas $Use([b]\ c!s(e),\cdot,\cdot) = vars(b)$ parce que l'ensemble $vars(e)$ sera pris en compte dans la(les) transition(s) inter-processus étiquetées par des étiquettes $x := e$.

Dans le deuxième cas on est toujours dans le cas de l'observation 5.3[p.106]. ▪

Lemme 5.11

$(\forall q_p)(q_p \in Q_p \Rightarrow Rlv(q_p) \subseteq X_p)$. ▪

Observation 5.5

Soit Rlv la solution du problème IP-RLV. Dans la suite on utilise au lieu de Rlv une autre fonction Rlv' qui calcule en plus l'utilité des signaux[22].

Soit L' le treillis complet : $2^{X \cup (C \times S)}$ où $X = \bigcup_{i=1}^{n} X_i$. On montre le calcul de cette fonction à la table 5.3[p.116]. On utilise les notations ci-dessus (pour la spécification, treillis, etc.). ▪

5.2.3 Propagation des constantes

Soient $\text{CONST}_i = (S_i, X_i \to \text{CONST}, F_i, T_i, \sqcup_{X_i \to \text{CONST}}, \to, Const_0^i, Const_i)$ les PFDS associés aux AECs $A_i, i \in 1 \,..\, \text{n}$ (ainsi qu'ils sont définis dans la section 5.1.4[p.106]).

La propagation des constantes inter-processus est l'instance IP-CONST du problème de flot de données inter-processus, dont les éléments sont montrés à la table 5.2[p.114].

Les fonctions de transfert $T : \Sigma \to F$ sont définies t.q. :

1. $T([b]\ y := e, i)(\ell_1, \ldots, \ell_i, \ldots, \ell_n) = (\ell_1, \ldots, T_i([b]\ y := e)(\ell_i), \ldots, \ell_n)$, $i \in 1 \,..\, \text{n}$,

22. elle sera aussi notée par Rlv

```
procedure CalculRlv (Rlv : Q → L; var Rlv' : Q → L')
  begin
    foreach (q ∈ Q)
      Rlv'(q) := Rlv(q);
    foreach (q -([b] c?s(x),·)-> q' ∧ x ∈ Rlv(q'))
      Rlv'(q') := Rlv'(q') ∪ {(c, s)};
    foreach (q -(x:=e,0)-> q')
      Rlv'(q) := Rlv'(q) ∪ (Rlv'(q') ∩ (C^int × S));
    foreach (q -([b] c!s(e),·)-> q')
      if (c ∈ C^ext ∧ vars(e) ⊆ Rlv_0(q))
        Rlv'(q) := Rlv'(q) ∪ {(c, s)};
  end.
```

TAB. 5.3 – *Utilité des données et signaux*

2. $T([b]\ c?s(y), i) = \begin{cases} \lambda f.[Env(c,s)/y]f & , \quad c \in C^{\text{ext}} \\ \lambda f.[\bot/y]f & , \quad c \in C^{\text{int}} \end{cases}$,

3. $T([b]\ c!s(e), i) = 1_L$,

4. Si $\langle x := e\rangle = \langle [b_1]\, c?s(x)\rangle \parallel \langle [b_2]\, c!s(e)\rangle$, avec $\langle [b_1]\, c?s(x)\rangle \in \Sigma(X_i)$ et $\langle [b_2]\, c!s(e)\rangle \in \Sigma(X_j)$ alors $T(x := e, 0) = \lambda f.[Eval_f(e)/x]\bar{\bot}_L$.

où les fonctions *Env* et $Eval_f$ sont celles de la section 5.1.4[p.106].

Si $Y : Q \to L$ est la solution du problème précédent (dans ce cas $lfp(Const)$) alors on prend au lieu de Y les fonctions $(Y_i)_{i\in 1..n}$ comme solution : $Y_i : Q_i \to (X_i \to \text{CONST})$, $Y_i(q_i)(x_i) = Y(q_i)(x_i)$.

On a aussi un résultat similaire à la lemme 5.7[p.107] :

Lemme 5.12 (Correction de la propagation des constantes)
Soient $\widehat{SP} = (\widehat{Q}, \widehat{\Sigma}, \{\xrightarrow{a} | \ a \in \widehat{\Sigma}\}, \hat{q}_0)$ et $Y = lfp(Const)$ la solution du problème CONST. Soit $\phi : \text{CONST} \to 2^{\mathbf{Z}}$ la fonction définie t.q. $\phi(\top) = \mathbf{Z}$, $\phi(\bot) = \emptyset$ et $\phi(x) = \{x\}$, pour $x \in \text{CONST}\backslash\{\bot, \top\}$. Le résultat suivant est vrai : $(\forall\ (\theta, \sigma, \rho))(\forall x)((\theta, \sigma, \rho) \in \widehat{Q} \wedge x \in \bigcup_{i=1}^{n} X_i \Rightarrow \sigma(x) \in \phi(Y(q)(x)))$. ∎

Preuve
La preuve se trouve dans l'annexe B[p.185].

□

5.2.4 Propagation des intervalles entiers

Soient $\text{INT}_i = (S_i, X_i \to \text{INT}, F_i, T_i, \sqcup_{X_i \to \text{INT}}, \to, Int_0^i, Int_i)$ les PFDs associés aux AECs $A_i, i \in 1 \,..\, \text{n}$ (ainsi qu'ils sont définis dans la section 5.1.5[p.108]).

La propagation des intervalles entiers inter-processus est l'instance IP-INT du problème de flot de données inter-processus, dont les éléments sont montrés à la table 5.2[p.114].

Les fonctions de transfert $T : \Sigma \to F$ sont définies t.q. :

1. $T([b]\ y := e, i)(\ell_1, \ldots, \ell_i, \ldots, \ell_n) = (\ell_1, \ldots, T_i([b]\ y := e)(\ell_i), \ldots, \ell_n)$, $i \in 1 \,..\, \text{n}$,
2. $T([b]\ c?s(y), i) = \begin{cases} \lambda f.[Env(c,s)/y]f & , \quad c \in C^{\text{ext}} \\ \lambda f.[\bot/y]f & , \quad c \in C^{\text{int}} \end{cases}$,
3. $T([b]\ c!s(e), i) = 1_L$,
4. Si $\langle x := e\rangle = \langle [b_1]\, c?s(x)\rangle \parallel \langle [b_2]\, c!s(e)\rangle$, avec $\langle [b_1]\, c?s(x)\rangle \in \Sigma(X_i)$ et $\langle [b_2]\, c!s(e)\rangle \in \Sigma(X_j)$ alors $T(x := e, 0) = \lambda f.[Eval_f(e)/x]\bar{\perp}_L$.

où les fonctions Env et $Eval_f$ sont celles de la section 5.1.5[p.108].

Si $Y : Q \to L$ est une approximation du point fixe $lfp(Int)$ alors on prend au lieu de Y les fonctions $(Y_i)_{i \in 1..n}$ comme solution : $Y_i : Q_i \to (X_i \to \text{INT})$, $Y_i(q_i)(x_i) = Y(q_i)(x_i)$.

Lemme 5.13
Soient $\widehat{SP} = (\widehat{Q}, \widehat{\Sigma}, \{\xrightarrow{a} | \ a \in \widehat{\Sigma}\}, \hat{q}_0)$ et Y la solution du problème IP-INT. Soit $\phi : \text{INT} \to 2^{\mathbf{Z}}$ la fonction définie t.q. $\phi(\top) = \mathbf{Z}$, $\phi(\bot) = \emptyset$ et $\phi((a,b)) = \{a, a+1, \ldots, b\}$. Le résultat suivant est vrai : $(\forall\ (q,\sigma,\rho))(\forall x)((q,\sigma,\rho) \in \widehat{Q} \wedge x \in \bigcup_{i=1}^{n} X_i \Rightarrow \sigma(x) \in \phi(Y(q)(x)))$. ∎

Preuve
Preuve similaire à la lemme 5.12[p.116]. □

5.3 Applications

Les invariants peuvent être utilisés pour simplifier les spécifications, en éliminant *le code mort* ([Muc97]) : les transitions qui ne seront jamais franchissable au moment de la simulation (ou exécution).

5.3.1 Recouvrement des variables actives

Il s'agit d'une simplification des spécifications fondée sur les solutions du problème IP-LIVE.

Définition 5.3 (Équivalence LIVE)
Soient $\widehat{SP} = (\hat{Q}, \hat{\Sigma}, \{\xrightarrow{a} | \ a \in \hat{\Sigma}\}, \hat{q_0})$ le modèle de la spécification
$SP = (S, C, (X_p, (Q_p, \Sigma(X_p), \{\xrightarrow{a}_p | \ a \in \Sigma(X_p)\}, q_0^p))_{p \in P})$, $Live \ : \ \bigcup_{p \in P} Q_p \ \rightarrow \ 2^{\bigcup_{p \in P} X_p}$
les solutions du problème IP-LIVE.

La relation d'équivalence LIVE est $\overset{\ell}{\equiv} \ \subseteq \hat{Q} \times \hat{Q}$ définie comme suit : si $(\theta, \sigma_i, \rho) \in \hat{Q}$, $1 \leq i \leq 2$ alors $(\theta, \sigma_1, \rho) \overset{\ell}{\equiv} (\theta, \sigma_2, \rho)$ ssi $(\forall x)(x \in Live(\theta) \Rightarrow \sigma_1(x) = \sigma_2(x))$. ▪

Lemme 5.14
La relation d'équivalence LIVE $\overset{\ell}{\equiv}$ est une relation de simulation :

$$
\begin{array}{l}
(\forall p_1)(\forall p_2)(\forall a) \\
\quad (\quad p_1 \overset{\ell}{\equiv} \ p_2 \wedge \ a \in \Sigma \ \wedge \\
\qquad (\forall q_1)(p_1 \xrightarrow{a} q_1 \Rightarrow \\
\qquad\quad (\exists q_2)(p_2 \xrightarrow{a} q_2 \ \wedge \ q_1 \overset{\ell}{\equiv} \ q_2) \ ) \\
\quad) \ .
\end{array}
$$

▪

Preuve
On montre le résultat en utilisant une induction en fonction de la forme de l'étiquette a. En fonction de la transition $p_1 \xrightarrow{a} q_1$ on identifie les règles $3.11_{[p.49]} \div 3.16_{[p.49]}$ qui l'ont pu générer et ensuite on montre que les mêmes règles peuvent être appliquées dans le cas de p_2 et on obtient une transition $p_2 \xrightarrow{a} q_2$ t.q. $q_1 \overset{\ell}{\equiv} \ q_2$. La preuve complète se trouve dans l'annexe $B_{[p.185]}$.

□

Lemme 5.15
La relation d'équivalence $\overset{\ell}{\equiv}^{-1}$ est une relation de simulation :

$$
\begin{array}{l}
(\forall p_1)(\forall p_2)(\forall a) \\
\quad (\quad p_1 \overset{\ell}{\equiv} \ p_2 \wedge \ a \in \Sigma \ \wedge \\
\qquad (\forall q_2)(p_2 \xrightarrow{a} q_2 \Rightarrow \\
\qquad\quad (\exists q_1)(p_1 \xrightarrow{a} q_1 \ \wedge \ q_1 \overset{\ell}{\equiv} \ q_2) \ ) \\
\quad) \ .
\end{array}
$$

▪

Preuve

La preuve est similaire à celle de la lemme 5.14[p.118] (on utilise les implications inverses).

□

Corollaire 5.1

La relation $\overset{\ell}{\equiv}$ est une relation de bisimulation. ▪

Pour les résultats suivants on utilise les notations :

Notation 5.2

1. $\widehat{SP} = (\hat{Q}_1, \hat{\Sigma}, \{\xrightarrow{a}_1 | \ a \in \hat{\Sigma}\}, \hat{q}_0^1)$ est le modèle de la spécification SP.
2. $\widehat{SP}/_{\overset{\ell}{\equiv}} = (\hat{Q}_2, \hat{\Sigma}, \{\xrightarrow{a}_2 | \ a \in \hat{\Sigma}\}, \hat{q}_0^2)$, avec $\hat{Q}_2 = \hat{Q}_1/_{\overset{\ell}{\equiv}}$ et $[p]_{\overset{\ell}{\equiv}} \xrightarrow{a}_2 [q]_{\overset{\ell}{\equiv}}$ ssi $p \xrightarrow{a}_1 q$. L'état $\hat{q}_0^2$ est $[\hat{q}_0^1]_{\overset{\ell}{\equiv}}$.
3. $\widehat{SP} \downarrow k = (\hat{Q}_3, \hat{\Sigma}, \{\xrightarrow{a}_3 | \ a \in \hat{\Sigma}\}, \hat{q}_0^3)$, avec
   $\hat{Q}_3 = \{(\theta, [k/\neg Live(\theta)]\sigma, \rho) \mid (\theta, \sigma, \rho) \in \hat{Q}_1\}$, $\hat{q}_0^3 = \hat{q}_0^1 \downarrow k$ et
   $(\theta, [k/\neg Live(\theta)]\sigma, \rho') \xrightarrow{a}_3 (\theta', [k/\neg Live(\theta')]\sigma', \rho')$ ssi $(\theta, \sigma, \rho) \xrightarrow{a}_1 (\theta', \sigma', \rho')$.
   La notation $[k/\neg Live(\theta)]\sigma$ signifie que la constante k substitue les variables qui ne sont pas *vivantes* dans l'état de contrôle θ.

▪

Lemme 5.16

$\widehat{SP} \overset{\ell}{\equiv} \widehat{SP}/_{\overset{\ell}{\equiv}}$. ▪

Lemme 5.17

$\widehat{SP}$ et $\widehat{SP} \downarrow k$ sont isomorphes. ▪

Observation 5.6

Dans IF ([Boz99]) cette technique s'applique syntaxiquement. Les commandes de IF ont une garde et une ou plusieurs actions séquentiels (ex. $q \xrightarrow{[b]a_1;\ldots;a_n} q'$). Pour chaque $x \notin Live(q')$ on ajoute à la fin de la séquence $a_1; \ldots; a_n$ l'action d'affectation $x := k$. ▪

5.3.2 Recouvrement des variables temporisées

Une application intéressante de la propagation de constantes permet la diminution du nombre de variables temporisées dans les automates temporisés ([AD94]). Concernant les automates temporisés il existe aujourd'hui plusieurs outils (académiques : KRONOS [Yov97], UPPAAL [ABB+01] ou industriels : *Object*GEODE [Tel]) pour la vérification et la validation des systèmes contenant des variable temporisées (horloges et/ou temporisateurs).

Dans la présentation des résultats ci-dessous on a opté pour une variante d'automate temporisé qui a été implémenté dans IF ([Boz99]) : il s'agit des automates temporisés avec échéances[23] ([BS97]).

Il faut mentionner que, en raison de la sémantique des variables temporisées, celles-ci ont une contribution importante dans l'augmentation des ressources de calcul (mémoire, temps) demandées pour la vérification des spécifications contenant tel type de variables. Pour les automates temporisés il existe une analyse d'activité et d'égalité des horloges ([DY96]). Notre but et de détecter dans chaque état quelles sont les variables temporisées qui gardent une différence constante entre eux. On a réduit ce problème à une propagation de constantes, en introduisant des nouvelles variables, celles-ci représentant la différence entre chaque paire de variables temporisées. Ensuite, pour ces variables on a appliqué la propagation des constantes. L'invariant ainsi calculé permet le recouvrement des variables temporisées.

Automates temporisés

Définition 5.4 (Syntaxe)

Un *automate temporisé* est un STE:

$(H, Q, \Sigma \times \Phi(H) \times 2^{H\times\mathbf{Z}}, \{\xrightarrow{a} | \ a \in \Sigma \times \Phi(H) \times 2^{H\times\mathbf{Z}}\}, q_0)$, où H est l'ensemble des variables temporisées (ou horloges), $\Phi(H)$ est l'ensemble de toutes les contraintes de la forme $c \leq h$, $h \leq c$, $\neg\delta$, $\delta \wedge \delta$, où $c \in \mathbf{Z}$, $h \in H$, $\delta \in \Phi(H)$.

Si $a \in \Sigma \times \Phi(H) \times 2^{H\times\mathbf{Z}}$ alors $a[3] \subseteq H \times \mathbf{Z}$ est l'ensemble des horloges *armées* dans la transition $\xrightarrow{a}$. On impose sur $a[3]$ la restriction suivante : $(\forall(h_1, c_1))(\forall(h_2, c_2))((h_1, c_1) \in a[3] \wedge (h_2, c_2) \in a[3] \Rightarrow h_1 \neq h_2)$. ▪

Définition 5.5 (Sémantique)

La *sémantique paresseuse* de l'automate temporisé

$(H, Q, \Sigma \times \Phi(H) \times 2^{H\times\mathbf{Z}}, \{\xrightarrow{a} | \ a \in \Sigma \times \Phi(H) \times 2^{H\times\mathbf{Z}}\}, q_0)$ est le STE

23. Sans restreindre la généralité de l'approche on considéré seulement le cas de la sémantique paresseuse ou les actions temporisées peuvent s'exécuter à n'importe quel moment

$\widehat{A}_\epsilon = (\widehat{Q}, \widehat{\Sigma}, \{ \overset{a}{\multimap\!\!\rightarrow} \mid a \in \widehat{\Sigma}\}, \hat{q}_0)$, où $\epsilon \in \mathbf{Z}$ est la constante de la progression du temps, $\widehat{\Sigma} = \Sigma \cup \{tick\}$ et les ensembles $\widehat{Q} \subseteq Q \times (H \to \mathbf{Z})$ et $\overset{a}{\multimap\!\!\rightarrow}$ sont obtenus inductivement en appliquant les règles suivantes :

$$\frac{-}{\hat{q}_0 = (q_0, 0^{|H|}) \in \widehat{Q}} \tag{5.1}$$

$$\frac{(q,\nu) \in \widehat{Q} \quad q \overset{(a, g, (h_{i_1}, c_{i_1}), \ldots, (h_{i_j}, c_{i_j}))}{\longrightarrow} q' \quad \nu(g) = \mathbf{t}}{\hat{q}' = (q', [c_{i_1}/h_{i_1}, \ldots, c_{i_j}/h_{i_j}]\nu) \in \widehat{Q} \quad (q,\nu) \overset{a}{\multimap\!\!\rightarrow} \hat{q}'} \tag{5.2}$$

$$\frac{(q,\nu) \in \widehat{Q}}{\hat{q}' = (q, \nu + \epsilon) \in \widehat{Q} \quad (q,\nu) \overset{tick}{\multimap\!\!\rightarrow} \hat{q}'} \tag{5.3}$$

Ci-dessus $\nu + \epsilon$ est une fonction de $H \to \mathbf{Z}$, $(\nu + \epsilon)(h) = \nu(h) + \epsilon$, $\forall h \in H$. ▪

Observation 5.7

Si deux horloges sont armées dans la même transition alors elles gardent une différence constante entre elles dans toute exécution du système. ▪

Recouvrement des horloges

Problème 5.1

Soit $A = (H, Q, \Sigma \times \Phi(H) \times 2^{H \times \mathbf{Z}}, \{\overset{a}{\longrightarrow} \mid a \in \Sigma \times \Phi(H) \times 2^{H \times \mathbf{Z}}\}, q_0)$ un automate temporisé. On voudrait obtenir un nouvel automate temporisé

$A' = (H', Q, \Sigma \times \Phi(H') \times 2^{H' \times \mathbf{Z}}, \{ \overset{a}{\underset{\setminus h}{\longrightarrow}} \mid a \in \Sigma \times \Phi(H') \times 2^{H' \times \mathbf{Z}}\}, q_0)$, avec $H' \subseteq H$ et $\overset{a}{\underset{\setminus h}{\longrightarrow}}$ obtenue de $\overset{a}{\longrightarrow}$ en recouvrant les horloges de H par ceux de H' et $\widehat{A_\epsilon}$ et $\widehat{A'_\epsilon}$ soient bisimilaires. ▪

Soient les ensembles des horloges $H = \{h_1, \ldots, h_n\}$ et des différences entre elles $H_d = \{h_i - h_j \mid h_i, h_j \in H \wedge 1 \leq i < j \leq n\}$.

Soit $n_d = |H_d|$. Soient f, g deux fonctions bijectives : $f : \{(i,j) \mid (i,j) \in \mathbf{N}^2 \wedge 1 \leq i < j \leq n\} \to \{1, \ldots, n_d\}$ et $g : \{1, \ldots, n_d\} \to \{(i,j) \mid (i,j) \in \mathbf{N}^2 \wedge 1 \leq i < j \leq n\}$.

On considère ATEMP un PFD, instance de CONST :

1. $S = (Q, \Sigma \times \Phi(H) \times 2^{H \times \mathbf{Z}}, \{\overset{a}{\longrightarrow} \mid a \in \Sigma \times \Phi(H) \times 2^{H \times \mathbf{Z}}\}, q_0)$.
2. $L = H_d \to$ CONST.
3. L'espace des fonctions est $F = \{\lambda f.[v/h]f \mid v \in \text{CONST} \wedge h \in H_d \wedge f \in L\}$.

4. Les fonctions de transfert $T : \Sigma \times \Phi(H) \times 2^{H \times \mathbf{Z}} \to F$ ont la définition suivante :
   $T(a, g, (h_{i_1}, c_{i_1}), \ldots, (h_{i_j}, c_{i_j})) = \lambda f.[(c_{i_\ell} - c_{i_m} / h_{f(i_\ell, i_m)})_{1 \leq \ell < m \leq j}]$.
5. L'opérateur de combinaison $\uplus$ est l'opérateur $\sqcup_L$.
6. Le problème de flot est en avant ($\delta = \rightarrow$).
7. L'approximation initiale est $Const_0 : Q \to L$, t.q. : $Const_0(q_0) = \bar{\top}_L$ et $Const_0(q') = \bar{\bot}_L$, pour tout $q' \neq q_0$.
8. L'opérateur de flot est $Const : (Q \to L) \to (Q \to L)$ où, pour tout $q \in Q$:

$$Const(Z)(q) = Const_0(q) \sqcup_L \bigsqcup_{\substack{a \in \Sigma \\ q' \in Pre_a(q)}}{}_L \; T(a)(Z(q')).$$

Lemme 5.18
ATEMP est un PFD monotone et l'espace des fonctions F a les propriétés demandées dans la définition 5.1[p.97]. ▪

Dans la suite on utilise le prédicat $Use : H \times (\Sigma \times \Phi(H) \times 2^{H \times \mathbf{Z}}) \to \{\mathbf{t}, \mathbf{f}\}$ avec la définition suivante :
$Use(h, a) = \mathbf{t} \iff (\exists (x, v))((x, v) \in a[3] \wedge h = x)) \vee Use'(h, a[2]) = \mathbf{t}$.
Le prédicat $Use' : H \times \Phi(H) \to \{\mathbf{t}, \mathbf{f}\}$ est défini inductivement :

1. $Use'(h, c \leq x) = \mathbf{t} \iff x = h$,
2. $Use'(h, x \leq c) = \mathbf{t} \iff x = h$,
3. $Use'(h, \neg\delta) = \neg Use'(h, \delta)$,
4. $Use'(h, \delta_1 \wedge \delta_2) = Use'(h, \delta_1) \wedge Use'(h, \delta_2)$.

Algorithme 9 (Recouvrement des horloges)
`Entrée :`
A, un automate temporisé, $Const$, la solution du problème de flot de données ATEMP et Use, le prédicat ci-dessus.
`Sortie :`
A', un automate temporisé.
`Méthode :`

La procédure *Recouvrement* (tables 5.4[p.124], 5.5[p.125], 5.6[p.126]). On commence par le calcul des règles de réécriture. Celles-ci sont stockées dans les files du vecteur RW (qui associe à chaque transition une file avec les règles de réécriture à appliquer). Les horloges de l'automate temporisé A sont introduites dans la file *Orol* et ensuite dans $|A.H|$ pas on essaie d'exprimer un horloge avec les horloges de *Orol* : supposons qu'on est au pas i. Cela signifie qu'on a examiné les premières $i-1$ horloges de *Orol* initiale (celle de la ligne $*$ dans la table 5.4[p.124]). Une partie d'entre elles sont stockées dans l'ensemble *Elim* et cela parce qu'au moment de leur traitement on a pu les réécrire dans chaque transition par une horloge trouvée dans *Orol* (c.à-d. la procédure *CalculRegles* de la table 5.5[p.125] a pu trouver, pour chaque transition ou l'horloge était *utilisée*, une autre horloge t.q. entre elles il y avait une différence constante - cf. *Const*). Les horloges qui n'ont pas pu être réécrites sont réintroduites dans *Orol* mais elles serviront seulement au calcul d'éventuelles règles de réécriture, elles ne seront pas examinées à nouveau. À la fin de cette étape on applique les règles calculées (table 5.6[p.126]) et le nouvel ensemble des horloges est *Orol*.

▪

Lemme 5.19

Soient A un automate temporisé et A' l'automat temporisé fournit par l'algorithme 9[p.122] avec l'entrée A. Soient $\widehat{A_\epsilon} = (\widehat{Q}_1, \widehat{\Sigma}, \{\xrightarrow{a}_1 |\ a \in \widehat{\Sigma}\}, \hat{q}_0^1)$ et $\widehat{A'_\epsilon} = (\widehat{Q}_2, \widehat{\Sigma}, \{\xrightarrow{a}_2 |\ a \in \widehat{\Sigma}\}, \hat{q}_0^2)$. La relation $\sim_h = \{(p_1, p_2) \mid (p_1, p_2) \in \widehat{Q}_1 \times \widehat{Q}_2 \wedge p_i = (\theta, \nu_i), 1 \leq i \leq 2 \wedge (\forall h)(h \in H' \Rightarrow \nu_1(h) = \nu_2(h))\}$ est une relation de bisimulation. ▪

Preuve

Pour montrer le résultat on utilise une induction structurelle en fonction de l'étiquette a et des règles 5.2[p.121] et 5.3[p.121]. La preuve complète se trouve dans l'annexe B[p.187].

□

Théorème 5.4

L'algorithme 9[p.122] fournit un automat temporisé A' qui est une solution du problème 5.1[p.121]. ▪

Preuve

Dans la lemme précédente on a montré que $\widehat{A_\epsilon} \sim_h \widehat{A'_\epsilon}$.

Dans la procédure *Recouvrement* de la table 5.4[p.124] la relation $Elim \cup Orol = H$ est un invariant et à la fin de la procédure on obtient $H' \subseteq H$.

□

```
    procedure Recouvrement (A: AUTTEMP;
                  Const : A.Q → (A.H_d → CONST);
                  Use : A.H × A.Σ → bool;
                  var A' : AUTTEMP)
      type
        RULE = A.H × A.H × CONST ∪ {nil};
        RULES = file de RULE;
        TRANS = {—a→| a ∈ A.Σ};
      var
        Orol : file de A.H;
        Elim : ensemble de A.H;
        RW : TRANS → RULES;
        rw : TRANS → RULE;
        i, N : int;
        x : A.H;
        contradictions : bool;
      procedure CalculRegles;
      procedure ApplicationRegles;
      begin
        A'.Q := A.Q; A'.q_0 := A.q_0;
        foreach (x ∈ A.H)
          Orol.Push(x);
*       N = | Orol |; Elim := ∅; i := 0;
        while (i ≤ N ∧ ¬Orol.Empty())
          x := Orol.Pop();
          i := i + 1;
          contradictions := f;
          CalculRegles(x, rw, A, Const, Use);
          if (¬contradictions);
            Elim := Elim ∪ {x};
            foreach (q —a→ q')
              RW(q —a→ q').Push(rw(q —a→ q'));
          else
            Orol.Push(x);
          endif
        endwhile
        ApplicationRegles(A, RW, A');
        A'.H := Orol;
      end.
```

TAB. 5.4 – *Recouvrement des horloges*

```
procedure CalculRegles (x : A.H;
                        var rw : TRANS → RULE;
                        A : AUTTEMP;
                        Const : A.Q → (A.H_d → CONST);
                        Use : A.H × A.Σ → bool)
  var
    find : bool;
  begin
    foreach (a ∈ A.Σ)
      if (Use(x, a))
        foreach (q -a-> q')
          find := f;
          foreach (y ∈ Orol)
            if (Const(q)(x − y) ∉ {⊤, ⊥})
              rw(q -a-> q') := (x, y, Const(q)(x − y));
              find := t;
              break;
            endif
          endfor
          if (¬find)
            contradictions := t;
        endfor
      else
        foreach (q -a-> q')
          rw(q -a-> q') := nil;
      endif
  end.
```

TAB. 5.5 – *Calcul des règles*

```
procedure ApplicationRegles (A : AUTTEMP;
            var RW : TRANS → RULES;
            var A' : AUTTEMP)
  var
    rw : RULE;
  begin
    foreach (a ∈ A.Σ)
      foreach (q --a--> q')
        while (¬RW(q --a--> q').Empty())
          rw := RW(q --a--> q').Pop();
          if (rw ≠ nil)
            A'.Σ := A'.Σ ∪ {[rw[2] + rw[3]/rw[1]]a};
            --[rw[2]+rw[3]/rw[1]]a--\h--> := --[rw[2]+rw[3]/rw[1]]a--\h--> ∪{(q, q')};
          endif
        endwhile
  end.
```

TAB. 5.6 – *Application des règles*

5.3.3 Tranchage par rapport aux données utiles

Ce calcul est une extension de l'analyse des variables actives ([BFG99a]). On calcule, pour chaque processus, l'ensemble de variables utiles dans chaque état. L'utilité est définie par rapport avec les sorties de l'OT : une variable est utile dans un état si sa valeur dans cet état pourrait être utilisée pour calculer la valeur du paramètre d'un signal de sortie qui apparaît dans l'OT. On peut considérer, de manière similaire à l'analyse des variables actives, qu'une variable est utile dans un état ssi il y a un chemin qui commence dans cet état t.q. la variable est utilisée avant d'être redéfinie dans le chemin. Mais, dans notre cas, on considère qu'une variable est utilisée seulement dans les sorties externes, mentionnées dans l'OT, où dans les affectations (via des entrées internes) des variables utiles.

Les pas précédents sont détaillés dans l'algorithme suivant :

Algorithme 10

`Entrée :`

1. La spécification $SP = (S, C, P)$, avec $P \ni p = (X_p, (Q_p, \Sigma(X_p), \{\xrightarrow{a}_p |\ a \in \Sigma(X_p)\}, q_0^p))$,

2. L'OT abstrait $TP_L = (Q_{\rm tp}, \Upsilon(L), \{\stackrel{a}{\longrightarrow}_{\rm tp}|\ a \in \Upsilon(L)\}, q_0^{\rm tp})$, où L est le treillis CONST ou INT.

`Sortie` : La spécification $SP \setminus_{\rm o} \Upsilon^s(L) = (S, C, P \setminus_{\rm o} \Upsilon^s(L))$, avec
$P \setminus_{\rm o} \Upsilon^s(L) \ni p \setminus_{\rm o} \Upsilon^s(L) = (X'_p, (Q_p, \Sigma(X_p), \{\stackrel{a}{\not\longrightarrow_{\rm o}} p|\ a \in \Sigma(X_p)\}, q_0^p))$.
`Méthode` :

1. On calcule le AEC union $S = \bigcup_{p \in P} (X_p, (Q_p, \Sigma(X_p), \{\stackrel{a}{\longrightarrow}_p|\ a \in \Sigma(X_p)\}, q_0^p))$.

2. Soit la transition $q_p \stackrel{[b]\ c\ !\ s(e)}{\longrightarrow}_p q'_p$. Si $(\exists \ell \in L)(\stackrel{c\ !\ s(\ell)}{\longrightarrow}_{\rm tp} \neq \emptyset)$ ou $(\stackrel{c\ !\ s(\oslash)}{\longrightarrow}_{\rm tp} \neq \emptyset)$ alors $Rlv_0(q_p) := Rlv_0(q_p) \cup Use(e)$, où $Use(e)$ est l'ensemble des variables qui apparaissent dans e.

3. On calcule la solution Rlv du problème IP-RLV.

4. On utilise la solution Rlv pour effectuer un tranchage de la spécification SP par rapport avec les données utiles. On obtient ainsi une spécification simplifiée

   $SP \setminus_{\rm o} \Upsilon^s(L) = (S, C, P \setminus_{\rm o} \Upsilon^s(L))$, avec

   $P \setminus_{\rm o} \Upsilon^s(L) \ni p \setminus_{\rm o} \Upsilon^s(L) = (X'_p, (Q_p, \Sigma(X_p), \{\stackrel{a}{\not\longrightarrow_{\rm o}} p|\ a \in \Sigma(X_p)\}, q_0^p))$.

   Le processus $p \setminus_{\rm o} \Upsilon^s(L)$ a les mêmes états et état initial avec p mais il utilise seulement les variables utiles : $X'_p = \bigcup_{q_p \in Q_p} Rlv(q_p)$.

   Les transitions de $\{\stackrel{a}{\not\longrightarrow_{\rm o}} p|\ a \in \Sigma(X_p)\}$ se construisent en appliquant les règles suivantes :

$$\frac{q_p \stackrel{[b]\ x:=e}{\longrightarrow}_p q'_p \quad x \in Rlv(q'_p)}{q_p \stackrel{[b]\ x:=e}{\not\longrightarrow_{\rm o}} p\ q'_p} \tag{5.4}$$

$$\frac{q_p \stackrel{[b]\ x:=e}{\longrightarrow}_p q'_p \quad x \notin Rlv(q'_p)}{q_p \stackrel{[b]\ \tau}{\not\longrightarrow_{\rm o}} p\ q'_p} \tag{5.5}$$

$$\frac{q_p \stackrel{[b]\ c?s(x)}{\longrightarrow}_p q'_p \quad x \in Rlv(q'_p)}{q_p \stackrel{[b]\ c?s(x)}{\not\longrightarrow_{\rm o}} p\ q'_p} \tag{5.6}$$

$$\frac{q_p \overset{[b]\ c?s(x)}{\longrightarrow}_p q'_p \quad x \notin Rlv(q'_p)}{q_p \overset{[b]\ c?s(\Omega)}{\overset{\longrightarrow}{\backslash o}}_p q'_p} \tag{5.7}$$

$$\frac{q_p \overset{[b]\ c!s(e)}{\longrightarrow}_p q'_p \quad (c,s) \in Rlv(q_p)}{q_p \overset{[b]\ c!s(e)}{\overset{\longrightarrow}{\backslash o}}_p q'_p} \tag{5.8}$$

$$\frac{q_p \overset{[b]\ c!s(e)}{\longrightarrow}_p q'_p \quad (c,s) \notin Rlv(q_p)}{q_p \overset{[b]\ c!s(\oslash)}{\overset{\longrightarrow}{\backslash o}}_p q'_p} \tag{5.9}$$

Ci-dessus $vars : AE_X \cup BE_X \to 2^X$ est la fonction qui rend les variables de X qui apparaissent dans une expression arithmétique ou logique.

▪

Exemple 5.1 *Le tranchage par rapport aux données utiles, appliqué à la spécification et l'*OT *de la figure* 4.1[p.89]*, produit la spécification montrée à la figure* 5.2[p.129].

Les transitions étiquetées avec $y := 1$ *et* $y := y * i$ *sont ré-étiquetées avec* τ *et la sortie* $co!pa(y)$ *devient* $co!pa(\oslash)$ *parce que* y *n'appartient pas à* Rlv_0 *de l'état source de la transition étiquetée* $co!pa(\oslash)$.

Observation 5.8

Dans pratique il est parfois impossible d'appliquer la règle 5.9[p.128] pour les variables de certains types. Dans ce cas ces variables deviennent utiles et on peut les ajouter aux ensembles Rlv_0. ▪

Observation 5.9

Le tranchage par rapport aux données utiles conserve le modèle de la spécification jusqu'aux valeurs concrètes des paramètres des signaux qui ne sont pas observés dans l'OT. On définit le renommage du modèle de la spécification $\widehat{SP}$ par rapport aux actions externes $\Upsilon^s(L)$ de la manière suivante : chaque action externe visible qui n'est pas spécifiée dans l'OT (c.à-d. $\overset{c!s(\oslash)}{\longrightarrow}_{\text{tp}} = \emptyset$) est renommée dans $c!s(\oslash)$. Les autres actions ne changent pas.

On utilise la règle

$$\frac{p \overset{c!s(v)}{\longrightarrow} q \quad \overset{c!s(\oslash)}{\longrightarrow}_{\text{tp}} = \emptyset}{p \overset{c!s(\oslash)}{\longrightarrow} q} \tag{5.10}$$

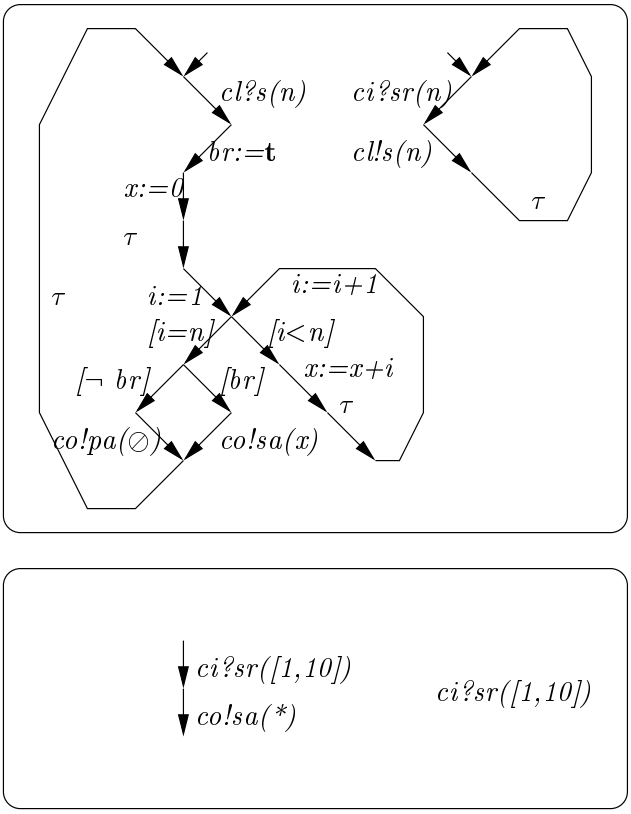

FIG. 5.2 – *Tranchage par rapport aux données utiles*

Ainsi, on ne considère pas la valeur exacte des paramètres des sorties qui n'apparaissent pas dans l'OT.

Le modèle renommé est dénoté par $\widehat{SP} \downarrow \Upsilon^s(L)$. ▪

Notation 5.3

Dans la suite on utilise les notations suivantes :

1. $SP = (S, C, (X_p, (Q_p, \Sigma(X_p), \{\xrightarrow{a}_p |\ a \in \Sigma(X_p)\}, q_0^p))_{p \in P})$.
2. $SP \setminus_{\mathrm{o}} \Upsilon^s(L) = (S, C, (X'_p, (Q_p, \Sigma(X_p), \{\xrightarrow[\setminus_{\mathrm{o}}]{a}_p |\ a \in \Sigma(X_p)\}, q_0^p))_{p \in P \setminus_{\mathrm{o}} \Upsilon^s(L)})$.
3. $\widehat{SP} \downarrow \Upsilon^s(L) = (Q_1, \Sigma, \{\xrightarrow{a}_1 |\ a \in \Sigma\}, q_1^0)$.
4. $SP \widehat{\setminus_{\mathrm{o}} \Upsilon^s}(L) = (Q_2, \Sigma, \{\xrightarrow{a}_2 |\ a \in \Sigma\}, q_2^0)$.

▪

Définition 5.6

1. Soit $\overset{\mathrm{v}}{\equiv} \subseteq Q_1 \times Q_2$ la relation d'équivalence définie comme suit : si $(\theta, \sigma_i, \rho_i) \in Q_i$, $1 \leq i \leq 2$ alors $(\theta, \sigma_1, \rho_1) \overset{\mathrm{v}}{\equiv} (\theta, \sigma_2, \rho_2)$ ssi $(\forall x)(x \in Rlv(\theta) \Rightarrow \sigma_1(x) = \sigma_2(x))$.

   Parce que les contextes des files ρ_1 et ρ_2 n'interviennent pas, on les omet (on écrit $(\theta, \sigma_1) \overset{\mathrm{v}}{\equiv} (\theta, \sigma_2)$).

2. Soit $\overset{\mathrm{q}}{\equiv} \subseteq Q_1 \times Q_2$ la relation d'équivalence définie comme suit : si $(\theta, \sigma_i, \rho_i) \in Q_i$, $1 \leq i \leq 2$ alors $(\theta, \sigma_1, \rho_1) \overset{\mathrm{q}}{\equiv} (\theta, \sigma_2, \rho_2)$ ssi $(\forall c \in C^{\mathrm{int}})(\rho_1(c) = \rho_2(c)\ mod \oslash)$, où $w_1 = w_2\ mod \oslash$ ssi $w_1 = w_2 = \epsilon$ ou $w_1 = s(v_1).w'_1$, $w_2 = s(v_2).w'_2$ et $w'_1 = w'_2\ mod \oslash$ et ($v_1 = v_2$ ou $v_1 = \oslash$ ou $v_2 = \oslash$).

   Parce que les contextes des variables σ_1 et σ_2 et l'état θ n'interviennent pas, on les omet (on écrit $\rho_1 \overset{\mathrm{q}}{\equiv} \rho_2$).

3. Soit $\sim_{\mathrm{o}} \subseteq Q_1 \times Q_2$ la relation $\{(q_1, q_2) \mid q_i = (\theta, \sigma_i, \rho_i), 1 \leq i \leq 2 \wedge (\theta, \sigma_1) \overset{\mathrm{v}}{\equiv} (\theta, \sigma_2) \wedge (\theta, \rho_1) \overset{\mathrm{q}}{\equiv} (\theta, \rho_2)\}$.

▪

Lemme 5.20
Si $\theta \xrightarrow{\alpha} \theta'$ et $(\theta, \sigma_1) \overset{\text{v}}{\equiv} (\theta, \sigma_2)$ alors

$$\begin{array}{l} (\theta', \left[(v_x)_{x \in Def(\alpha) \setminus Rlv(\theta')} / Def(\alpha) \setminus Rlv(\theta')\right] \sigma_1) \\ \quad \overset{\text{v}}{\equiv} \\ (\theta', \left[(v_x)_{x \in Def(\alpha) \setminus Rlv(\theta')} / Def(\alpha) \setminus Rlv(\theta')\right] \sigma_2) \ . \end{array}$$

Ci-dessus $Def(\alpha) \setminus Rlv(\theta') = \emptyset$ signifie $(\theta', \sigma_1) \overset{\text{v}}{\equiv} (\theta', \sigma_2)$. ▪

Preuve
Conformément à la section 5.2.2[p.113], on a $Rlv(\theta) \supseteq Rlv(\theta') \setminus Def(\alpha) \cup Use(\alpha)$.
Donc $Rlv(\theta') \subseteq Rlv(\theta) \cup Def(\alpha)$ et parce que les valeurs des variables de $Def(\alpha) \setminus Rlv(\theta')$ sont remplacées avec les mêmes valeurs ainsi dans σ_1 et dans σ_2 on obtient la conclusion de la lemme.

□

Les deux lemmes suivants montrent que la relation $\sim_{\text{o}}$ est une relation de simulation (de $\widehat{SP} \downarrow \Upsilon^s(L)$ par $SP \widehat{\backslash_{\text{o}} \Upsilon^s}(L)$ mais aussi réciproquement) et donc c'est une bisimulation (théorème 5.5[p.132]).

Lemme 5.21

$$\begin{array}{l} (\forall p_1)(\forall p_2)(\forall a) \\ \quad (\quad p_1 \sim_{\text{o}} p_2 \wedge \ a \in \Sigma \wedge \\ \qquad (\forall q_1)(q_1 \in Q_1 \wedge p_1 \xrightarrow{a}_1 q_1 \Rightarrow \\ \qquad\quad (\exists q_2)(q_2 \in Q_2 \ \wedge \ p_2 \xrightarrow{a}_2 q_2 \ \wedge \ q_1 \sim_{\text{o}} q_2) \ ) \\ \quad) \ . \end{array}$$

▪

Preuve
Soient $a \in \Sigma$, $p_1 \sim_{\text{o}} p_2$ et $q_1 \in Q_1$ t.q. $p_1 \xrightarrow{a}_1 q_1$.
Parce que $(\theta, \sigma_1, \rho_1) = p_1 \sim_{\text{o}} p_2$ on a $p_2 = (\theta, \sigma_2, \rho_2)$ et $(\theta, \sigma_1) \overset{\text{v}}{\equiv} (\theta, \sigma_2)$ et $\rho_1 \overset{\text{q}}{\equiv} \rho_2$.
On montre l'existence de $q_2 \in Q_2$ t.q. $p_2 \xrightarrow{a}_2 q_2$ et $q_1 \sim_{\text{o}} q_2$ en utilisant une induction en fonction de la forme de l'étiquette a et les règles 3.11[p.49] ÷ 3.16[p.49] et 5.10[p.128]. La preuve complète se trouve dans l'annexe B[p.189].

□

Lemme 5.22

$$
\begin{array}{l}
(\forall p_1)(\forall p_2)(\forall a) \\
\quad (\quad p_1 \sim_{\mathrm{o}} p_2 \wedge \ a \in \Sigma \wedge \\
\qquad (\forall q_2)(q_2 \in Q_2 \wedge p_2 \xrightarrow{a}_2 q_2 \Rightarrow \\
\qquad\quad (\exists q_1)(q_1 \in Q_1 \ \wedge \ p_1 \xrightarrow{a}_1 q_1 \ \wedge \ q_1 \sim_{\mathrm{o}} q_2) \ ) \\
\quad) \ .
\end{array}
$$

∎

Preuve

Soient $a \in \Sigma$, $p_1 \ \sim_{\mathrm{o}} \ p_2$ et $q_2 \in Q_2$ t.q. $p_2 \xrightarrow{a}_2 q_2$.

À cause du fait que $(\theta, \sigma_2, \rho_2) = p_2 \ \sim_{\mathrm{o}} \ p_1$ on a $p_1 = (\theta, \sigma_1, \rho_1)$ et $(\theta, \sigma_1) \overset{\mathrm{v}}{\equiv} (\theta, \sigma_2)$ et $\rho_1 \overset{\mathrm{q}}{\equiv} \rho_2$.

On montre l'existence de $q_1 \in Q_1$ t.q. $p_1 \xrightarrow{a}_1 q_1$ et $q_1 \ \sim_{\mathrm{o}} q_2$ en utilisant une induction sur la forme de l'étiquette a et les règles $3.11_{[\mathrm{p.49}]} \div 3.21_{[\mathrm{p.50}]}$ et $5.4_{[\mathrm{p.127}]} \div 5.9_{[\mathrm{p.128}]}$.

On raisonne seulement dans le cas des règles $3.17_{[\mathrm{p.50}]} \div 3.21_{[\mathrm{p.50}]}$ parce que dans les cas des règles $3.11_{[\mathrm{p.49}]} \div 3.16_{[\mathrm{p.49}]}$ on peut utiliser les implications inverses de la lemme $5.21_{[\mathrm{p.131}]}$. La preuve complète se trouve dans l'annexe $B_{[\mathrm{p.193}]}$.

□

Théorème 5.5 (Correction de la tranchage par rapport aux données utiles)

Soient $SP = (S, C, P)$ une spécification, TP_L un OT abstrait.

Le modèle de SP renommé par rapport aux sorties observables $\Upsilon^s(L)$ et le modèle de $SP \setminus_{\mathrm{o}} \Upsilon^s(L)$ sont bisimilaires. ∎

Preuve

Conformément aux deux lemmes précédentes ($5.21_{[\mathrm{p.131}]}$ et $5.22_{[\mathrm{p.131}]}$), $\sim_{\mathrm{o}}$ est une relation de bisimulation. Les états q_1^0 et q_2^0 sont les mêmes donc $q_1^0 \ \sim_{\mathrm{o}} q_2^0$ et donc $\widehat{SP} \downarrow \Upsilon^s(L) \ \sim_{\mathrm{o}} \widehat{SP \setminus_{\mathrm{o}} \Upsilon^s(L)}$.

□

Observation 5.10

Le tranchage par rapport aux données utiles exploite les variables utiles seulement à un niveau syntaxique, en éliminant celles qui ne sont pas utiles et leurs dépendances dans la spécification.

Néanmoins c'est possible de les utiliser plus profondément, par exemple en utilisant une technique de réinitialisation des variables inutiles avec une valeur par défaut dès qu'elles deviennent inutiles et obtenant ainsi un modèle bisimilaire et réduit. ∎

5.3.4 Tranchage par rapport aux contraintes

Dans la suite on considère que les *contraintes* s'expriment par les élément des treillis complets INT et CONST. Ces contraintes correspondent aux contraintes trouvées dans TTCN ([fS92a]).

On utilise les contraintes attachées aux signaux de l'ensemble des entrées contrôlables et les contraintes attachées aux entrées de l'OT abstrait pour simplifier la spécification. Cette simplification est réalisée dans trois pas :

- Ces contraintes sont attachées d'abord aux entrées de la spécification qui, dans une exécution synchrone de la spécification et de l'OT, *peuvent* se synchroniser avec celles de l'OT.
- Ensuite, en utilisant les problèmes de flot intra et inter-processus (CONST et INT présentés ci-dessus), on calcule une approximation conservatrice, dans chaque état de contrôle, des valeurs des variables.
- Finalement, cette information est utilisée pour évaluer les gardes des transitions et pour éliminer celles qui ne seront jamais franchissables à l'exécution.

Les pas précédents sont détaillés dans l'algorithme suivant :

Algorithme 11

`Entrée :`

1. La spécification $SP = (S, C, P)$, avec $P \ni p = (X_p, (Q_p, \Sigma(X_p), \{\xrightarrow{a}_p |\ a \in \Sigma(X_p)\}, q_0^p))$,
2. L'OT abstrait $TP_L = (Q_{\mathrm{tp}}, \Upsilon(L), \{\xrightarrow{a}_{\mathrm{tp}} |\ a \in \Upsilon(L)\}, q_0^{\mathrm{tp}})$, où L est le treillis CONST ou INT.
3. L'ensemble des entrées contrôlables Σ_{f}.

`Sortie :` La spécification $SP \setminus_{\mathrm{c}} \Sigma_{\mathrm{f}} = (S, C, P \setminus_{\mathrm{c}} \Sigma_{\mathrm{f}})$, avec
$P \setminus_{\mathrm{c}} \Sigma_{\mathrm{f}} \ni p \setminus_{\mathrm{c}} \Sigma_{\mathrm{f}} = (X_p, (Q'_p, \Sigma(X_p), \{\xrightarrow[\setminus_{\mathrm{c}}]{a}_p |\ a \in \Sigma(X_p)\}, q_0^p))$.

`Méthode :`

1. Les contraintes de Σ_{f} et celles attachées aux entrées de l'OT abstrait sont attachées aux entrées de la spécification qui, dans une exécution synchrone de la spécification et de l'OT, *peuvent* se synchroniser avec celles de l'OT.

   Ceci est fait formellement par le moyen des fonctions *Env* des problèmes IP-CONST et IP-INT :

$$Env : C \times S \to \text{CONST},\ Env(c,s) = \begin{cases} \ell & , \quad c\ ?\ s(\ell) \in \Sigma_{\mathrm{f}} \\ \top & , \quad \text{sinon} \end{cases}$$

$$Env : C \times S \to \text{INT},\ Env(c,s) = \begin{cases} \ell & , \quad c\ ?\ s(\ell) \in \Sigma_{\mathrm{f}} \\ (-\infty, +\infty) & , \quad \text{sinon} \end{cases}$$

2. On calcule les (approximations des) solutions $(Const_p)_{p\in P}$ et $(Int_p)_{p\in P}$ des problèmes IP-CONST et IP-INT. On les dénotent de manière générique par

   $(Val_p : Q_p \to (X_p \to \text{VAL}))_{p\in P}$.

3. On utilise les fonctions $(Val_p)_{p\in P}$ pour évaluer les gardes des transitions et pour éliminer celles qui ne seront jamais franchissables. On obtient ainsi une spécification simplifiée $SP \setminus_{\mathrm{c}} \Sigma_{\mathrm{f}}$.

   Pour la construction des processus $p \setminus_{\mathrm{c}} \Sigma_{\mathrm{f}}$ on applique les règles suivantes :

$$\frac{-}{q_p^0 \in Q'_p} \tag{5.11}$$

$$\frac{q_p \in Q'_p \quad q_p \xrightarrow{[b]\ \alpha}_p q'_p \quad Val_p(q_p)(b) \neq \{\mathbf{f}\}}{q'_p \in Q'_p \quad q_p \xrightarrow[\setminus_{\mathrm{c}}]{[b]\ \alpha}_p q'_p} \tag{5.12}$$

▪

Exemple 5.2 *Le tranchage par rapport aux contraintes, appliqué à la spécification et l'*OT *de la figure* 5.2[p.129]*, produit la spécification montrée à la figure* 5.3[p.135].

La valeur **t** *de la variable br est propagée vers l'état source de la transition avec la garde* $[\neg br]$ *et ainsi on se rend compte que cette transition et la suivante* $co!pa(\Omega)$ *ne seront jamais tirables au moment de l'exécution.*

Ces transitions sont éliminées de la spécification.

L'approximation obtenue par le problème de flot INT*, avec l'approximation initiale dans l'ensemble des entrées contrôlables* $ci?sr([1,10])$ *pour la valeur du paramètre* x *dans la transition* $co!sa(x)$ *est l'intervalle entier* $[1,100]$.

Théorème 5.6 (Correction de la tranchage par rapport aux contraintes)
Soient $SP = (S, C, P)$ une spécification, TP_L un OT abstrait et Σ_{f} un ensemble d'entrées contrôlables qui couvrent TP_L. Le produit synchrone entre les modèles de SP et $SP \setminus_{\mathrm{c}} \Sigma_{\mathrm{f}}$ avec l'OT et sous l'influence des entrées contrôlables Σ_{f} sont fortement bisimilaires :

$$\prod(\widehat{SP}, TP, \Sigma_{\mathrm{f}}) \sim \prod(\widehat{SP \setminus_{\mathrm{c}} \Sigma_{\mathrm{f}}}, TP, \Sigma_{\mathrm{f}}) \ .$$

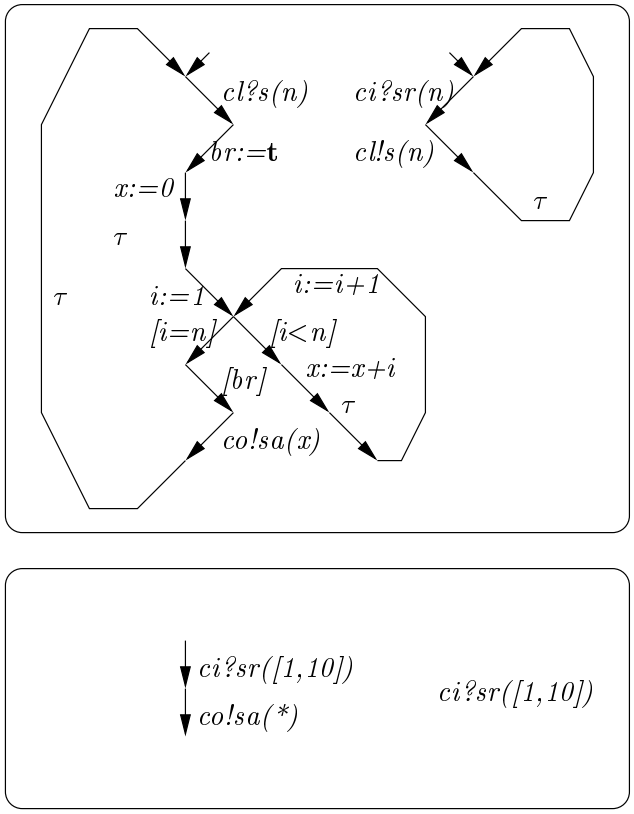

FIG. 5.3 – *Tranchage par rapport aux contraintes*

■

Preuve

On montre que l'égalité entre les états est aussi une bisimulation et donc c'est la bisimulation forte. La preuve est similaire à la preuve du théorème 4.1$_{[p.90]}$ et se trouve dans l'annexe $B_{[p.195]}$.

□

Chapitre 6

Mise en œuvre

6.1 L'environnement de validation IF

IF ([Boz99]) est un environnement de validation ouvert qui fournit une représentation intermédiaire pour les systèmes temporisés asynchrones. La description qui suit se réfère à la version *statique* de la représentation intermédiaire : elle ne prévoit pas la création/destruction des objets (ou processus légers)[1]. Une nouvelle version de la représentation intermédiaire IF ([BGM02]) ajoute ces caractéristiques et aussi améliore le fonctionnement de l'outil d'exploration, en permettant des systèmes dans lesquels on peut réunir des composantes décrites dans une technique de spécification formelle et du code exécutable. En outre, la structure des états de contrôle est similaire à celle de STATECHARTS ([Har87]), avec une structure hiérarchique, permettant de factoriser le comportement commun.

Cet environnement fournit aussi plusieurs *techniques de validation* (de simulation interactive à la vérification automatique des propriétés de logique temporelle, génération de cas de test, génération de code exécutable), plusieurs *niveaux de représentation des programmes* (permettant d'appliquer quelques techniques d'optimisation issues du domaine de la compilation et de calculer des abstractions).

L'environnement est aussi *ouvert* et *évolutif* et donc les connexions avec d'autres outils sont réalisées en partageant soit des formats d'entrées et de sorties soit des bibliothèques de composantes. Pour réaliser les connexions on fournit quelques interfaces de programmation APIs[2].

1. des caractéristiques largement employées dans UML ou JAVA
2. *application programming interface* en anglais

6.1.1 Architecture

L'architecture globale de l'environnement de validation IF et les connexions entre les outils composants sont décrites dans la figure 6.1[p.138].

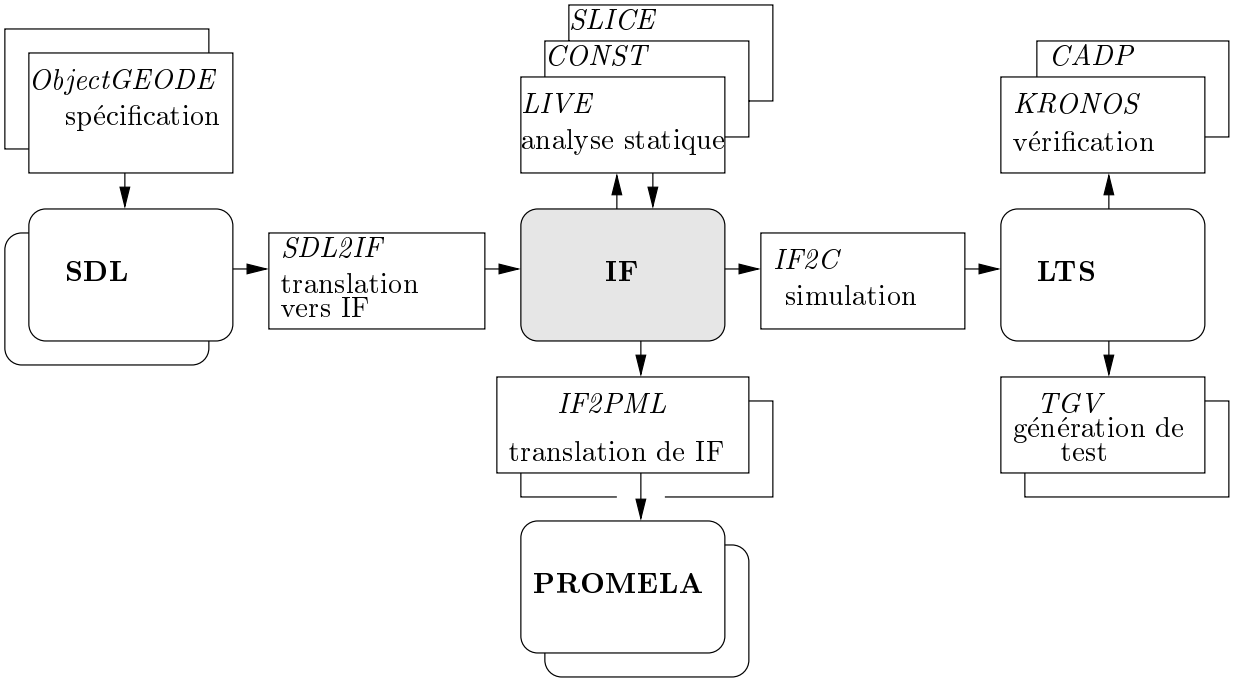

FIG. 6.1 – *L'environnement de validation* IF

IF s'appuie sur trois niveaux de représentation des programmes :

1. Le niveau de la **spécification** est la description initiale du programme (dans les langages existantes, ex. SDL [Sec94]). Cette description est traduite (automatiquement et avec les restrictions indiquées dans la section 6.2[p.144]) dans sa représentation IF. On considère seulement le formalisme d'entrée SDL mais les connexions avec autres langages (LOTOS [fS89], PROMELA [Hol91]) sont aussi possibles.

2. Le niveau **intermédiaire** correspond à la représentation IF [BFG+99c]. Dans IF un système est exprimé par un ensemble de processus parallèles qui communiquent de manière asynchrone via des canaux ou de manière synchrone via des portes de communication. Les processus sont des automates temporisés avec

échéances [BST97a], étendus avec des variables discrètes. Les transitions sont des commandes gardées, qui sont des entrées, sorties synchrones/asynchrones, affectations des variables et des horloges. Les canaux peuvent être des files, piles où sacs et ils peuvent être bornés ou non-bornés et avec pertes ou sans pertes.

L'interface API permet de consulter et de modifier l'arbre abstrait de la représentation IF. À ce niveau on peut aussi appliquer des optimisations (ex. celle de l'analyse des variables actives) ou calculer des abstractions. Ceci est possible à cause du fait que les variables, les horloges, les canaux et la structure de la communication sont encore explicites. L'interface sert aussi à implémenter des outils de traduction entre IF et autres formalismes de spécification.

3. Le niveau du *modèle* permet l'accès au STE qui représente le comportement du programme IF. En fonction de l'application considérée on propose trois types de API :

   (a) La *représentation implicite énumérative* consiste dans un ensemble de fonctions C et de structures de données permettant le calcul à la demande des successeurs d'un état quelconque (à la manière de OPEN-CAESAR [Gar98]). Les fonctions et les structures des données sont générées par le compilateur IF2C et le code obtenu peut être lié avec un programme *générique* d'exploration qui effectue une analyse *à la volée*.

   (b) Dans la *représentation symbolique* les ensembles d'états et de transitions du STE sont exprimés en utilisant leurs prédicats caractéristiques définis sur un ensemble fini de variables. Ces prédicats sont implémentés en utilisant des diagrammes de décision binaires (BDDs). Les outils qui utilisent cette API et qui sont implémentés sont l'outil de vérification symbolique ALDÉBARAN et l'outil de génération du modèle minimal.

   (c) La *représentation explicite énumérative* consiste dans un fichier qui contient un STE avec la bibliothèque d'accès correspondante. Quoique une telle représentation explicite n'est pas appropriée pour des grands systèmes, elle est utile en pratique pour minimiser les abstractions du modèle par rapport à des relations de bisimulation.

6.1.2 Description des composantes

On présente les principales composantes de l'environnement avec quelques outils externes pour lesquels il existe une connexion :

1. Les composantes du niveau de la spécification : il s'agit de la boîte à outils commerciale *Object*GEODE [Tel] de TTT qui supporte les formalismes SDL, MSC, OMT, UML. Cette boîte à outils offre l'accès à l'arbre abstrait d'une spécification SDL via une API. L'API a été utilisée pour développer l'outil de traduction SDL2IF [3] qui génère une spécification IF équivalente de point de vue opérationnel avec une spécification SDL donnée. L'outil ne traduit pas les aspects dynamiques de SDL (ex. la création des instances de processus).

   Les aspects liés à la traduction sont détaillés dans la section 6.2[p.144].

2. Les composantes du niveau intermédiaire : il s'agit de plusieurs outils d'analyse statique, utilisés pour transformer les spécifications IF :

   LIVE Cette transformation [4] ([BFG00a]) concerne la *réaffectation* des *variable mortes*. Elle s'applique aux variables mais aussi aux paramètres des messages. Elle s'est montrée efficace dans la génération du modèle (on obtient fréquemment des facteurs de réduction 100 en gardant le même comportement de la spécification initiale).

   SLICE, CONST Ces transformations [5] ([BFG00b]) permettent l'abstraction automatique d'une spécification donnée par l'élimination de certaines parties de la spécification en tenant compte d'un critère de tranchage calculé à partir d'une propriété à vérifier ou d'un objectif de test (on prend comme critère de tranchage les entrées contrôlables et les sorties observables).

3. L'outil IF2PML [BDHS00] est développé à Eindhoven TU pour la traduction des spécifications IF dans PROMELA.

4. Les composantes du niveau sémantique sont :

   CADP [Gar98] est une boîte à outils pour la vérification des spécifications LOTOS. CADP est développé par l'équipe VASY de INRIA Rhône-Alpes et par VERIMAG. L'environnement IF est connecté à deux outils de vérification de CADP :

3. environ 6 300 lignes de codes
4. 780 lignes de codes
5. 3 500 lignes de codes

ALDÉBARAN (basé sur les relations de bisimulation) et EVALUATOR (le μ-calcul alternant). Pour les deux outils les séquences de diagnostique sont calculées au niveau STE et peuvent être traduites dans MSC pour être observées au niveau de la spécification.

KRONOS [Yov97] est un outil de vérification symbolique des formules TCTL sur des automates temporisés communicants. Sa connexion avec IF est effectuée en exprimant les états de contrôle et les variables discrètes dans la représentation énumérative implicite et les horloges dans une représentation symbolique (des polyèdres).

TGV [FJJV96] est un générateur de séquences de test de conformité pour des systèmes distribués (développé conjointement par VERIMAG et le projet PAMPA de IRISA. Les cas de test sont calculés pendant l'exploration du modèle et ils sont sélectionnés en utilisant des objectifs de test.

6.1.3 La représentation intermédiaire

Dans IF un système est un ensemble de processus (des machines à états comme les processus SDL) qui communiquent de manière asynchrone via des files d'attente (qui peuvent être bornés ou non-bornés, avec pertes ou sans pertes). Chaque processus peut envoyer (respectivement recevoir) des messages vers (respectivement de) toute file. Le comportement temporisé du système peut être contrôlé par des horloges (comme les automates temporisés [ACD93, HNSY94]) et des variables temporisées (des *temporisateurs* SDL), qui peuvent être affectées à une constante, réinitialisées et qui expirent quand elles atteignent la valeur 0.

Définition 6.1 (Système IF)
Un système IF est le tuple SYS = $(globDef, \text{PROCS})$ où :

1. $globDef = (typeDef, sigDef, varDef, bufDef)$ est une liste de définitions globales, où $typeDef$ est une liste de définitions de types, $sigDef$ définit une liste de signaux paramétrés (comme dans SDL), $varDef$ est une liste de définitions de variables globales et $bufDef$ est une liste de files utilisées par les processus pour communiquer de manière asynchrone par échange de signaux.

2. PROCS définit un ensemble de processus (décrit ci-dessous).

▪

Définition 6.2 (Processus IF)
Un processus $P \in$ PROCS est le tuple $(varDef, Q, \text{TRANS})$, où :

1. *varDef* est une liste de définitions des variables locales (on inclut ici les horloges et les temporisateurs),
2. Q est l'ensemble d'états de contrôle pour lesquels on définit les attributs suivants :
   (a) $stable(q)$ et $init(q)$ sont des attributs booléens, où seulement les états *stables*[6] sont visibles au niveau sémantique,
   (b) pour les états stables, l'attribut $tpc(q)$ est un prédicat défini pour toute variable *visible* dans le processus P (celles globales et celles locales dans P) et qui définit la condition de la progression de temps[7].
   (c) les attributs $save(q)$ et $discard(q)$ sont des listes de filtres de la forme
   `signal-list` [**in** `buf`] [**if** `cond`].
   $save(q)$ est utilisé pour implémenter les instructions **save** de SDL. Son effet est de garder tous les signaux de la liste `signal-list` dans le canal `buf` quand la condition `cond` est vraie.
   $discard(q)$ est utilisé pour implémenter l'annulation implicite dans SDL des signaux qui ne peuvent pas être consommés dans l'état q. À la lecture du signal d'entrée suivant de `buf` tous les signaux qui le précèdent dans `buf` et qui doivent être annulés sont annulés dans la même transition atomique.
3. TRANS est un ensemble de transitions de contrôle qui consiste en deux types de transitions entre deux états de contrôle $q, q' \in Q$:
   (a) les transitions d'entrée dont l'exécution devient possible grâce à la lecture d'un signal de l'une des files d'attente (comme c'est le cas dans SDL) :
   $$q \xrightarrow[(\mathtt{u})]{\mathtt{g} \mapsto \mathtt{input;body}} q' \text{ et}$$
   (b) les transitions internes qui ne dépendent pas des communications :
   $$q \xrightarrow[(\mathtt{u})]{\mathtt{g} \mapsto \mathtt{body}} q'.$$

6. ceux dont $stable(q)$ est vrai et qui correspondent aux états SDL, à la différence d'états *non-stables* qui ont été introduits dans la traduction de SDL vers IF en vue de la syntaxe d'IF qui ne permet pas de garde au milieu d'une transition
7. Dans les états non-stables le temps ne peut pas avancer

Dans les deux cas :

(a) `g` est un prédicat représentant la *garde* de la transition et qui peut dépendre des variables visibles dans le processus (ci-inclus les horloges, les temporisateurs, et les files (pour lesquelles un ensemble de primitives fournit l'accès))

(b) `body` est une séquence d'actions atomiques de types suivants :

i. des *sorties* de la forme : ⟨ **output** *sig*(*par_list*) **to** *buf* ⟩ qui ont l'effet d'ajouter un *signal* de la forme *sig*(*par_list*) à la file *buf*.

ii. des affectations usuelles.

iii. des affectations de temporisateurs : ⟨ **set** *timer* := *exp* ⟩. Cette affectation arme le temporisateur *timer* et lui affecte la valeur de *exp*. La valeur d'un temporisateur actif diminue avec la progression du temps. À la différence des temporisateurs SDL dont on peut connaître seulement le moment de leurs expiration (quand ils atteignent la valeur 0), les valeurs des temporisateurs IF peuvent être connues à tout moment.
Les horloges sont toujours actives et leurs valeurs augmentent avec la progression du temps.

iv. des réinitialisations des temporisateurs et des horloges, qui ont les mêmes effets que ceux obtenus en inactivant un temporisateur ou l'affectation de la valeur 0 aux horloges.

(c) L'attribut `u` ∈ {**eager**, **delayable**, **lazy**} définit le type d'urgence pour chaque transition. Les transitions **eager** ont une priorité absolue à l'égard de la progression du temps, les transitions **delayable** peuvent laisser le temps s'écouler mais seulement si dans le moment suivant elles restent actives. Les transitions **lazy** ne peuvent pas empêcher l'avancement du temps. Ces types d'urgence ont été introduits dans [BST97b].

(d) `input` est une entrée de la forme ⟨ **input** *sig*(*reference_list*) **from** *buf* [**if** *cond*] ⟩, où : *sig* est un signal, *reference_list* est une liste de références (des variables qui peuvent être affectées) dans laquelle on stocke les paramètres reçus, *buf* est le nom de la file dont le signal devrait être lu et *cond* définit une condition pour que le signal reçu soit accepté; *cond* peut dépendre aussi des paramètres reçus.

▪

6.2 Traduction de SDL vers IF

Dans cette section on décrit la traduction d'une spécification SDL dans une spécification IF. On s'est intéressé dans ce type de traduction afin de permettre l'accès aux spécifications SDL des outils académiques de vérification (connectés déjà au compilateur IF) : KRONOS ([Yov97]), SPIN ([Hol91]), INVEST ([BL99]) ou la boîte à outils CADP ([Gar98]).

L'outil SDL2IF[8] qui implémente la traduction fait partie de l'environnement de validation IF (voir figure 6.1[p.138]). On a utilisé l'interface SDL de TTT ([Tel]) et la syntaxe SDL décrite dans [OFMP+94] (p.431). Seulement un sous-ensemble de SDL est traduit vers IF : le comportement[9], la structure et les données.

Parmi les restrictions notables de la traduction on mentionne :

1. l'aspect hiérarchique de la structure SDL n'est pas traduit directement dans IF,
2. id. pour les communications,
3. certains types SDL ne peuvent pas se traduire directement dans des types IF,
4. certaines expressions (par exemple les appels de procédure),
5. la création dynamique de processus.

6.2.1 La structure

SDL offre la possibilité de structurer hiérarchiquement les programmes. On utilise dans ce but les *blocs*, les *sous-structures*, les *processus*, les *services*, etc. Les systèmes IF sont *mis à plat* : un système IF a un seul niveau de processus qui communiquent directement via des files d'attente. La traduction de SDL vers IF prend en compte cet aspect en effectuant un aplatissage[10] du système structuré SDL.

Le mécanisme de communication de SDL est aussi structuré et dans ce but on peut utiliser diverses constructions : des *canaux*, des *routes de signaux*, des *points de connexion*, etc. Ce mécanisme est transformé en mécanisme de communication point-à-point via des files d'attente en calculant statiquement, pour chaque sortie, un seul processus recevant (et sa file correspondante).

Tous les types SDL prédéfinis (*boolean*, *integer*, *natural*, *real*, *pid*, *character*, *duration*, *time*), les types *array*, *record*, *range* et ceux énumérés peuvent être traduits par des types

8. 6 100 lignes de code
9. *behaviour* en anglais
10. *flattening* en anglais

IF. Quand la traduction n'est pas possible, par exemple dans le cas des types abstraits, on traduit ce type par un type *abstrait* de IF. On traduit seulement la signature et pour la simulation l'utilisateur doit fournir une implémentation adéquate. Dans la table 6.1[p.145] on fournit un exemple concernant la traduction des types.

Syntaxe SDL	Syntaxe IF
SYNONYM SEQ256 INTEGER = 3; SYNONYM SEQ1000 INTEGER = 32;	On remplace partout SEQ256 avec 3 On remplace partout SEQ1000 avec 32
SYNTYPE NbEltMaxType = NATURAL CONSTANTS 0:SEQ256 ENDSYNTYPE SeqMod256Type;	nbeltmaxtype = range 0 .. 3;
SYNTYPE SeqModType = NATURAL CONSTANTS 0:SEQ1000 ENDSYNTYPE SeqModType;	seqmodtype = range 0 .. 32;
NEWTYPE SourceType LITERALS SSCOP,USER; ENDNEWTYPE SourceType;	enum sscop,user;
NEWTYPE ListType ARRAY(NbEltMaxType,SeqModType); ENDNEWTYPE ListType;	listtype = array [0 .. 3] of seqmodtype;
NEWTYPE LossListType STRUCT liste ListType nbelt NbEltMaxType; currentcursor NbEltMaxType; ENDNEWTYPE LossListType;	losslisttype = record liste : listtype; nbelt : nbeltmaxtype; currentcursor : nbeltmaxtype; end;

TAB. 6.1 – *Traduction des types*

Dans SDL tous les signaux sont paramétrés implicitement avec le *pid* (identificateur de processus) du processus émetteur et par conséquence dans IF tous les signaux ont un paramètre supplémentaire de type **pid**.

6.2.2 Les processus

Pour chaque instance de processus SDL on génère un processus IF équivalent et on lui associe une file d'entrée implicite. Une file supplémentaire est introduite pour simuler l'environnement.

Si le nombre d'instances d'un processus SDL varie dans un certain intervalle, on génère le nombre maximum d'instances.

Les variables

Les variables ou les temporisateurs SDL locaux deviennent des variables ou des temporisateurs IF locaux. On défini aussi les variables locales **sender**, **offspring** et **parent** qui, dans SDL, sont des expressions définies implicitement.

Les variables *exportées* ou *importées* seront définies dans IF au niveau du système.

La plupart des expressions SDL peuvent se traduire dans IF. Les exceptions (ex. les appels de procédures *call*) sont traduites dans la même forme mais elles sont commentées.

Les états

Tous les états SDL (*start* et *stop* ci-inclus) sont traduits dans des états IF *stables*. À cause du fait que les transitions IF ont une structure plus simple que les transitions SDL on introduit des états IF *non-stables*[11] pour chaque *decision* ou *label* de SDL qui apparaît dans une transition SDL. Les ensembles *save* et *discard* d'un état IF stable sont les mêmes avec ceux de l'état SDL correspondant.

La traduction des états étiquetés par $*$ se fait par un aplatissage[12] : on duplique la transition correspondante pour tous les états qui correspondent au symbole $*$.

Les transitions

On génère une transition IF pour chaque chemin minimal (SDL) entre deux états IF. Elle contient les déclencheurs[13] et les actions définies dans ce chemin, dans le même ordre.

Toutes les transitions générées ont implicitement l'attribut **eager** (elles ont une priorité plus grande que le progrès du temps) et ainsi la traduction préserve la notion du progrès du temps de l'outil *Object*GEODE.

entrées: Les signaux d'entrée SDL sont traduits directement dans les entrées IF avec le paramètre *sender* apparaissant explicitement (dans SDL il apparaît implicitement et dans ce cas on retrouve l'identité du processus émetteur dans la variable `sender`).

11. dont les ensembles *save* et *discard* sont vides et un tel état est la source d'une seule transition sortante qui corresponde à une branche de la transition SDL

12. *flattening* en anglais

13. *triggers* en anglais

Les entrées spontanées ⟨`input none`⟩ sont traduites par l'affectation à la variable **sender** de l'identificateur *pid* du processus où se trouve ce déclencheur. Dans ce cas la transition ne contient pas la partie `input`.

expirations de temporisateurs: Dans IF elles ne sont pas notifiées par des signaux d'expiration : chaque consommation d'un signal d'expiration dans un processus SDL est traduite dans une transition sans la partie `input` qui teste si la valeur du temporisateur correspondant est *zero* et ensuite effectue une réinitialisation du temporisateur. La réinitialisation est nécessaire pour empêcher la consommation multiple du signal qui marque l'expiration du temporisateur.

entrées prioritaires: Elles sont traduites dans des entrées normales en renforçant la garde de toutes les entrées moins prioritaires et aussi l'ensemble *save* de l'état source. La garde de chaque entrée moins prioritaire est une conjonction entre la garde ancienne et un terme avec la signification : *il n'y a pas de signal d'entrée plus prioritaire dans le canal d'entrée*. Tous les signaux moins prioritaires sont sauvés explicitement si la condition : *au moins une entrée avec une priorité plus grande existe dans le canal d'entrée* est vraie. Ces tests peuvent être exprimés de manière effective en utilisant des prédicats prédéfinis sur les canaux de communication.

signal continu: Dans le cas des transitions SDL déclenchées par ce type d'entrée on génère des transitions IF sans la partie `input` mais avec une garde équivalente au signal continu.

condition permettante: Une condition permettante qui suit à une entrée SDL est traduite dans une entrée avec une garde après, où les paramètres reçus peuvent être testés.

tâche: Ces constructions SDL sont traduites dans des affectations IF. Les tâches informelles SDL deviennent des commentaires dans IF.

(ré)initialisations: L'initialisation d'un temporisateur SDL devient une initialisation de temporisateur IF. Dans ce cas la valeur absolue $\langle now + T \rangle$ devient (en IF) la valeur relative $\langle T \rangle$. Les réinitialisations des temporisateurs SDL devient des réinitialisations des temporisateurs IF.

sorties: Les sorties SDL deviennent des sorties IF dans les conditions suivantes :

- Si on peut identifier directement le destinataire à cause du fait que la clause *pid-expression* est présente dans la sortie alors on prend *pid-expression* comme destination du signal IF.

– Sinon on tient compte de la clause *via* et des chemins de communication SDL pour calculer statiquement l'ensemble de toutes les destinations possibles. Si cet ensemble contient exactement une seule instance de processus elle sera la destination de la sortie IF, sinon la sortie n'est pas traduite et l'utilisateur est averti.

Chaque sortie traduite a comme premier paramètre le `pid` du processus émetteur.

décision: Chaque alternative d'une décision formelle SDL est traduite dans une garde qui est placée au début d'une transition IF avec comme état source l'état non-stable correspondant.

création dynamique: Elle n'est pas traduite.

procédures: On insère directement le graphe de la procédure au lieu de son appel dans le graphe de processus.

Dans les tables 6.2[p.149] et 6.3[p.150] on fournit deux exemples concernant la manière de traduire les transitions.

6.3 Bibliothèque d'analyse statique

La bibliothèque implémente un moteur de calcul de solutions pour des problèmes d'analyse statique exprimés par équations de flux de données. L'architecture de la bibliothèque assure un développement à la fois rapide mais aussi fiable des outils de calcul des solutions. Une hiérarchie de classes C++ modélise (de manière similaire à [DC96]) les composantes de calcul : l'information, les fonctions de transfert, le treillis. Les algorithmes implémentés dans la bibliothèque sont des adaptations des algorithmes appliqués auparavant dans les optimisations pour compilateurs, pour les programmes séquentiels ([Muc97]).

Les principales composantes de la bibliothèque sont :

1. *le noyau*
   Il comprend les structures de données génériques (qui modélisent la distribution de l'information dans chaque état, le transfert de l'information) qui seront spécialisées pour chaque outil développé, les structures de données qui modélisent les éléments des treillis et les algorithmes qui, en utilisant ces structures, calculent itérativement les solutions des équations de flux de données. Il compte environ 3 300 lignes de code.

Syntaxe SDL	Syntaxe IF
STATE Idle; INPUT AaEstablishRequest(BR) CALL ResetTransmitter(tbuffer, tqueue, VTS, VTA, VTPS, VTPA, VTPD); CALL clearReceiverBuffer(rbuffer); TASK rtqueue!cursor:=0; TASK VTCC:=1; CALL Increment256(VTSQ); CALL Initialize(VRMR, VRR, 20); TASK BgnPdu!NSQ:=VTSQ; TASK BgnPdu!NMR:=VRMR; TASK LastBgnPdu:=BgnPDU; OUTPUT BgnInvoke(BgnPDU); SET (now+100, TimerCC); NEXTSTATE OutgoingConnectionPending;	from idle :eager input aaestablishrequest(br) from sscop do /* call resettransmitter(tbuffer, tqueue, vts, vta, vtps, vtpa, vtpd) */ /* call clearreceiverbuffer(rbuffer) */ rtqueue.cursor := 0, vtcc := 1, /* call increment256(vtsq) */ /* call initialize(vrmr, vrr, 20) */ bgnpdu.nsq := vtsq, bgnpdu.nmr := vrmr, lastbgnpdu := bgnpdu, output bgninvoke(bgnpdu) to EnvBuffer, set timercc := (0 + 100.000000) /* nextstate */ to outgoingconnectionpending;

TAB. 6.2 – *Traduction des transitions*

Syntaxe SDL	Syntaxe IF
STATE OutgoingResynchronizationPending;	from outgoingresynchronizationpending
	:eager
INPUT TimerCC;	if (timercc = 0)
	to q109;
DECISION VTCC >= MaxCC;	
(TRUE):	from q109 :eager
	if (vtcc >= 4) = true
	do
OUTPUT MAaErrorIndication(OO, 0);	output maaerrorindication(oo, 0)
	to EnvBuffer,
TASK EndPdu!source:=SSCOP;	endpdu.source := sscop,
OUTPUT ENDInvoke(EndPdu);	output endinvoke(endpdu)
	to EnvBuffer,
TASK LastEndPdu:=EndPdu;	lastendpdu := endpdu,
OUTPUT AaReleaseIndication(SSCOP);	output aareleaseindication(sscop)
	to EnvBuffer
NEXTSTATE Idle;	/* nextstate */
	to idle;
(FALSE):	from q109 :eager
	if (vtcc >= 4) = false
	do
TASK VTCC:=VTCC + 1;	vtcc := (vtcc + 1),
OUTPUT RSInvoke(LastRsPdu);	output rsinvoke(lastrspdu)
	to EnvBuffer,
SET (now +100, TimerCC);	set timercc := (0 + 100.000000)
NEXTSTATE	/* nextstate */
OutgoingResynchronizationPending;	to outgoingresynchronizationpending;
ENDDECISION;	

TAB. 6.3 – *Traduction des transitions*

2. *le calcul d'activité*
   Il s'agit du calcul, au moment de la compilation, de l'information suivante : dans un état de contrôle de la spécification, quelles sont les variables *actives* (celles dont la valeur pourrait être utile à partir de cet état). Le résultat du calcul servira au calcul des variables *inactives* qui seront remises à une valeur constante dans les transitions qui entrent dans cet état. Il compte environ 780 lignes de code.

3. *propagation de constantes*
   Il s'agit du calcul, au moment de la compilation, de l'information suivante : quelle est la valeur d'une variable dans un état de contrôle au moment de l'exécution ? Cette information pourrait être utilisée pour éliminer certaines transitions (dont on peut vérifier par cette analyse d'invariance qu'elles ne seront jamais franchissable à l'exécution). L'outil CONST qui implémente ce calcul compte environ 2 500 lignes de code.

   Un autre outil qui utilise cette technique est l'outil CLOCKREDCT, qui implémente le recouvrement des variables temporisées IF. Celles-ci sont d'abord remplacées par des horloges conformément aux règles de la table 6.4[p.151].

SDL	Temporisateurs IF	Horloges IF
$set(v, exp)$	$set\ v\ :=\ exp$	$set\ v\ :=\ -exp$
$reset(v)$	$set\ v\ :=\ -1$	$set\ v\ :=\ 1$
$expire(v)$	$v\ ==\ 0$	$v\ ==\ 0$

TAB. 6.4 – *Remplacement*

   On introduit ensuite C_n^2 variables qui représentent la différence entre chaque paire d'horloges. Ensuite on applique une propagation de constantes qui nous servira à déduire les règles de recouvrement.

   L'outil compte environ 1 000 lignes de codes.

4. *le calcul d'utilité*
   Ce calcul est une adaptation pour les systèmes asynchrones (avec des processus communicants par files d'attente) de la technique classique de tranchage utilisée dans le débougage des programmes ([Wei84, Tip94]). On calcule les parties utiles de la spécification en rapport avec certains signaux observables/contrôlables et leurs paramètres. L'outil qui implémente cette technique compte environ 2 900 lignes de code.

5. *la propagation des copies*
   C'est une technique aussi bien connue et utilisée dans les compilateurs ([Muc97]). Propagation des copies est utile pour simplifier les cas de test symboliques ([BJR00]). Dans ces cas de test on trouve souvent des séquences d'actions comme la suivante : $x := y; z := x$. Cette séquence pourrait être réduite à : $z := y$. L'outil qui implémente cette technique compte environ 700 lignes de code.

Les classes principales de la bibliothèque sont présentées dans la figure 6.2[p.152].

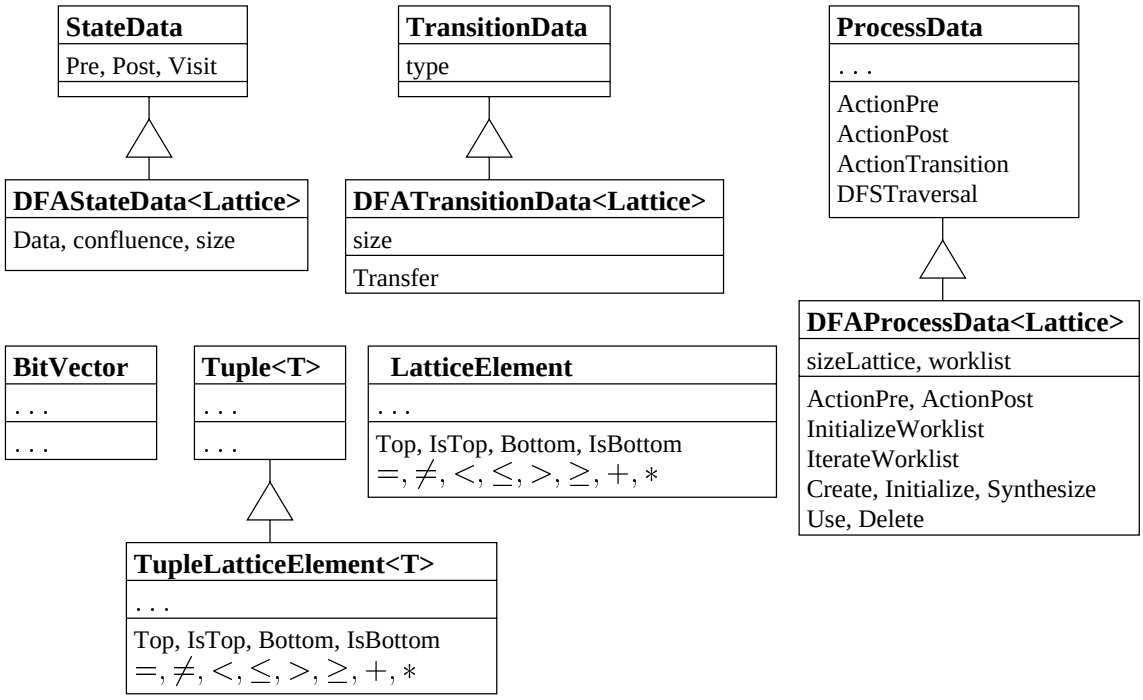

FIG. 6.2 – *Les classes de la bibliothèque*

La classe *BitVector* implémente l'ensemble des parties d'un ensemble V. La classe paramétrée *Tuple* sert à définir des objets qui sont les éléments d'un produit cartésien. La classe dérivée *TupleLatticeElement* sert à implémenter le produit cartésien de treillis.

La classe *LatticeElement* fournit une interface unique pour tous les éléments des treillis. Ces classes servent comme des paramètres pour les classes *DFAStateData*, *DFATransitionData*, *DFAProcessData*.

Les objets de type *StateData*, *TransitionData* et *ProcessData* sont des attributs calculés des nœuds de l'arbre abstrait d'une spécification IF.

On spécialise ces classes en fonction des analyses souhaitées: ex. pour l'analyse des variables actives, les classes correspondantes sont : *LiveStateData*, *LiveTransitionData* et *LiveProcessData*. La classe *LiveStateData* est dérivée de la classe *DFAStateData*$\langle BVLatticeElement\rangle$. Les classes *LiveTransitionData* et *LiveProcessData* sont dérivées de manière similaire des classes *DFATransitionData*$\langle BVLatticeElement\rangle$ et *DFAProcessData*$\langle BVLatticeElement\rangle$.

Au niveau de chaque processus (dans les classes dérivées de *ProcessStateData*) on implémente les algorithmes de parcours de graphes, de l'entretien de la liste de travail, des itérations, etc.

6.4 Applications

On utilise comme études de cas deux spécifications de protocoles: un protocole mono-processus (SSCOP) et un protocole multi-processus (MASCARA). Pour le séquentiel de vol d'ARIANE-5 on présente les résultats obtenus en appliquant le recouvrement des horloges.

6.4.1 Le protocole SSCOP

Le protocole SSCOP [14] fait partie des protocoles de la pile ATM [15]. SSCOP est normalisé sous la référence ITU-T Q2110 ([dT94]). Il est conçu pour transférer d'une façon sûre des données à haut débit entre deux entités large bande. SSCOP est une des sous-couches de la couche AAL [16] (figure 6.3$_{[p.154]}$) qui a pour rôle essentiel d'adapter le service fourni par la couche ATM physique au type des données passant par la connexion établie entre deux extrémités.

Cette étude de cas a été fournie par le CNET [17] Lannion dans le cadre de l'action FORMA [18]. Sa spécification SDL compte environ 9000 lignes de code et comporte un seul processus avec 10 états et 210 transitions.

SSCOP fournit à la couche utilisatrice les services suivants :

- Intégrité de séquencement : les SDUs [19] soumises par l'utilisateur au protocole SSCOP sont numérotées dans l'ordre où elles seront soumises pour transfert,

14. Service Specific Connection Oriented Protocol
15. Asynchronous Transfer Mode
16. ATM adaptation layer
17. Centre National d'Études de Télécommunications de France Telecom
18. action soutenue par le programme DSP STTC/CNRS/MENRT : « Maîtrise de systèmes complexes réactifs et sûrs»
19. *service data unit* en anglais

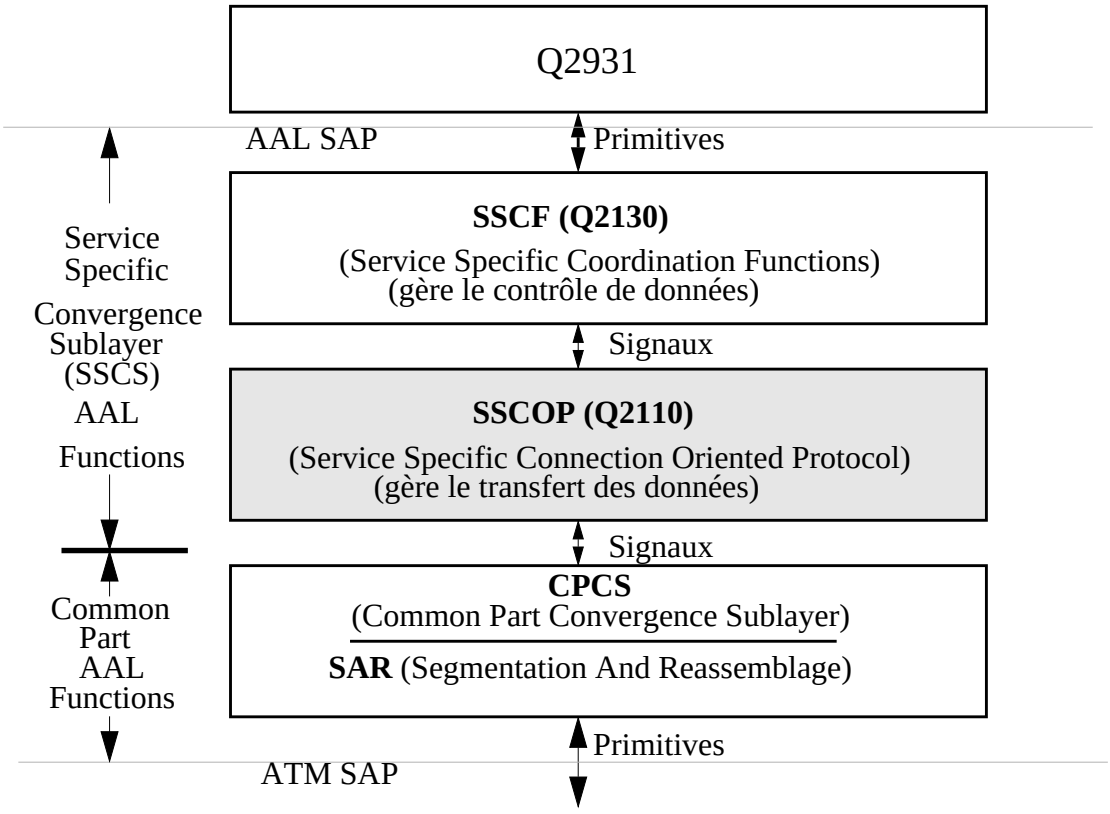

FIG. 6.3 – *Situation de* SSCOP *dans la pile de protocoles ATM*

- Correction d'erreur par retransmission sélective : détection par le récepteur de SDU manquantes et demande de retransmission,
- Contrôle de flux par commande du débit par le récepteur,
- Indication d'erreur à la couche de gestion,
- Maintien de la liaison entre deux entités SSCOP homologues en l'absence prolongée de transfert de données,
- Recherche de données locales, si nécessaire, parmi les SDU non encore libérées,
- Contrôle de la connexion : établissement, maintien, relâchement et résynchronisation,
- Transfert de données usager en mode garanti ou non,
- Détection et recouvrement d'erreur de protocole,
- Indication d'état (de la fenêtre de transfert) entre les deux extrémités associées.

L'analyse des variables actives a été utilisée dans le travail de validation de deux entités SSCOP communicantes ([BFG+00d]), reliées entre elles par des files d'attentes bornées (modélisant ainsi une couche inférieure fiable, sans perte de signaux), et susceptibles d'accepter différents signaux de la couche supérieure. En outre, dans les expérimentations effectuées on a utilisé les outils *Object*GEODE ([Tel]) et ALDÉBARAN ([Fer88, BFKM98]). La réinitialisation des variables inactives a permis d'obtenir des gains importants (on a obtenu des réductions d'environ 95% dans la taille des graphes obtenus).

Quoique les résultats précédents n'ont pas été obtenus en utilisant directement l'environnement de validation IF, on peut estimer que les résultats obtenus en appliquant les analyses des variables actives et des variables utiles sont de même magnitude avec celui mentionné ci-dessus.

On a effectué aussi des expérimentations avec les techniques de propagation des constantes et des intervalles. Comme indicateur d'efficacité on a utilisé le rapport entre le nombre de transitions évaluées (avec les invariants obtenus) à une valeur booléenne constante (vrai ou faux) et le nombre total de transitions. Les résultats obtenus n'ont pas l'efficacité de ceux obtenus par les analyses d'activité et d'utilité (environ 6% des gardes ont pu être évaluées à une valeur constante) mais ils montrent bien la nécessité d'utiliser des techniques de génération d'invariants plus puissantes.

Les résultats fournis par l'analyse du contrôle dépendent des critères choisis, pour 5 critères de tranchage on a obtenu une moyenne de 96% d'états et 98% de transitions éliminés (dans la spécification initiale).

En ce qui concerne l'analyse d'utilité on considère comme indicateurs le nombre de fois dont on a appliqué les règles $5.4_{[p.127]} \div 5.9_{[p.128]}$: pour 6 critères de tranchage on a obtenu en moyenne 1% applications de la règle $5.5_{[p.127]}$ (par rapport avec le nombre total d'application des règles $5.4_{[p.127]}$ et $5.5_{[p.127]}$) et 56% applications de la règle $5.7_{[p.128]}$ (par rapport avec le nombre total d'application des règles $5.6_{[p.127]}$ et $5.7_{[p.128]}$).

6.4.2 Le protocole MASCARA

Le protocole MASCARA[20] ([DPDea98]) est un protocole d'accès en liaisons de données[21] conçu pour les réseaux de communication locaux ATM sans fil[22] et développé par le consortium WAND[23] ([Con96]). C'est un protocole de taille industrielle (sa description compte environ 309 pages dans le format textuel SDL et les parties qui ont été vérifiées ont (dans IF) 9 processus, avec 537 états et 630 transitions et respectivement 9 processus, avec 463 états et 556 transitions).

Un réseau ATM sans fil est composé par un réseau ATM fixe qui connecte plusieurs APs (*point d'accès*) distribués de manière géographique et qui étendent de manière transparente les services ATM vers les MTs (*terminaux mobiles*).

Les tâches du protocole MASCARA sont :

1. la médiation entre les APs et les MTs qui connectent le réseau ATM fixe au réseau ATM radio et

2. le contrôle de l'accès au support de transmission radio.

Le protocole MASCARA est organisé sur plusieurs niveaux. On s'intéresse seulement au niveau de contrôle : MCL.

La tâche du niveau MCL est de s'assurer que les MTs peuvent initialement établir une connexion avec un AP et que cette connexion fonctionne avec une bonne qualité et avec un nombre minimal d'interruptions (pendant la durée dont elle est valide).

Ce niveau contrôle périodiquement la qualité de la connexion radio et il maintient une liste de APs voisins avec une bonne qualité de la connexion (en vue de remplacement

20. Mobile Access Scheme based on Contention and Reservation for ATM
21. *medium access control protocol* en anglais
22. *wireless* ATM en anglais
23. *Wireless* ATM *Network Demonstrator* en anglais

éventuel de AP dans le cas de la détérioration de la connexion courante). Le niveau MCL est différent en fonction de l'entité qui l'utilise (AP ou MT). Il est composé par quatre parties : DC (*le contrôle dynamique*), SS (*le contrôle de la stabilité d'état*[24]), RCL (le contrôle radio) et GMC (le contrôle MASCARA générique). On s'intéresse seulement à la partie DC.

La tâche principale de la partie DC est d'établir et de débloquer des *associations* et des connexions virtuelles. DC est composée par plusieurs entités (agents) :

1. *l'agent générique dynamique* est l'entité de plus haut niveau dans la DC et sa tâche est la *gestion de l'association*. Il expédie les requêtes des niveaux supérieurs concernant les associations et les connexions existantes vers l'agent d'association indiqué, il gère les adresses pour les associations et il informe le niveau supérieur des associations perdues.

2. *l'agent d'association* de MT et l'entité homologue dans le AP associé sont responsables pour la gestion et le contrôle d'une seule association. Chaque association peut comprendre un nombre variable de canaux virtuels. La tâche des entités homologues aux agents d'association est la création des canaux virtuels de données et du contrôle, d'établir une correspondance entre les adresses du niveau ATM et les identificateurs de connexion du niveau de l'agent MVC et de renvoyer les requêtes. On trouve un seul agent d'association dans MT et plusieurs (correspondant aux MTs) dans AP.

3. *l'agent* MVC[25] de MT et son entité homologue dans AP gère une seule connexion et s'occupe de l'allocation de ressources pour cette connexion.

Le protocole MASCARA a été vérifié dans le cadre du projet VIRES[26] ([Vir]). Les travaux de vérification se sont concentrées sur le niveau MCL : la partie DC ([JG01]) et la partie SS ([BDHS00]).

Les outils d'analyse statique présentés dans la section 6.3[p.148] ont été utilisés surtout dans les travaux de vérification entreprises au laboratoire VERIMAG ([JG01]) et les résultats obtenus sont présentés dans la table 6.5[p.158]. La propriété mentionnée se réfère à l'établissement d'une association entre les entités MP et AP.

On peut remarquer l'efficacité des techniques (surtout si on se rapporte à leurs faible coût en espace et temps calcul).

24. *steady state control* en anglais
25. *medium access control virtual channel agent* en anglais
26. Verifying Industrial REactive Systems

Entités	Optimisations	Taille graphes d'état			
		Transitions		États	
AP	génération directe du modèle	30.689.244	100%	7.308.400	100%
	LIVE	1.536.699	5%	351.202	4,80%
	SLICE & LIVE par rapport à une propriété	12.644	0,04%	4.556	0,06%
MT	génération directe du modèle	12.811.961	100%	4.388.765	100%
	LIVE	325.312	2,53%	63.628	1,44%
	SLICE & LIVE par rapport à une propriété	16.854	0,13%	4.018	0,09%

TAB. 6.5 – *Optimisations entreprises sur la partie* DC

6.4.3 Le séquentiel de vol ARIANE-5

Le séquentiel de vol est un logiciel embarqué qui contrôle le lanceur pendant son vol à partir de la plate-forme de lancement et jusqu'à l'orbite. Dans le cadre de la validation du protocole du séquentiel de vol ARIANE-5 ([BLM01]) on a utilisé une spécification qui contient environ 4 000 lignes de code SDL. Cette spécification décrit les tâches de *contrôle* et d'*observation* des blocs propulseurs (EAP[27], EPC[28], EPS[29]) de la fusée (il s'agit ici principalement de l'allumage et de l'extinction des ces blocs) et de *configuration de vol* (à la suite du largage des blocs au fur et à mesure que la fusée atteigne l'orbite).

Les temporisateurs interviennent dans la spécification dans la phase d'allumage : chaque action d'allumage doit s'exécuter à un moment précis et fixé au préalable (moment fixé par l'affectation d'un temporisateur et dont l'expiration indique quand l'action doit s'exécuter). Dans la phase de largage on utilise des temporisateurs pour indiquer une fenêtre d'opportunité pour l'exécution d'une action.

L'utilisation de l'outil qui implémente la technique de recouvrement des variables temporisées a conduit à une diminution importante du nombre des horloges[30] (environ 63 % de variables ont été recouverts).

27. Étage accélérateur à poudre
28. Étage principal cryotechnique
29. Étage à propergol stockable
30. On a traduit les temporisateurs SDL par des horloges IF

Chapitre 7

Conclusions

7.1 Bilan

Ce travail se situe dans le cadre de la vérification et de la validation des systèmes répartis, particulièrement les protocoles de télécommunication. Nous nous sommes intéressés au problème de la génération automatique de tests de conformité pour les protocoles de télécommunication et plus précisément à l'amélioration d'un outil de génération de séquences de test (TGV) dans la direction d'un traitement efficace des données (traitées en énumération) et pour concevoir un langage d'expression pour les objectifs de test.

Traitement efficace des données

L'outil TGV ([FJJV97, FJJV96]) utilise des objectifs de test décrits au niveau du modèle de la spécification : les paramètres des signaux sont traités en énumération et le produit synchrone (l'opération de composition nécessaire pour l'obtention d'un cas de test) se fait entre ce type des OTs et le modèle de la spécification. Or, le nombre d'états du modèle pourrait être exponentiel par rapport à la taille de la spécification et pour pallier ce problème il y a plusieurs techniques :

1. On construit à la volée le modèle de la spécification (la solution proposée dans [JM97]).

2. On utilise des OTs décrits au niveau de la spécification (AECs) et on les composent avec la spécification. Le cas de test obtenu est ensuite rendu déterministe (en utilisant quelques heuristiques). Ceci est la solution proposée dans [RBJ00].

Notre solution consiste dans l'utilisation des OTs décorés avec des contraintes imposées aux valeurs des paramètres de signaux. Un OT étendu est une abstraction pour une famille des OTs simples (utilisés par l'outil TGV).

En tenant compte des contraintes attachées aux OTs et en utilisant des techniques diverses d'analyse de programmes (analyse statique, interprétation abstraite et tranchage) on peut soit simplifier la spécification, soit raffiner les contraintes des paramètres de signaux de sortie de l'OT

Une première analyse proposée (analyse de contrôle) élimine les parties de la spécification qui n'interviennent pas dans son modèle (à cause du fait que dans la spécification ces états et transitions sont accessibles seulement via une transition étiquetée par une action d'entrée, entrée externe mais pas contrôlable).

Les autres analyses proposées peuvent être formulées comme des analyses de flot de données. Il s'agit ici des analyses d'activité et d'utilité des variables[1] et des propagations des intervalles et des constantes. Toutes ces analyses ont été formulées en utilisant une approche équationnel qui construit à partir du graphe de flot de contrôle de la spécification un opérateur de flot, dont la solution (soit le plus petit point fixe soit une approximation de celui-ci calculée en utilisant les techniques de l'interprétation abstraite) représente l'invariant recherché (les variables actives/utiles dans chaque état ou les valeurs des variables dans chaque état, valeurs estimées par des constantes ou par des intervalles entiers).

Ces invariants sont utilisés ensuite pour simplifier la spécification. En utilisant les variables actives/utiles dans chaque état on peut effectuer une abstraction de la spécification :

- soit en éliminant les variables qui ne sont actives dans aucun état et en affectant une valeur quelconque (mais unique) aux variables qui sont inactives dans un état, dans toutes les transitions entrantes dans cet état

- soit en éliminant les affectations aux variables inutiles dans un état dans toutes les transitions sortantes de cet état.

En utilisant les constantes et les intervalles entiers on peut supprimer le *code mort* de la spécification (les transitions dont la valeur des gardes, calculée en utilisant ces invariants, est fausse).

Dans chaque cas mentionné ci-dessus les modèles des nouvelles spécifications sont bisimilaires aux modèles des spécifications initiales.

1. Ces analyses sont classiques en compilation, sur les graphes de flot de contrôle. Ici, on les a étendues et formalisées pour des AECs par files d'attente

Les analyses proposées ont été implémentées dans une bibliothèque d'analyse statique qui fait partie de l'environnement de validation IF.

Langage d'expression pour les objectifs de test

Les objectifs de test étendus sont une extension naturelle des objectifs de test utilisés par l'outil TGV et décrits dans le format de l'outil ALDÉBARAN. Nous proposons leur description dans le langage IF.

7.2 Perspectives

À court terme on aimerait étendre la bibliothèque d'analyse statique avec des analyses plus précises que la propagation des constantes ou celle des intervalles entiers : il s'agit pratiquement d'y ajouter les treillis des polyèdres convexes[2] ([CH78, HPR97]) et celui des congruences linéaires[3] ([Gra90]).

Dans la suite on vise l'amélioration des analyses statiques inter-processus. Couramment, un problème d'analyse statique inter-processus est transformé dans un problème intra-processus en effectuant une approximation de l'AEC qui est obtenu par la composition asynchrone de tous les AECs de la spécification, par un AEC dans lequel toutes les communications asynchrones sont transformées dans des communications synchrones. On voudrait améliorer cette transformation en diminuant le nombre de communications synchrones obtenues à partir des communications asynchrones.

Ensuite on voudrait considérer des techniques plus sophistiquées, par exemple les abstractions ([LGS+94, DGG97]), les optimisations effectuées pour la parallélisation du code séquentiel (en éliminant certaines dépendances de données dans les nid-boucles [PW86, GKT91, CDRV96]), les optimisations par remplacement des boucles avec des meta-transitions ([Boi98, Ann01]) et les techniques de calcul des invariants non-linéaires ([BBF+00]), obtenus en combinant les dernières deux types de techniques.

On aimerait ensuite étendre les principes de la méthodologie proposés dans cette thèse aux protocoles infinis (avec ou sans paramètres).

2. pour calculer des invariants linéaires sur les variables du système

3. pour calculer des invariants utiles dans des problèmes, comme par exemple la vectorisation automatique ou l'analyse des dépendances des boucles [BBF+00]

Annexe A

Algorithmes

```
procedure SCC − Solver (P : PFD; var X : P.S.Q → P.L)
  type FILE = file de P.S.Q;
  var SCCs : pile de FILE;
  procedure
    IterateWorklist;
    InitializeWorklist;
  begin
    InitializeWorklist();
    IterateWorklist();
  end.
```

Continuation dans les tables A.2[p.164] *et A.4*[p.166]

TAB. A.1 – *Stratégie* SCC-STR - *I*

```
procedure InitializeWorklist
  var
    x : P.S.Q;
    LL, Dfn : P.S.Q → int;
    cnt : int;
    stk : pile de P.S.Q;
    InComponent : P.S.Q → bool;
  procedure Strong − CC(x : P.S.Q);
  begin
    SCCs := ∅;
    stk := ∅;
    cnt := 0;
    foreach (x ∈ P.S.Q)
      Dfn(x) := −1;
      LL(x) := −1;
      InComponent(x) := false;
    endfor
    foreach (x ∈ P.S.Q)
      if (Dfn(x) = −1)
        Strong − CC(x);
    endfor
  endproc InitializeWorklist;
```

Continuation dans la table A.3 [p.165]

TAB. A.2 – *Stratégie* SCC-STR - *II*

```
      procedure Strong − CC ( x : P.S.Q )
        var
          y : P.S.Q;
          a : P.S.Σ;
          scc : file de P.S.Q;
        begin
          cnt := cnt + 1;
          Dfn(x) := cnt;
          LL(x) := cnt;
          if (P.δ = ←)
(3)         stk.Push(x);
          foreach (a ∈ P.S.Σ ∧ y ∈ Post_a(x))
            if (Dfn(y) = −1)
              // l'état y n'est pas visité
              Strong − CC(y);
            if (not InComponent(y))
              LL(x) := min(LL(x), LL(y));
          endfor
          if (P.δ = →)
(4)         stk.Push(x);
          if (LL(x) = Dfn(x))
            scc := ∅;
            while (stk ≠ ∅)
              y := stk.Top();
              if (Dfn(y) < Dfn(x))
                break;
(5)           y := stk.Pop();
              InComponent(y) := true;
(6)           scc.Push(y);
            endwhile
            SCCs.Push(scc);
          endif
        endproc Strong − CC;
```

TAB. A.3 – *Stratégie* SCC-STR - *III*

```
      procedure IterateWorklist
        var
          scc : file de P.S.Q;
        procedure IterateOneWorklist(var wl : FILE)
          var
            ℓ : P.L;
            q, q' : P.S.Q;
            C : ensemble de P.S.Q;
          begin
            C := wl;
            X := P.X_0;
            while (wl ≠ ∅)
              q := wl.Pop();
              ℓ := P.E(X)(q);
(6)           if (ℓ ≠ X(q))
                X(q) := ℓ;
                if (P.δ = →)
(7)               wl.Push(P.S.Post(q) ∩ C);
                else
(8)               wl.Push(P.S.Pre(q) ∩ C);
                endif
              endif
            endwhile
          endproc IterateOneWorklist ;
        begin
          if (P.δ = ←)
            SCCs.Reverse();
          while (SCCs ≠ ∅)
            scc := SCCs.Pop();
            IterateOneWorklist(scc);
          endwhile
        endproc IterateWorklist ;
```

TAB. A.4 – *Stratégie* SCC-STR - *IV*

```
      procedure Iterative − Solver (P : PFD; var X : P.S.Q → P.L)
        type
          FILE = file de P.S.Q;
        var
          wl : FILE;
        procedure InitializeWorklist(var wl : FILE)
          var
            Stk : pile de P.S.Q;    Vis : ensemble de P.S.Q;
            q, x : P.S.Q;
          begin
            Stk := ∅;    Vis := ∅;
            Stk.Push(P.S.q0);
            while (Stk ≠ ∅)
              q := Stk.Pop();
              Vis := Vis ∪ {q};
(1)           if (P.δ = ←)
(2)             wl.Push(q);
              foreach (x ∈ Post(q))
                if (x ∉ Vis ∪ Stk)
                  Stk.Push(x);
              endfor
(3)           if (P.δ = →)
(4)             wl.Push(q);
            endwhile
(5)         wl.Reverse();
          endproc InitializeWorklist;
        procedure IterateWorklist(var wl : FILE)
          var
            ℓ : P.L;
            q, q' : P.S.Q;
          begin
            X := P.X0;
            while (wl ≠ ∅)
              q := wl.Pop();
              ℓ := P.E(X)(q);
(6)           if (ℓ ≠ X(q))
                X(q) := ℓ;
                if (P.δ = →)
(7)               wl.Push(P.S.Post(q));
                else
(8)               wl.Push(P.S.Pre(q));
                endif
              endif
            endwhile
          endproc IterateWorklist;
        begin
          InitializeWorklist(wl);
          IterateWorklist(wl);
        end.
```

TAB. A.5 – *Stratégie* WLIST-STR

```
procedure SCSC − Solver (P : PFD; var X : P.S.Q → P.L)
  type
    FILE = file de P.S.Q;
    PILE = pile de P.S.Q;
  var
    x : P.S.Q;
    cnt : int;
    LL, Dfn : P.S.Q → int;
    stk : pile de P.S.Q;
    SCCs : pile de FILE;
    Elarg : P.S.Q → bool;
  procedure
    Initialize;
    Strong − SCC(x : P.S.Q, truecall : bool);
    Split (x : P.S.Q; scc : FILE);
    Iterate;
    Widening (var wl : FILE);
    Narrowing;
  begin
    Initialize();
    Iterate();
  end.
```

Continuation dans les tables A.7[p.169], A.8[p.170], A.9[p.170] A.10[p.171], A.11[p.172], A.12[p.172]

TAB. A.6 – *Stratégie* SCSC-STR - *I*

```
*     procedure Strong − SCC ( x : P.S.Q, truecall : bool )
        var
          y : P.S.Q;
          a : P.S.Σ;
*         loop : bool;
          scc : FILE;
        begin
          cnt := cnt + 1;
          LL(x) := cnt;
          Dfn(x) := cnt;
*         loop := false;
          stk.Push(x);
          foreach (a ∈ P.S.Σ ∧ y ∈ Post_a(x))
            if (Dfn(y) = −1)
              Strong − SCC(y, truecall);
            endif
            if (not InComponent(y))
              if (LL(y) ≤ LL(x))
                LL(x) := LL(y);
*               loop := true;
              endif
            endif
          endfor
(1)       if (LL(x) = Dfn(x))
            scc := ∅;
            while (stk ≠ ∅)
              y := stk.Top();
              if (Dfn(y) < Dfn(x))
                break;
              y := stk.Pop();
              InComponent(y) := true;
              scc.Push(y);
            endwhile
*           if (truecall)
              SCCs.Push(scc);
*           if (loop)
*             Elarg(x) := true;
*             Split(x, scc);
*           endif
          endif
        endproc Strong − SCC;
```

TAB. A.7 – *Stratégie* SCSC-STR - *II*

```
procedure Split (x : P.S.Q; scc : FILE)
  var
    y : P.S.Q;
  begin
    foreach (y ∈ scc ∧ y ≠ x)
      Dfn(y) := −1;
      LL(y) := −1;
      InComponent(y) := false;
    endfor
    foreach (a ∈ P.S.Σ ∧ y ∈ Post_a(x))
      if (Dfn(y) = −1)
        Strong − SCC(y, false);
  endproc Split ;
```

TAB. A.8 – *Stratégie* SCSC-STR - *III*

```
procedure Iterate
  var
    scc : file de P.S.Q;
  begin
    while (scc ≠ ∅)
      scc := SCCs.Pop();
      Widening(scc);
    endwhile
    Narrowing();
  endproc Iterate ;
```

TAB. A.9 – *Stratégie* SCSC-STR - *IV*

```
procedure Widening(var wl : FILE)
  var
    ℓ, ℓ' : P.L;
    q, q' : P.S.Q;
    C : ensemble de P.S.Q;
  begin
    C := wl;
    X := P.X_0;
    while (wl ≠ ∅)
      q := wl.Pop();
      ℓ := P.E(X)(q);
      if (ℓ ⊑ X(q))
        ℓ' := X(q);
      else if (ℓ ⋢ X(q) ∧ ¬Elarg(q))
        ℓ' := ℓ;
      else if (ℓ ⋢ X(q) ∧ Elarg(q))
        ℓ' := X(q)∇ℓ;
      endif
      if (ℓ' ≠ X(q))
        X(q) := ℓ';
        wl.Push(P.S.Post(q) ∩ C);
      endif
    endwhile
  endproc Widening;
```

TAB. A.10 – *Stratégie* SCSC-STR - *V*

```
procedure Narrowing
  var
    Y, Z : P.S.Q → P.L;
    q : P.S.Q;
  begin
    repeat
      Y := X;
      foreach (q ∈ P.S.Q)
        Z(q) := P.E(X)(q);
      foreach (q ∈ P.S.Q)
        X(q) := X(q)△Z(q);
    until (X = Y);
  endproc Narrowing;
```

TAB. A.11 – *Stratégie* SCSC-STR - *VI*

```
procedure Initialize
  begin
    SCCs := ∅;
    stk := ∅;
    cnt := 0;
    foreach (x ∈ P.S.Q)
      Dfn(x) := −1;
      LL(x) := −1;
      InComponent(x) := false;
      Elarg(x) := false;
    endfor
    foreach (x ∈ P.S.Q)
      if (Dfn(x) = −1)
        Strong − SCC(x,true);
    endfor
  endproc Initialize;
```

TAB. A.12 – *Stratégie* SCSC-STR - *VII*

Annexe B

Preuves

Lemme 2.1[p.29]

(1[p.29]). En utilisant la définition, on montre par exemple la réflexivité : $(x_1, x_2) \sqsubseteq (x_1, x_2) \overset{def}{\iff} x_1 \sqsubseteq_1 x_1 \wedge x_2 \sqsubseteq_2 x_2 \overset{def}{\iff} vrai$.

(3[p.29]). On montre, en utilisant les définitions, que $(\sqcup X_1, \sqcup X_2)$ est une borne supérieure et ensuite qu'il est la plus petite borne supérieure.

□

Lemme 2.2[p.30]

Preuve similaire au lemme 2.1[p.29].

□

Définition B.1 (Treillis dual)

Le treillis *dual* de treillis complet $(L, \sqsubseteq, \sqcup, \sqcap, \bot, \top)$ est le treillis $(L, \sqsubseteq^{-1}, \Cup, \Cap, \curlywedge, \curlyvee)$ t.q. $x \Cup y = x \sqcap y$, $x \Cap y = x \sqcup y$ et $\curlywedge = \top$ et $\curlyvee = \bot$. ▪

Lemme B.1

Le treillis dual d'un treillis complet est complet. ▪

Lemme B.2

Soient $(L, \sqsubseteq, \sqcup, \sqcap, \bot, \top)$ un treillis complet, $(L, \sqsubseteq^{-1}, \Cup, \Cap, \curlywedge, \curlyvee)$ son treillis dual et $f : L \to L$ une fonction totale. Si f est continue dans $(L, \sqsubseteq, \sqcup, \sqcap, \bot, \top)$ alors elle est aussi dans $(L, \sqsubseteq^{-1}, \Cup, \Cap, \curlywedge, \curlyvee)$ (cela signifie $(\forall X)(X \subseteq L \wedge X \neq \emptyset \Rightarrow f(\sqcap X) = \sqcap f(X))$) ▪

Lemme 2.4[p.32]

Soit $(x_n)_{n\in\mathbf{N}}$ une chaîne. Il faut montrer que $f(\sqcup_{n\in\mathbf{N}}x_n) = \sqcup_{n\in\mathbf{N}}f(x_n)$. L treillis complet assure l'existence de l'élément $\sqcup_{n\in\mathbf{N}}x_n$. Comme $x_i \sqsubseteq \sqcup_{n\in\mathbf{N}}x_n$, pour tout $i \in \mathbf{N}$ et f est monotone on obtient $f(x_i) \sqsubseteq f(\sqcup_{n\in\mathbf{N}}x_n)$, pour tout $i \in \mathbf{N}$ et donc $\sqcup_{n\in\mathbf{N}}f(x_n) \sqsubseteq f(\sqcup_{n\in\mathbf{N}}x_n)$.

Le fait que $(x_n)_{n\in\mathbf{N}}$ est une chaîne et L est de la hauteur finie implique que $(x_n)_{n\in\mathbf{N}}$ est constante à partir d'un certain rang, soit-il n_0. Donc $\sqcup_{n\in\mathbf{N}}x_n = x_{n_0}$ et on obtient $f(\sqcup_{n\in\mathbf{N}}x_n) = f(x_{n_0})$. Mais $f(x_{n_0}) \sqsubseteq \sqcup_{n\in\mathbf{N}}f(x_n)$ et donc on conclut la preuve.

□

Théorème 2.1[p.32]

($1_{[p.32]}$).

Le fait que L est un treillis complet assure l'existence de $\sqcup_{n\in\mathbf{N}}f^n(\bot)$. Soit $\phi \doteqdot \sqcup_{n\in\mathbf{N}} f^n(\bot)$. On montre par induction que $(\forall n)(n \geq 0 \Rightarrow f^n(\bot) \sqsubseteq \mathit{lfp}(f))$:

- Comme L est un treillis complet on a que $f^0(\bot) = \bot \sqsubseteq \mathit{lfp}(f)$.
- Supposons-nous que $f^n(\bot) \sqsubseteq \mathit{lfp}(f)$. f est monotone donc on obtient que $f^{n+1}(\bot) \sqsubseteq f(\mathit{lfp}(f)) = \mathit{lfp}(f)$.

Donc $\{f^n(\bot) \mid n \geq 0\} \sqsubseteq \mathit{lfp}(f)$ et donc $\phi \sqsubseteq \mathit{lfp}(f)$.

La deuxième partie du cas se prouve de manière analogue.

($2_{[p.32]}$).

Soit $\phi \doteqdot \sqcup_{n\in\mathbf{N}}f^n(\bot)$, on obtient alors : $f(\phi) = f(\sqcup_{n\in\mathbf{N}}f^n(\bot)) \overset{\text{f cont.}}{=} \sqcup_{n\in\mathbf{N}}f^{n+1}(\bot) = \phi$. Donc $\phi \in \Phi_f$ et ensuite, comme $\mathit{lfp}(f) \doteqdot \sqcap\Phi_f$ on a $\mathit{lfp}(f) \sqsubseteq \phi$.

f est continue donc elle est monotone et on a prouvé dans le cas $1_{[p.32]}$ que $\phi \sqsubseteq \mathit{lfp}(f)$.

□

Théorème 2.4[p.35]

Quand f est continue, la preuve se trouve dans [Cou98], page 28. Si f est monotone et L est de la hauteur finie on effectue la preuve en deux étapes :

1. On montre que $\sqcup_{k\in\mathbf{N}}x^k$ est un point fixe et
2. $\sqcup_{k\in\mathbf{N}}x^k \sqsubseteq \mathit{lfp}(f)$.

1. La séquence $(x^k)_{k\in\mathbf{N}}$ est croissante[1]. Comme la hauteur de L est finie on obtient que la séquence $(x^k)_{k\in\mathbf{N}}$ est constante à partir d'un certain rang : $(\exists k_0)(\forall k)(k \geq k_0 \Rightarrow x^k = x^{k_0})$. Donc $\sqcup_{k\in\mathbf{N}}x^k = x^{k_0}$.

Fait B.1

1. on montre par induction

$(\forall k)(k \geq k_0 \wedge i \in \beta(k) \Rightarrow f_i(x_1^k, \ldots, x_n^k) = x_i^{k_0})$. ▪

Preuve

On montre par induction et on s'appuie sur le fait que la séquence $(x^k)_{k \in \mathbf{N}}$ est constante à partir de k_0.

□

Maintenant on a $f(\sqcup_{k \in \mathbf{N}} x^k) = f(x^{k_0}) = (f_i(x_1^k, \ldots, x_n^k))_{1 \leq i \leq n}$. Prenons $k_i = \min\{k \mid k \geq k_0 \wedge i \in \beta(k)\}$ et $k' = \max_{i=1}^{n} k_i$. On obtient $f(x^{k_0}) = x^{k'} = x^{k_0}$ et donc x^{k_0} est un point fixe.

2. On a l'inégalité $lfp(f) \sqsubseteq x^{k_0}$. Il nous reste à montrer l'inégalité inverse : $x^{k_0} \sqsubseteq lfp(f)$.

Fait B.2

$(\forall k)(k \in \mathbf{N} \Rightarrow x^k \sqsubseteq lfp(f))$. ▪

Preuve

On montre par induction : $x^0 = (\bot, \ldots, \bot) \sqsubseteq lfp(f)$ [2] et supposant que $x^k \sqsubseteq lfp(f)$, comme f est monotone on obtient $f(x^k) \sqsubseteq lfp(f)$. On a deux cas :

(a) Si $i \in \beta(k)$ on a $x_i^{k+1} = f_i(x_1^k, \ldots, x_n^k) \sqsubseteq lfp_i(f)$,

(b) Si $i \notin \beta(k)$ on a $x_i^{k+1} = x_i^k \sqsubseteq lfp_i(f)$.

Donc on a $x^{k+1} \sqsubseteq lfp(f)$.

□

En utilisant le fait précédent on obtient $x^{k_0} = \sqcup_{k \in \mathbf{N}} x^k \sqsubseteq lfp(f)$.

□

Lemme 4.1[p.87]

Soient $\widehat{SP} = (\hat{Q}_1, \hat{\Sigma}, \{\xrightarrow{a}_1 \mid a \in \hat{\Sigma}\}, \hat{q}_0^1)$ et $\widehat{SP \setminus_{\mathrm{f}} \Sigma_{\mathrm{f}}} = (\hat{Q}_2, \hat{\Sigma}, \{\xrightarrow{a}_2 \mid a \in \hat{\Sigma}\}, \hat{q}_0^2)$. On observe que les deux STEs ont le même ensemble d'étiquettes $(\hat{\Sigma})$ et cela résulte de la définition 4.1[p.86] de l'utilité de code: $(\forall p)(\forall a)(p \in P \wedge a \in \Sigma_p \Rightarrow \xrightarrow[\setminus \mathrm{f}]{a}_p \subseteq \xrightarrow{a}_p)$.

2. car L est un treillis complet

Il faudra prouver $\hat{Q}_2 \subseteq \hat{Q}_1$ et $\xrightarrow{a}_2 \subseteq \xrightarrow{a}_1, \forall a \in \hat{\Sigma}$. À cause du fait que $\widehat{SP}$ et $\widehat{SP \setminus_{\mathrm{f}} \Sigma_{\mathrm{f}}}$ sont implicitement connexes (ils contiennent seulement les états accessibles), il suffit de prouver que chaque état accessible dans $\hat{Q}_2$ appartient à $\hat{Q}_1$ et que tous ses successeurs dans $\hat{Q}_2$ sont aussi dans $\hat{Q}_1$.

On le montre en utilisant l'induction :

1. $\hat{q}_0^1 = \hat{q}_0^2$ et
2. $(\forall q)(\forall a)(q \in \hat{Q}_1 \cap \hat{Q}_2 \wedge a \in \hat{\Sigma} \Rightarrow Post_a^2(q) \subseteq Post_a^1(q))$.

1. Conformément aux règles $4.1_{[\mathrm{p.86}]}$ et $3.11_{[\mathrm{p.49}]}$ $\hat{q}_0^2 = ((q_0^p)_{p\in P}, 0^{\sum_{p\in P}|X_p|}, \epsilon^{|C^{\mathrm{int}}|}) = \hat{q}_0^1$.

2. Soient $a \in \hat{\Sigma}$, $q \in \hat{Q}_1 \cap \hat{Q}_2$ et $q' \in \hat{Q}_2$ t.q. $q \xrightarrow{a}_2 q'$. On montre que $q' \in \hat{Q}_1$ et $q \xrightarrow{a}_1 q'$ en utilisant une induction structurelle en fonction de l'étiquette a et des règles $3.11_{[\mathrm{p.49}]} \div 3.16_{[\mathrm{p.49}]}$ et $4.1_{[\mathrm{p.86}]} \div 4.5_{[\mathrm{p.86}]}$:

   (a) Soit $a = \tau \in \hat{\Sigma}$. La transition $q \xrightarrow{\tau}_2 q'$ a été obtenue en appliquant une des règles $3.12_{[\mathrm{p.49}]}$, $3.14_{[\mathrm{p.49}]}$ et $3.16_{[\mathrm{p.49}]}$:
   $(3.12_{[\mathrm{p.49}]})$. On a appliqué la règle :

$$\frac{q \doteqdot (\theta, \sigma, \rho) \in \hat{Q}_2 \quad q_p \xrightarrow[\setminus_{\mathrm{f}}]{[b]\ x:=e}_p q'_p \quad \sigma(b) = \mathbf{t} \quad \sigma(e) = v}{q' \doteqdot ([q'_p/p]\theta, [v/x]\sigma, \rho) \in \hat{Q}_2 \quad q \xrightarrow{\tau}_2 q'} \ .$$

   Les transitions de la forme $q_p \xrightarrow[\setminus_{\mathrm{f}}]{[b]\ x:=e}_p q'_p$ ont été obtenues uniquement à partir des transitions $q_p \xrightarrow{[b]\ x:=e}_p q'_p$ en appliquant la règle $4.2_{[\mathrm{p.86}]}$.
   Donc la règle $3.12_{[\mathrm{p.49}]}$ peut être appliquée dans $\widehat{SP}$:

$$\frac{q \doteqdot (\theta, \sigma, \rho) \in \hat{Q}_1 \quad q_p \xrightarrow{[b]\ x:=e}_p q'_p \quad \sigma(b) = \mathbf{t} \quad \sigma(e) = v}{q' \doteqdot ([q'_p/p]\theta, [v/x]\sigma, \rho) \in \hat{Q}_1 \quad q \xrightarrow{\tau}_1 q'} \ .$$

   $(3.14_{[\mathrm{p.49}]})$. On applique un raisonnement similaire celui du cas de la règle $3.12_{[\mathrm{p.49}]}$ mais cette fois-ci en utilisant la règle $4.3_{[\mathrm{p.86}]}$.
   $(3.16_{[\mathrm{p.49}]})$. On applique un raisonnement similaire celui du cas de la règle $3.12_{[\mathrm{p.49}]}$ mais cette fois-ci en utilisant la règle $4.5_{[\mathrm{p.86}]}$.

(b) Soit $a = c?s(v) \in \hat{\Sigma}$. La transition $q \xrightarrow{c?s(v)}_2 q'$ a été obtenue en appliquant seulement la règle $3.15_{[p.49]}$. On applique un raisonnement similaire celui du cas de la règle $3.12_{[p.49]}$ mais cette fois-ci en utilisant la règle $4.4_{[p.86]}$.

(c) Soit $a = c!s(v) \in \hat{\Sigma}$. La transition $q \xrightarrow{c!s(v)}_2 q'$ a été obtenue en appliquant seulement la règle $3.13_{[p.49]}$. On applique un raisonnement similaire celui du cas de la règle $3.12_{[p.49]}$ mais cette fois-ci en utilisant la règle $4.3_{[p.86]}$.

□

Théorème $4.1_{[p.90]}$
Soient :

1. $TP = (X_{\mathrm{tp}}, (Q_{\mathrm{tp}}, \Sigma_{\mathrm{tp}}, \{\xrightarrow{a}_{\mathrm{tp}} | \ a \in \Sigma_{\mathrm{tp}}\}, q_0^{\mathrm{tp}}))$.
2. $\widehat{SP} = (\hat{Q}_1, \hat{\Sigma}, \{\xrightarrow{a}_1 | \ a \in \hat{\Sigma}\}, \hat{q}_0^1)$.
3. $\widehat{SP \setminus_{\mathrm{f}} \Sigma_{\mathrm{f}}} = (\hat{Q}_2, \hat{\Sigma}, \{\xrightarrow{a}_2 | \ a \in \hat{\Sigma}\}, \hat{q}_0^2)$.
4. $\prod(\widehat{SP}, TP, \Sigma_{\mathrm{f}}) = (Q_1, \hat{\Sigma}, \{\overset{a}{\multimap}_1 | \ a \in \hat{\Sigma}\}, q_0^1)$.
5. $\prod(\widehat{SP \setminus_{\mathrm{f}} \Sigma_{\mathrm{f}}}, TP, \Sigma_{\mathrm{f}}) = (Q_2, \hat{\Sigma}, \{\overset{a}{\multimap}_2 | \ a \in \hat{\Sigma}\}, q_0^2)$.

On reprend les règles $3.35_{[p.82]} \div 3.40_{[p.82]}$ mais cette fois-ci avec les notations utilisées dans la preuve pour $\widehat{SP}$, $\widehat{SP \setminus_{\mathrm{f}} \Sigma_{\mathrm{f}}}$, $\prod(\widehat{SP}, TP, \Sigma_{\mathrm{f}})$, $\prod(\widehat{SP \setminus_{\mathrm{f}} \Sigma_{\mathrm{f}}}, TP, \Sigma_{\mathrm{f}})$ et pour $i \in 1 \,..\, 2$:

$$\frac{-}{q_0^i = (\hat{q}_0^i, q_0^{\mathrm{tp}}) \in Q_i} \tag{B.1}$$

$$\frac{(\hat{q}_i, q_{\mathrm{tp}}) \in Q_i \quad \hat{q}_i \xrightarrow{\tau}_i \hat{q}_i' \quad q_{\mathrm{tp}} \notin \{q_a^{\mathrm{tp}}, q_r^{\mathrm{tp}}\}}{(\hat{q}_i', q_{\mathrm{tp}}) \in Q_i \quad (\hat{q}_i, q_{\mathrm{tp}}) \overset{\tau}{\multimap}_i (\hat{q}_i', q_{\mathrm{tp}})} \tag{B.2}$$

$$\frac{(\hat{q}_i, q_{\mathrm{tp}}) \in Q_i \quad \hat{q}_i \xrightarrow{c?s(v)}_i \hat{q}_i' \quad q_{\mathrm{tp}} \xrightarrow{c?s(\ell)}_{\mathrm{tp}} q'_{\mathrm{tp}} \quad v \in \phi(\ell)}{(\hat{q}_i', q'_{\mathrm{tp}}) \in Q_i \quad (\hat{q}_i, q_{\mathrm{tp}}) \overset{c?s(v)}{\multimap}_i (\hat{q}_i', q'_{\mathrm{tp}})} \tag{B.3}$$

$$\frac{(\hat{q}_i, q_{\mathrm{tp}}) \in Q_i \quad \hat{q}_i \xrightarrow{c?s(v)}_i \hat{q}_i' \quad c?s(\ell) \in \Sigma_{\mathrm{f}} \quad v \in \phi(\ell)}{(\hat{q}_i', q_{\mathrm{tp}}) \in Q_i \quad (\hat{q}_i, q_{\mathrm{tp}}) \overset{c?s(v)}{\multimap}_i (\hat{q}_i', q_{\mathrm{tp}})} \tag{B.4}$$

$$\frac{(\hat{q}_i, q_{\mathrm{tp}}) \in Q_i \quad \hat{q}_i \xrightarrow{c!s(v)}_i \hat{q}'_i \quad q_{\mathrm{tp}} \xrightarrow{c!s(\ell)}_{\mathrm{tp}} q'_{\mathrm{tp}} \quad v \in \phi(\ell) \vee \ell = \oslash}{(\hat{q}'_i, q'_{\mathrm{tp}}) \in Q_i \quad (\hat{q}_i, q_{\mathrm{tp}}) \overset{c!s(v)}{\multimap}_i (\hat{q}'_i, q'_{\mathrm{tp}})} \tag{B.5}$$

$$\frac{(\hat{q}_i, q_{\mathrm{tp}}) \in Q_i \quad \hat{q}_i \xrightarrow{c!s(v)}_i \hat{q}'_i}{(\hat{q}'_i, q_{\mathrm{tp}}) \in Q_i \quad (\hat{q}_i, q_{\mathrm{tp}}) \overset{c!s(v)}{\multimap}_i (\hat{q}'_i, q_{\mathrm{tp}})} \tag{B.6}$$

On montre que l'égalité entre les états est aussi une bisimulation et donc c'est la bisimulation forte et comme $q_0^1 = q_0^2$ on obtient $q_0^1 \sim q_0^2$ et donc $\prod(\widehat{SP}, TP, \Sigma_{\mathrm{f}}) \sim \prod(\widehat{SP \setminus_{\mathrm{f}} \Sigma}_{\mathrm{f}}, TP, \Sigma_{\mathrm{f}})$:

$$\begin{array}{lll} (\forall p_1)(\forall p_2)(\forall a) & (& (p_1, p_2) \in Q_1 \times Q_2 \wedge p_1 = p_2 \wedge \ a \in \hat{\Sigma} \wedge \\ & & (\forall q_1)(q_1 \in Q_1 \wedge p_1 \overset{a}{\multimap}_1 q_1 \Rightarrow \\ & & \quad (\exists q_2)(q_2 \in Q_2 \ \wedge \ p_2 \overset{a}{\multimap}_2 q_2 \ \wedge \ q_1 = q_2) \ ) \\ & & \wedge \\ & & (\forall q_2)(q_2 \in Q_2 \wedge p_2 \overset{a}{\multimap}_2 q_2 \Rightarrow \\ & & \quad (\exists q_1)(q_1 \in Q_1 \ \wedge \ p_1 \overset{a}{\multimap}_1 q_1 \ \wedge \ q_1 = q_2) \ ) \\ &) & \end{array} \ . \tag{B.7}$$

La deuxième partie résulte en utilisant le corollaire $4.1_{[\mathrm{p.90}]}$.

On montre la première partie en utilisant une induction structurelle en fonction de l'étiquette a et des règles $B.1_{[\mathrm{p.177}]} \div B.6_{[\mathrm{p.178}]}$:

1. Soit $a = \tau \in \hat{\Sigma}$, $q_1 \in Q_1$ et $p_1 \overset{\tau}{\multimap}_1 q_1$.

   Observons-nous que la transition $p_1 \overset{\tau}{\multimap}_1 q_1$ a pu être construite en utilisant la règle $B.2_{[\mathrm{p.177}]}$:

   $$\frac{p_1 \doteqdot (\hat{p}_1, q_{\mathrm{tp}}) \in Q_1 \quad \hat{p}_1 \xrightarrow{\tau}_1 \hat{q}_1 \quad q_{\mathrm{tp}} \notin \{q_a^{\mathrm{tp}}, q_r^{\mathrm{tp}}\}}{q_1 \doteqdot (\hat{q}_1, q_{\mathrm{tp}}) \in Q_1 \quad p_1 \overset{\tau}{\multimap}_1 q_1} \ .$$

   On voudrait appliquer la même règle mais cette fois-ci pour $\prod(\widehat{SP \setminus_{\mathrm{f}} \Sigma}_{\mathrm{f}}, TP, \Sigma_{\mathrm{f}})$:

   $$\frac{p_2 \doteqdot (\hat{p}_2, q_{\mathrm{tp}}) \in Q_2 \quad \hat{p}_2 \xrightarrow{\tau}_2 \hat{q}_2 \quad q_{\mathrm{tp}} \notin \{q_a^{\mathrm{tp}}, q_r^{\mathrm{tp}}\}}{q_2 \doteqdot (\hat{q}_2, q_{\mathrm{tp}}) \in Q_2 \quad p_2 \overset{\tau}{\multimap}_2 q_2} \ .$$

   Donc, si on prouve l'existence d'un état $\hat{q}_2 \in \hat{Q}_2$ t.q. $\hat{q}_1 = \hat{q}_2$ et $\hat{p}_2 \xrightarrow{\tau}_2 \hat{q}_2$ on obtient $q_1 = q_2$ et parce que $\hat{p}_1 = \hat{p}_2$ (car $p_1 = p_2$) on voit que les prémisses de la

règle $B.2_{[\text{p.177}]}$ dans $\prod(\widehat{SP \setminus_{\mathrm{f}} \Sigma_{\mathrm{f}}}, TP, \Sigma_{\mathrm{f}})$ sont vraies et donc on obtient $q_2 \in Q_2$ et $p_2 \xrightarrow{\tau}_2 q_2$.

La transition $\hat{p}_1 \xrightarrow{\tau}_1 \hat{q}_1$ a été obtenue en appliquant une des règles $3.12_{[\text{p.49}]}$, $3.14_{[\text{p.49}]}$, $3.16_{[\text{p.49}]}$:

$(3.12_{[\text{p.49}]})$. La règle appliquée est

$$\frac{\hat{p}_1 \doteqdot (\theta, \sigma, \rho) \in \hat{Q}_1 \quad q_p \xrightarrow{[b]\ x:=e}_p q'_p \quad \sigma(b) = \mathbf{t} \quad \sigma(e) = v}{\hat{q}_1 \doteqdot ([q'_p/p]\theta, [v/x]\sigma, \rho) \in \hat{Q}_1 \quad \hat{p}_1 \xrightarrow{\tau}_1 \hat{q}_1} \ .$$

Mais les transitions de la forme $q_p \xrightarrow{[b]\ x:=e}_p q'_p$ sont gardées par $\setminus_{\mathrm{f}}$ conformément à la règle $4.2_{[\text{p.86}]}$ et donc la règle $3.12_{[\text{p.49}]}$ peut être appliquée dans $\widehat{SP \setminus_{\mathrm{f}} \Sigma_{\mathrm{f}}}$:

$$\frac{\hat{p}_2 = \hat{p}_1 \doteqdot (\theta, \sigma, \rho) \in \hat{Q}_2 \quad q_p \xrightarrow[\setminus_{\mathrm{f}}]{[b]\ x:=e}_p q'_p \quad \sigma(b) = \mathbf{t} \quad \sigma(e) = v}{\hat{q}_2 \doteqdot ([q'_p/p]\theta, [v/x]\sigma, \rho) \in \hat{Q}_2 \quad \hat{p}_2 \xrightarrow{\tau}_2 \hat{q}_2} \ .$$

Donc on obtient $\hat{q}_2 = \hat{q}_1 = ([q'_p/p]\theta, [v/x]\sigma, \rho)$ et $\hat{q}_2 \in \hat{Q}_2$ et $\hat{p}_2 \xrightarrow{\tau}_2 \hat{q}_2$.

$(3.14_{[\text{p.49}]})$. On applique un raisonnement similaire celui du cas de la règle $3.12_{[\text{p.49}]}$: on tient compte du fait que $\setminus_{\mathrm{f}}$ garde les transitions de la forme $q_p \xrightarrow{[b]\ c!s(e)}_p q'_p$ conformément à la règle $4.3_{[\text{p.86}]}$.

$(3.16_{[\text{p.49}]})$. La règle appliquée est

$$\frac{\hat{p}_1 \doteqdot (\theta, \sigma, \rho) \in \hat{Q}_1 \quad q_p \xrightarrow{[b]\ c?s(x)}_p q'_p \quad \sigma(b) = \mathbf{t} \quad c \in C^{\mathrm{int}} \quad \rho(c) = (s, v) \cdot w}{\hat{q}_1 \doteqdot ([q'_p/p]\theta, [v/x]\sigma, [w/c]\rho) \in \hat{Q}_1 \quad \hat{p}_1 \xrightarrow{\tau}_1 \hat{q}_1} \ .$$

On a $\hat{Q}_2 \ni \hat{p}_2 = \hat{p}_1 = (\theta, \sigma, \rho)$ (car $p_1 = p_2$). Parce que $\sigma(b) = \mathbf{t}$ et $c \in C^{\mathrm{int}}$ et $\rho(c) = (s, v) \cdot w$ on voit que la seule chose dont on a besoin pour appliquer la règle $3.16_{[\text{p.49}]}$ pour $\widehat{SP \setminus_{\mathrm{f}} \Sigma_{\mathrm{f}}}$ est $q_p \xrightarrow[\setminus_{\mathrm{f}}]{[b]\ c?s(x)}_p q'_p$.

Supposons-nous que $(q_p, q'_p) \notin \xrightarrow[\setminus_{\mathrm{f}}]{[b]\ c?s(x)}_p$. Parce que $c \in C^{\mathrm{int}}$ la règle $4.5_{[\text{p.86}]}$ est la seule pouvant s'appliquer et le fait qu'elle ne peut pas être appliquée est dû au fait que dans tout processus r il n'y a pas de transitions de la forme $q_r \xrightarrow[\setminus_{\mathrm{f}}]{[b']\ c!s(e)}_p q'_r$.

Cela signifie que, pour tout processus r, aucune transition de la forme $q_r \overset{[b'] \; c!s(e)}{\longrightarrow}_p q'_r$ n'existe pas (car conformément à la règle $4.3_{[\text{p.86}]}$ $\setminus_{\text{f}}$ garde les transitions de cette forme).

Mais cela est faux car la file c contient dans l'état (θ, σ, ρ) le message $s(v) : \rho(c) = (s, v) \cdot w$.

Donc on a $q_p \overset{[b] \; c?s(x)}{\xrightarrow[\setminus_{\text{f}}]{}}_p q'_p$ et maintenant on peut appliquer la règle $3.16_{[\text{p.49}]}$ pour $\widehat{SP \setminus_{\text{f}} \Sigma_{\text{f}}}$:

$$\frac{\hat{p}_2 \doteqdot (\theta, \sigma, \rho) \in \hat{Q}_2 \quad q_p \overset{[b] \; c?s(x)}{\xrightarrow[\setminus_{\text{f}}]{}}_p q'_p \quad \sigma(b) = \mathbf{t} \quad c \in C^{\text{int}} \quad \rho(c) = (s, v) \cdot w}{\hat{q}_2 \doteqdot ([q'_p/p]\theta, [v/x]\sigma, [w/c]\rho) \in \hat{Q}_2 \quad \hat{p}_2 \xrightarrow{\tau}_2 \hat{q}_2}$$

et donc on a $\hat{q}_2 = \hat{q}_1 = ([q'_p/p]\theta, [v/x]\sigma, [w/c]\rho)$ et $\hat{q}_2 \in \hat{Q}_2$ et $\hat{p}_2 \xrightarrow{\tau}_2 \hat{q}_2$.

2. Soit $a = c?s(v) \in \hat{\Sigma}$, $q_1 \in Q_1$ et $p_1 \overset{c?s(v)}{\multimap}_1 q_1$.

   Observons-nous que la transition $p_1 \overset{c?s(v)}{\multimap}_1 q_1$ a été obtenue en utilisant une des règles $B.3_{[\text{p.177}]}$, $B.4_{[\text{p.177}]}$:

   $(B.3_{[\text{p.177}]})$. La règle appliquée est

$$\frac{p_1 \doteqdot (\hat{p}_1, q_{\text{tp}}) \in Q_1 \quad \hat{p}_1 \xrightarrow{c?s(v)}_1 \hat{q}_1 \quad q_{\text{tp}} \xrightarrow{c?s(\ell)}_{\text{tp}} q'_{\text{tp}} \quad v \in \phi(\ell)}{q_1 \doteqdot (\hat{q}_1, q'_{\text{tp}}) \in Q_1 \quad p_1 \overset{c?s(v)}{\multimap}_1 q_1} \; .$$

   On a $\hat{p}_1 \in \hat{Q}_1$ (car $p_1 \in Q_1$) et donc la transition $\hat{p}_1 \xrightarrow{c?s(v)}_1 \hat{q}_1$ a été obtenue en appliquant seulement la règle $3.15_{[\text{p.49}]}$:

$$\frac{\hat{p}_1 \doteqdot (\theta, \sigma, \rho) \in \hat{Q}_1 \quad q_p \overset{[b] \; c?s(x)}{\longrightarrow}_p q'_p \quad \sigma(b) = \mathbf{t} \quad c \in C^{\text{ext}}}{\hat{q}_1 \doteqdot (\theta', [v/x]\sigma, \rho) \in \hat{Q}_1 \quad \hat{p}_1 \xrightarrow{c?s(v)}_1 \hat{q}_1} \; .$$

   La transition $q_p \overset{[b] \; c?s(x)}{\longrightarrow}_p q'_p$ est gardée par $\setminus_{\text{f}}$ conformément à la règle $4.4_{[\text{p.86}]}$ (et en plus parce que Σ_{f} couvre TP).

   Donc $q_p \overset{[b] \; c?s(x)}{\xrightarrow[\setminus_{\text{f}}]{}}_p q'_p$ et on peut appliquer la règle $3.15_{[\text{p.49}]}$ pour $\widehat{SP \setminus_{\text{f}} \Sigma_{\text{f}}}$:

$$\frac{\hat{p}_2 \doteqdot (\theta, \sigma, \rho) \in \hat{Q}_2 \quad q_p \overset{[b] \; c?s(x)}{\xrightarrow[\setminus_{\text{f}}]{}}_p q'_p \quad \sigma(b) = \mathbf{t} \quad c \in C^{\text{ext}}}{\hat{q}_2 \doteqdot (\theta', [v/x]\sigma, \rho) \in \hat{Q}_2 \quad \hat{p}_2 \xrightarrow{c?s(v)}_2 \hat{q}_2} \; .$$

Maintenant on peut appliquer la règle $B.3_{[\text{p.177}]}$ pour $\prod(\widehat{SP \setminus_{\text{f}} \Sigma_{\text{f}}}, TP, \Sigma_{\text{f}})$:

$$\frac{p_2 \doteqdot (\hat{p}_2, q_{\text{tp}}) \in Q_2 \quad \hat{p}_2 \xrightarrow{c?s(v)}_2 \hat{q}_2 \quad q_{\text{tp}} \xrightarrow{c?s(\ell)}_{\text{tp}} q'_{\text{tp}} \quad v \in \phi(\ell)}{q_2 \doteqdot (\hat{q}_2, q'_{\text{tp}}) \in Q_2 \quad p_2 \xrightarrow{c?s(v)}_2 q_2} .$$

($B.4_{[\text{p.177}]}$). La règle appliquée est

$$\frac{p_1 \doteqdot (\hat{p}_1, q_{\text{tp}}) \in Q_1 \quad \hat{p}_1 \xrightarrow{c?s(v)}_1 \hat{q}_1 \quad c?s(\ell) \in \Sigma_{\text{f}} \quad v \in \phi(\ell)}{q_1 \doteqdot (\hat{q}_1, q_{\text{tp}}) \in Q_1 \quad p_1 \xrightarrow{c?s(v)}_1 q_1} .$$

On applique un raisonnement similaire celui du cas de la règle $B.3_{[\text{p.177}]}$.

3. Soit $a = c!s(v) \in \hat{\Sigma}$, $q_1 \in Q_1$ et $p_1 \xrightarrow{c!s(v)}_1 q_1$.

   Observons-nous que la transition $p_1 \xrightarrow{c!s(v)}_1 q_1$ a été obtenue en utilisant une des règles $B.5_{[\text{p.178}]}$, $B.6_{[\text{p.178}]}$:

   ($B.5_{[\text{p.178}]}$). La règle appliquée est

$$\frac{p_1 \doteqdot (\hat{p}_1, q_{\text{tp}}) \in Q_1 \quad \hat{p}_1 \xrightarrow{c!s(v)}_1 \hat{q}_1 \quad q_{\text{tp}} \xrightarrow{c!s(\ell)}_{\text{tp}} q'_{\text{tp}} \quad v \in \phi(\ell) \vee \ell = \oslash}{q_1 \doteqdot (\hat{q}_1, q'_{\text{tp}}) \in Q_1 \quad p_1 \xrightarrow{c!s(v)}_1 q_1} .$$

   On a $\hat{p}_1 \in \hat{Q}_1$ (car $p_1 \in Q_1$) donc la transition $\hat{p}_1 \xrightarrow{c!s(v)}_1 \hat{q}_1$ a été obtenue en appliquant seulement la règle $3.13_{[\text{p.49}]}$:

$$\frac{\hat{p}_1 \doteqdot (\theta, \sigma, \rho) \in \hat{Q}_1 \quad q_p \xrightarrow{[b]\ c!s(e)}_p q'_p \quad \sigma(b) = \mathbf{t} \quad \sigma(e) = v \quad c \in C^{\text{ext}}}{\hat{q}_1 \doteqdot ([q'_p/p]\theta, \sigma, \rho) \in \hat{Q}_1 \quad \hat{p}_1 \xrightarrow{c!s(v)}_1 \hat{q}_1} .$$

   En tenant compte que la transition $q_p \xrightarrow{[b]\ c!s(e)}_p q'_p$ est gardée par $\setminus_{\text{f}}$ conformément à la règle $4.3_{[\text{p.86}]}$, on peut appliquer la règle $3.13_{[\text{p.49}]}$ pour $\widehat{SP \setminus_{\text{f}} \Sigma_{\text{f}}}$:

$$\frac{\hat{p}_2 \doteqdot (\theta, \sigma, \rho) \in \hat{Q}_2 \quad q_p \xrightarrow{[b]\ c!s(e)}_p q'_p \quad \sigma(b) = \mathbf{t} \quad \sigma(e) = v \quad c \in C^{\text{ext}}}{\hat{q}_2 \doteqdot ([q'_p/p]\theta, \sigma, \rho) \in \hat{Q}_2 \quad \hat{p}_2 \xrightarrow{c!s(v)}_2 \hat{q}_2} .$$

Maintenant on peut appliquer la règle $B.5_{[p.178]}$ pour $\prod(\widehat{SP \setminus_{\mathrm{f}} \Sigma_{\mathrm{f}}}, TP, \Sigma_{\mathrm{f}})$:

$$\frac{p_2 \doteqdot (\hat{p}_2, q_{\mathrm{tp}}) \in Q_2 \quad \hat{p}_2 \xrightarrow{c!s(v)}_2 \hat{q}_2 \quad q_{\mathrm{tp}} \xrightarrow{c!s(\ell)}_{\mathrm{tp}} q'_{\mathrm{tp}} \quad v \in \phi(\ell) \vee \ell = \oslash}{q_2 \doteqdot (\hat{q}_2, q'_{\mathrm{tp}}) \in Q_2 \quad p_2 \xrightarrow{c!s(v)}_2 q_2} .$$

($B.6_{[p.178]}$). On applique un raisonnement similaire celui du cas de la règle $B.5_{[p.178]}$.

Donc $B.7_{[p.178]}$ est prouvée et comme $q_0^1 = q_0^2$ on obtient $q_0^1 \sim q_0^2$ et donc $\prod(\widehat{SP}, TP, \Sigma_{\mathrm{f}}) \sim \prod(\widehat{SP \setminus_{\mathrm{f}} \Sigma_{\mathrm{f}}}, TP, \Sigma_{\mathrm{f}})$.

□

Lemme $5.1_{[p.105]}$

Observons d'abord que $1_L = \lambda Z.\emptyset \cup (Z \setminus \emptyset)$, $\bar{X} = \lambda Z.X \cup (Z \setminus X)$ et $\bar{\emptyset} = \lambda Z.\emptyset \cup (Z \setminus X)$ et donc ces fonctions appartiennent à F. Soient $f = \lambda Z.A_f \cup (Z \setminus B_f)$ et $g = \lambda Z.A_g \cup (Z \setminus B_g)$ deux fonctions de F. On montre $f \circ g, f \cup g, f \cap g \in F$:

1. $(f \circ g)(X) = A_f \cup (g(X) \setminus B_f) = A_f \cup ((A_g \cup (X \setminus B_g)) \setminus B_f) =$
   $A_f \cup ((A_g \setminus B_f) \cup (X \setminus B_g \setminus B_f)) = (A_f \cup (A_g \setminus B_f)) \cup (X \setminus (B_g \cup B_f))$.
2. $(f \cup g)(X) = A_f \cup (X \setminus B_f) \cup A_g \cup (X \setminus B_g) = (A_f \cup A_g) \cup (X \setminus (B_f \cap B_g))$.
3. On montre d'abord les identités :

$$A \cup (B \setminus C) = (A \cup B) \setminus (C \setminus A) \tag{B.8}$$

$$(A_1 \setminus B_1) \cap (A_2 \setminus B_2) = ((A_1 \cap A_2) \setminus B_1) \setminus B_2 \tag{B.9}$$

Pour l'identité $B.8_{[p.182]}$: $A \cup (B \setminus C) = A \cup (B \cap \bar{C}) = (A \cup B) \cap (A \cup \bar{C}) = (A \cup B) \cap \overline{C \setminus A} = (A \cup B) \setminus (C \setminus A)$.

Pour l'identité $B.9_{[p.182]}$: $(A_1 \setminus B_1) \cap (A_2 \setminus B_2) = A_1 \cap \bar{B}_1 \cap A_2 \cap \bar{B}_2 = (A_1 \cap A_2) \cap (\bar{B}_1 \cap \bar{B}_2) = ((A_1 \cap A_2) \setminus B_1) \setminus B_2$.

Donc conformément à $B.8_{[p.182]}$ on a :

$f(Z) = A_f \cup (Z \setminus B_f) = (A_f \cup Z) \setminus (B_f \setminus A_f)$ et

$g(Z) = A_g \cup (Z \setminus B_g) = (A_g \cup Z) \setminus (B_g \setminus A_g)$.

Donc $(f \cap g)(Z) = ((A_f \cup Z) \setminus (B_f \setminus A_f)) \cap ((A_g \cup Z) \setminus (B_g \setminus A_g))$.

Maintenant on applique $B.9_{[p.182]}$ et on obtient : $(f \sqcap g)(Z) = ((A_f \cup Z) \cap (A_g \cup Z)) \setminus (B_f \setminus A_f) \setminus (B_g \setminus A_g)$.

Mais $(A_f \cup Z) \cap (A_g \cup Z) = (A_f \cap (A_g \cup Z)) \cup (Z \cap (A_g \cup Z)) =$

$Z \cup ((A_f \cap A_g) \cup (Z \cap A_f)) = Z \cup (A_f \cap A_g)$.

Donc $(f \sqcap g)(Z) = (A_f \cap A_g) \cup (Z \setminus (B_f \setminus A_f) \setminus (B_g \setminus A_g))$.

□

Lemme $5.5_{[p.107]}$

Observons d'abord que $1_L, \bar{\top}, \bar{\bot} \in F$:

1. $1_L = f_0 \circ f_1 \circ \ldots \circ f_n$, où $F \ni f_i = \lambda Z.[Z(x_i)/x_i]\ Z$, pour tout $i \in \{0, \ldots n\}$ et $X = \{x_0, \ldots, x_n\}$,
2. $\bar{\top} = f_0 \circ f_1 \circ \ldots \circ f_n$, où $F \ni f_i = \lambda Z.[\top/x_i]\ Z$, pour tout $i \in \{0, \ldots n\}$ et $X = \{x_0, \ldots, x_n\}$,
3. $\bar{\bot} = f_0 \circ f_1 \circ \ldots \circ f_n$, où $F \ni f_i = \lambda Z.[\bot/x_i]\ Z$, pour tout $i \in \{0, \ldots n\}$ et $X = \{x_0, \ldots, x_n\}$.

Soient $f_i = \lambda Z_i.[v_i/x_i]\ Z_i \in F$, $v_i \in \text{CONST}$, $x_i \in X$, pour $i \in \{1, 2\}$. On montre $f_1 \circ f_2, f_1 \sqcup_F f_2, f_1 \sqcap_F f_2 \in F$:

1. $(f_1 \circ f_2)(Z) = f_1(f_2(Z)) = f_1([v_2/x_2]Z) = [v_1/x_1][v_2/x_2]Z$.

   Donc on obtient $f_1 \circ f_2 \in F$.

2. $(f_1 \sqcup_F f_2)(Z) = [v_1/x_1]Z \sqcup_L [v_2/x_2]Z = \begin{cases} [v_1 \sqcup_L v_2/x_1]\ Z & , \quad x_1 = x_2 \\ [v_1/x_1][v_2/x_2]Z & , \quad x_1 \neq x_2 \end{cases}$

   Donc $f_1 \sqcup_F f_2 \in F$.

3. $f_1 \sqcap_F f_2 \in F$ se montre de manière similaire au cas $f_1 \sqcup_F f_2 \in F$.

□

Lemme $5.7_{[p.107]}$

On montre le résultat pour $\hat{q}_0$ et ensuite on montre que si le résultat est vrai pour un état alors il est vrai pour tous ses successeurs :

1. L'état initial de $\widehat{SP}$ est $\hat{q}_0 = (q_0, \bar{0}, \bar{\epsilon})$. On a $Const_0(q_0) = \bar{\top}$ et cette valeur ne se modifie pas pour $Y(q_0)$ (car l'opérateur de combinaison est $\sqcup_{\text{CONST}}$).

   Donc on a $\bar{0}(x) = 0 \in \phi(Y(q_0)(x)) = \phi(\top) = \mathbf{Z}$, pour toutes les variables x.

2. Supposons-nous que le résultat est vrai dans l'état (q, σ, ρ). On montre qu'il est vrai aussi dans l'état (q', σ', ρ') obtenu par l'application de la transition $q \xrightarrow{[b]\ \alpha} q'$.

   Donc on a $\sigma(x) \in \phi(Y(q)(x))$, $\forall x \in X$ et on montre que $\sigma'(x) \in \phi(Y(q')(x))$, $\forall x \in X$.

   On a $T([b]\ \alpha)(Y(q)) \sqsubseteq Y(q')$ (cela s'obtient parce que $Const_0(q_0) = \bar{\top}$ dans l'expression de l'opérateur de flot dans le problème CONST).

   L'action α peut avoir une des formes :

   (a) Soit α l'action $x := e$. On a $\sigma'(y) = \begin{cases} \sigma(e) & , \quad y = x \\ \sigma(y) & , \quad y \neq x \end{cases}$.

   Donc $\sigma'(x) = \sigma(e) \in \phi(Y(q)(e))$. Mais $Y(q)(e) = T([b]\ \alpha)(Y(q))(x) \sqsubseteq Y(q')(x)$. Donc on obtient $\sigma'(x) \in \phi(Y(q')(x))$.

   Si $y \neq x$, alors $\sigma'(y) = \sigma(y) \in \phi(Y(q)(y))$. Mais $Y(q)(y) = T([b]\ \alpha)(Y(q))(y) \sqsubseteq Y(q')(y)$ et donc on obtient $\sigma'(y) \in \phi(Y(q')(y))$, pour $y \in X \setminus \{x\}$.

   (b) Soit α l'action $c\ ?\ s(x)$. Dans notre cas $C^{\text{int}} = \emptyset$ (car notre spécification a un seul processus et on peut considérer, sans restreindre la généralité, que les files d'entrée diffèrent des files de sortie, pour chaque processus d'une spécification).

   On fait aussi la convention que si on applique la règle $3.15_{[\text{p.49}]}$ pour plusieurs $v \in D \setminus \{\oslash\}$ on prend $Env(c, s) = \top$, pour tout $(c, s) \in C^{\text{ext}} \times S$. Si on considère l'application de la règle $3.15_{[\text{p.49}]}$ seulement pour $v_0 \in D \setminus \{\oslash\}$ alors $Env(c, s) = v_0$, pour (c, s) considérée.

   En fonction des deux cas considérés on peut avoir $T([b]\ \alpha)(Y(q))(x) = \top$ ou $T([b]\ \alpha)(Y(q))(x) = v_0$ et pour $y \neq x$ $T([b]\ \alpha)(Y(q))(y) = Y(q)(y)$.

   En tenant aussi compte du fait que $T([b]\ \alpha)(Y(q)) \sqsubseteq Y(q')$ on obtient $Y(q)(y) \sqsubseteq Y(q')(y)$, pour $y \neq x$ et soit $\top = Y(q')(x)$ soit $v_0 \sqsubseteq Y(q')(x)$.

   On a (conformément à la règle $3.15_{[\text{p.49}]}$) $\sigma'(y) = \begin{cases} v & , \quad y = x \\ \sigma(y) & , \quad y \neq x \end{cases}$.

   Si $y \neq x$ alors $\sigma'(y) = \sigma(y) \in \phi(Y(q)(y))$ et parce que $Y(q)(y) \sqsubseteq Y(q')(y)$ on obtient $\sigma'(y) \in \phi(Y(q')(y))$ pour $y \neq x$.

   Si $y = x$ alors soit $Y(q')(x) = \top$ et $\sigma'(x) = v \in \mathbf{Z} = \phi(\top) = \phi(Y(q')(x))$ soit $Y(q')(x) = v_0$ et $\sigma'(x) = v_0 \in \{v_0\} = \phi(v_0) = \phi(Y(q')(x))$.

   (c) Soit α l'action $c\ !\ s(e)$.

   Pour toute variable x on a $\sigma'(x) = \sigma(x)$ et $Y(q) = T([b]\ \alpha)(Y(q)) \sqsubseteq Y(q')$.

   Donc $\sigma'(x) = \sigma(x) \in \phi(Y(q)(x)) \subseteq \phi(Y(q')(x))$.

□

Lemme 5.12[p.116]

La preuve est similaire à la preuve de la lemme 5.7[p.107]. Soit l'état $(\theta', \sigma', \rho')$. On montre que si le résultat est vrai dans tous les états (θ, σ, ρ) qui précèdent de manière *causale* $(\theta', \sigma', \rho')$ alors le résultat est vrai dans $(\theta', \sigma', \rho')$.

Les cas quand l'action α de la transition est $x := e$, $c\,!\,s(e)$ et $c\,?\,s(x)$ avec $c \in C^{\text{ext}}$ se traitent de manière similaire à ceux de la lemme 5.7[p.107].

Dans le cas quand α est $c\,?\,s(x)$ avec $c \in C^{\text{int}}$ on s'appuie sur le fait que le résultat est vrai dans l'état $(\theta_1, \sigma_1, \rho_1)$ t.q. $(\theta_1, \sigma_1, \rho_1) \xrightarrow{\tau} (\theta', \sigma', \rho')$ (conformément à la règle 3.16[p.49]) mais il est vrai aussi (conformément à l'hypothèse inductive) dans l'état $(\theta_2, \sigma_2, \rho_2)$ qui, conformément à la règle 3.15[p.49], a permis l'introduction du message (s, v) dans le canal c.

On observe aussi que l'effet est identique (pour les contextes de variables) avec celui de l'existence d'une transition $\theta_2 \xrightarrow{x:=e} \theta'$.

Mais dans le STE-union on a introduit ce type d'arêtes pour le même raison.

On peut maintenant effectuer un raisonnement similaire au cas $x := e$ de l'analyse intra-processus.

□

Lemme 5.14[p.118]

Soient $p_i \doteqdot (\theta, \sigma_i, \rho)$, $1 \leq i \leq 2$. $p_1 \overset{\ell}{\equiv} p_2$ implique $(\forall x)(x \in \mathit{Live}(\theta) \Rightarrow \sigma_1(x) = \sigma_2(x))$. On montre la lemme en utilisant une induction en fonction de la forme de l'étiquette a :

1. $a = \tau \in \hat{\Sigma}$. Dans ce cas la transition $p_1 \xrightarrow{\tau} q_1$ a été obtenue par l'application d'une des règles 3.12[p.49], 3.14[p.49] et 3.16[p.49] :

   (a)
   $$\frac{p_1 \doteqdot (\theta, \sigma_1, \rho) \in \hat{Q} \quad q_p \xrightarrow{[b]\,x:=e}_p q'_p \quad \sigma_1(b) = \mathbf{t} \quad \sigma_1(e) = v}{q_1 \doteqdot ([q'_p/p]\theta, [v/x]\sigma_1, \rho) \in \hat{Q} \quad p_1 \xrightarrow{\tau} q_1}\ .$$

   À partir de la définition de l'opérateur de flot du problème IP-LIVE on obtient $\mathit{vars}(b) \cup \mathit{vars}(e) \subseteq \mathit{Live}(q_p)$. Comme $(\forall x)(x \in \mathit{Live}(q_p) \Rightarrow \sigma_1(x) = \sigma_2(x))$ on obtient $\sigma_2(b) = \sigma_1(b) = \mathbf{t}$ et $\sigma_2(e) = \sigma_1(e) = v$.

   Donc cette règle peut être appliquée aussi pour p_2 avec la conclusion
   $q_2 \doteqdot ([q'_p/p]\theta, [v/x]\sigma_2, \rho) \in \hat{Q}$ et $p_2 \xrightarrow{\tau} q_2$.

   Il reste à montrer que $q_1 \overset{\ell}{\equiv} q_2$. Cela est montré en observant que $\mathit{Live}(q'_p) \subseteq \mathit{Live}(q_p) \cup \{x\}$ et que v substitue x aussi dans σ_1 et σ_2.

(b)

$$\frac{p_1 \doteqdot (\theta, \sigma_1, \rho) \in \hat{Q} \quad q_p \xrightarrow{[b]\, c!s(e)}_p q'_p \quad \sigma_1(b) = \mathbf{t} \quad \sigma_1(e) = v \quad c \in C^{\text{int}} \quad \rho(c) = w}{q_1 \doteqdot ([q'_p/p]\theta, \sigma_1, [w \cdot (s, v)/c]\rho) \in \hat{Q} \quad p_1 \xrightarrow{\tau} q_1} .$$

De manière similaire au cas précédent on montre $\sigma_2(b) = \sigma_1(b) = \mathbf{t}$ et $\sigma_2(e) = \sigma_1(e) = v$.
Donc cette règle s'applique aussi à p_2 et on obtient $p_2 \xrightarrow{\tau} q_2$ avec
$q_2 \doteqdot ([q'_p/p]\theta, \sigma_2, [w \cdot (s, v)/c]\rho) \in \hat{Q}$.
À cause du fait que $Live(q'_p) \subseteq Live(q_p)$ on obtient $q_1 \overset{\ell}{\equiv} q_2$.

(c)

$$\frac{p_1 \doteqdot (\theta, \sigma_1, \rho) \in \hat{Q} \quad q_p \xrightarrow{[b]\, c?s(x)}_p q'_p \quad \sigma_1(b) = \mathbf{t} \quad c \in C^{\text{int}} \quad \rho(c) = (s, v) \cdot w \quad v \in D \setminus \{\oslash\}}{q_1 \doteqdot ([q'_p/p]\theta, [v/x]\sigma_1, [w/c]\rho) \in \hat{Q} \quad p_1 \xrightarrow{\tau} q_1} .$$

$vars(b) \subseteq Live(q_p)$ implique $\sigma_2(b) = \mathbf{t}$ et donc cette règle s'applique aussi pour p_2 et on obtient $p_2 \xrightarrow{\tau} q_2$ avec $q_2 \doteqdot ([q'_p/p]\theta, [v/x]\sigma_2, [w/c]\rho) \in \hat{Q}$.
À cause du fait que $Live(q'_p) \subseteq Live(q_p) \cup \{x\}$ et que v substitue x aussi dans σ_1 et σ_2 on obtient $q_1 \overset{\ell}{\equiv} q_2$.

2. $a = c\ !\ s(v) \in \hat{\Sigma}$. Dans ce cas la transition $p_1 \xrightarrow{c\ !\ s(v)} q_1$ a été obtenue par l'application de la règle 3.13$_{[\text{p.49}]}$:

$$\frac{p_1 \doteqdot (\theta, \sigma_1, \rho) \in \hat{Q} \quad q_p \xrightarrow{[b]\, c!s(e)}_p q'_p \quad \sigma_1(b) = \mathbf{t} \quad \sigma_1(e) = v \quad c \in C^{\text{ext}}}{q_1 \doteqdot ([q'_p/p]\theta, \sigma_1, \rho) \in \hat{Q} \quad p_1 \xrightarrow{c!s(v)} q_1} .$$

Ceci est un cas similaire au cas de la règle 3.12$_{[\text{p.49}]}$. On obtient $\sigma_2(b) = \sigma_1(b) = \mathbf{t}$ et $\sigma_2(e) = \sigma_1(e) = v$.

Donc cette règle s'applique aussi à p_2 et on obtient $p_2 \xrightarrow{c\ !\ s(v)} q_2$ avec $q_2 \doteqdot ([q'_p/p]\theta, \sigma_2, \rho) \in \hat{Q}$.

Parce que $Live(q'_p) \subseteq Live(q_p)$ on obtient aussi $q_1 \overset{\ell}{\equiv} q_2$.

3. $a = c\ ?\ s(v) \in \hat{\Sigma}$. Dans ce cas la transition $p_1 \xrightarrow{c\ !\ s(v)} q_1$ a été obtenue par l'application de la règle 3.15$_{[\text{p.49}]}$. On effectue un raisonnement similaire celui de la règle 3.16$_{[\text{p.49}]}$.

□

Lemme 5.19[p.123]

On montre d'abord

$$(\forall p_1)(\forall p_2)(\forall a) \quad (\quad p_1 \sim_h p_2 \wedge a \in \widehat{\Sigma} \wedge \\ (\forall q_1)(p_1 \xrightarrow{a}_1 q_1 \Rightarrow (\exists q_2)(p_2 \xrightarrow{a}_2 q_2 \ \wedge \ q_1 \sim_h q_2) \ ) \tag{B.10}$$

et ensuite

$$(\forall p_1)(\forall p_2)(\forall a) \quad (\quad p_1 \sim_h p_2 \wedge a \in \widehat{\Sigma} \wedge \\ (\forall q_2)(p_2 \xrightarrow{a}_2 q_2 \Rightarrow (\exists q_1)(p_1 \xrightarrow{a}_1 q_1 \ \wedge \ q_1 \sim_h q_2) \ ). \tag{B.11}$$

On montre $B.10_{[p.187]}$ en utilisant une induction structurelle en fonction de l'étiquette a et des règles $5.2_{[p.121]}$ et $5.3_{[p.121]}$:

1. Soit $a \in \Sigma$. La transition $p_1 \xrightarrow{a}_1 q_1$ a été obtenue en utilisant la règle $5.2_{[p.121]}$

$$\frac{p_1 \doteqdot (\theta, \nu_1) \in \widehat{Q}_1 \quad \theta \xrightarrow{(a, g, (h_{i_1}, c_{i_1}), \ldots, (h_{i_j}, c_{i_j}))} \theta' \quad \nu_1(g) = \mathbf{t}}{q_1 \doteqdot (\theta', \underbrace{[c_{i_1}/h_{i_1}, \ldots, c_{i_j}/h_{i_j}]\nu_1}_{\nu_1'}) \in \widehat{Q}_1 \quad p_1 \xrightarrow{a}_1 q_1} \ .$$

On a $p_1 \sim_h p_2$ donc $p_2 = (\theta, \nu_2)$. La transition $\theta \xrightarrow{(a, g, (h_{i_1}, c_{i_1}), \ldots, (h_{i_j}, c_{i_j}))} \theta'$ existe aussi dans A' (avec peut être quelques horloges réécrites avec des horloges de H') : $\theta \xrightarrow{(a, g', (h_{j_1}, c_{j_1}), \ldots, (h_{j_k}, c_{j_k}))} \theta'$.

On s'intéresse maintenant à la valeur $\nu_2(g')$. On voudrait montrer que si $\nu_1(g) = \mathbf{t}$ alors $\nu_2(g') = \mathbf{t}$. Si $g' = g$ ($Use(g) \subseteq H'$) alors $\nu_2(g) = \nu_1(g) = \mathbf{t}$. Si $g' \neq g$ ($Use(g) \setminus H' \neq \emptyset$) alors g' a été obtenue à partir de g en réécrivant (on suppose, sans restreindre la généralité) l'horloge $h \in H$ avec $h' + c$, $h' \in H'$, $c \in \mathbf{Z}$. Donc $\nu_2(g') = \nu_2([h' + c/h]\ g)$. Mais $Use([h' + c/h]\ g) \subseteq H'$ et donc $\nu_2([h' + c/h]\ g) = \nu_1([h' + c/h]\ g) = \nu_1(g) = \mathbf{t}$. On a $\nu_1([h' + c/h]\ g) = \nu_1(g)$ parce que $\nu_1(h) = \nu_1(h') + c = \nu_1(h' + c)$.

On peut maintenant appliquer la règle $5.2_{[p.121]}$

$$\frac{p_2 \doteqdot (\theta, \nu_2) \in \widehat{Q}_2 \quad \theta \xrightarrow{(a, g', (h_{j_1}, c_{j_1}), \ldots, (h_{j_k}, c_{j_k}))} \theta' \quad \nu_2(g') = \mathbf{t}}{q_2 \doteqdot (\theta', \underbrace{[c_{j_1}/h_{j_1}, \ldots, c_{j_k}/h_{j_k}]\nu_2}_{\nu_2'}) \in \widehat{Q}_2 \quad p_2 \xrightarrow{a}_2 q_2} \ .$$

Il faut montrer aussi que $q_1 \sim_h q_2$ et cela équivaut à montrer que $(\forall h')(h' \in H' \Rightarrow \nu'_1(h') = \nu'_2(h'))$.

Observons-nous que les horloges de H' qui ne sont pas armées dans

$\theta \xrightarrow{(a, g, (h_{i_1}, c_{i_1}), \ldots, (h_{i_j}, c_{i_j}))} \theta'$ ne sont sûrement pas armées dans

$\theta \xrightarrow{(a, g', (h_{j_1}, c_{j_1}), \ldots, (h_{j_k}, c_{j_k}))} \theta'$.

En tenant compte aussi du fait que $(\forall h')(h' \in H' \Rightarrow \nu_1(h') = \nu_2(h'))$ il résulte que $(\forall h')(h' \in H' \setminus \{h_{i_1}, \ldots, h_{i_j}\} \Rightarrow \nu'_1(h') = \nu'_2(h'))$.

On a $\{h_{j_1}, \ldots, h_{j_k}\} = H' \cap \{h_{i_1}, \ldots, h_{i_j}\}$ et donc les horloges de $\{h_{j_1}, \ldots, h_{j_k}\}$ sont armées dans les transitions $\theta \xrightarrow{(a, g, (h_{i_1}, c_{i_1}), \ldots, (h_{i_j}, c_{i_j}))} \theta'$ et

$\theta \xrightarrow{(a, g', (h_{j_1}, c_{j_1}), \ldots, (h_{j_k}, c_{j_k}))} \theta'$ avec la même valeur. Donc $(\forall h')(h' \in \{h_{j_1}, \ldots, h_{j_k}\} \Rightarrow \nu'_1(h') = \nu'_2(h'))$.

Il faut chercher maintenant le cas des horloges appartenant à l'ensemble $\{h_{i_1}, \ldots, h_{i_j}\} \setminus \{h_{j_1}, \ldots, h_{j_k}\}$ qui, dans $\theta \xrightarrow{(a, g', (h_{j_1}, c_{j_1}), \ldots, (h_{j_k}, c_{j_k}))} \theta'$ sont réécrites par des horloges de H'. Mais à cause du fait que $\{h_{j_1}, \ldots, h_{j_k}\} = H' \cap \{h_{i_1}, \ldots, h_{i_j}\}$ ces horloges n'appartiennent pas à H' et donc ils n'influencent pas l'égalité de ν'_1 et ν'_2 sur H'.

2. Soit $a = tick$. La transition $p_1 \xrightarrow{tick}_1 q_1$ a été obtenue en utilisant la règle 5.3[p.121]

$$\frac{p_1 \doteqdot (\theta, \nu_1) \in \widehat{Q}_1}{q_1 \doteqdot (\theta, \nu_1 + \epsilon) \in \widehat{Q}_1 \quad p_1 \xrightarrow{tick}_1 q_1} \ .$$

La même règle peut être appliquée dans $\widehat{A'_\epsilon}$:

$$\frac{p_2 \doteqdot (\theta, \nu_2) \in \widehat{Q}_2}{q_2 \doteqdot (\theta, \nu_2 + \epsilon) \in \widehat{Q}_2 \quad p_2 \xrightarrow{tick}_2 q_2} \ .$$

Il faut montrer aussi que $q_1 \sim_h q_2$ et cela équivaut à montrer que $(\forall h')(h' \in H' \Rightarrow (\nu_1 + \epsilon)(h') = (\nu_2 + \epsilon)(h'))$.

Mais $(\nu_1 + \epsilon)(h') = \nu_1(h') + \epsilon = \nu_2(h') + \epsilon = (\nu_2 + \epsilon)(h')$.

L'équation $B.11_{[p.187]}$ se montre de manière similaire avec $B.10_{[p.187]}$.

□

Lemme $5.21_{[p.131]}$

Soient $a \in \Sigma$, $p_1 \sim_o p_2$ et $q_1 \in Q_1$ t.q. $p_1 \xrightarrow{a}_1 q_1$.

Parce que $(\theta, \sigma_1, \rho_1) = p_1 \sim_o p_2$ on a $p_2 = (\theta, \sigma_2, \rho_2)$ et $(\theta, \sigma_1) \overset{v}{\equiv} (\theta, \sigma_2)$ et $\rho_1 \overset{q}{\equiv} \rho_2$.

On montre l'existence de $q_2 \in Q_2$ t.q. $p_2 \xrightarrow{a}_2 q_2$ et $q_1 \sim_o q_2$ en utilisant une induction en fonction de la forme de l'étiquette a et les règles $3.11_{[p.49]} \div 3.16_{[p.49]}$ et $5.10_{[p.128]}$.

1. Soit $a = \tau \in \Sigma$. La transition $p_1 \xrightarrow{\tau}_1 q_1$ a été obtenue par l'utilisation d'une des règles $3.12_{[p.49]}$, $3.14_{[p.49]}$, $3.16_{[p.49]}$:

   $(3.12_{[p.49]})$. La règle appliquée est :

$$\frac{p_1 \doteqdot (\theta, \sigma_1, \rho_1) \in Q_1 \quad q_p \overset{[b]\ x:=e}{\longrightarrow}_p q'_p \quad \sigma_1(b) = \mathbf{t} \quad \sigma_1(e) = v}{q_1 \doteqdot (\theta', [v/x]\sigma_1, \rho_1) \in Q_1 \quad p_1 \xrightarrow{\tau}_1 q_1} .$$

   On a deux cas :

   (a) Si $x \in Rlv(q'_p)$ alors la transition $q_p \overset{[b]\ x:=e}{\longrightarrow}_p q'_p$ est conservée par $\backslash_o$ conformément à la règle $5.4_{[p.127]}$. Parce que $(\theta, \sigma_1) \overset{v}{\equiv} (\theta, \sigma_2)$ et $vars(b) \cup vars(e) \subseteq Rlv(\theta)$ on a $\sigma_2(b) = \sigma_1(b) = \mathbf{t}$ et $\sigma_2(e) = \sigma_1(e)$.

   Donc la règle $3.12_{[p.49]}$ peut être appliquée pour $\widehat{SP \backslash_o \Upsilon^s}(L)$:

$$\frac{p_2 \doteqdot (\theta, \sigma_2, \rho_2) \in Q_2 \quad q_p \overset{[b]\ x:=e}{\nrightarrow_o}_p q'_p \quad \sigma_2(b) = \mathbf{t} \quad \sigma_2(e) = v}{q_2 \doteqdot (\theta', [v/x]\sigma_2, \rho_2) \in Q_2 \quad p_2 \xrightarrow{\tau}_2 q_2} .$$

   On se trouve dans les conditions de la lemme $5.20_{[p.131]}$ et donc $(\theta', \sigma_1[v/x]) \overset{v}{\equiv} (\theta', \sigma_2[v/x])$ et on obtient que $q_1 \sim_o q_2$ et dans ce cas la lemme est montrée.

   (b) Si $x \notin Rlv(q'_p)$ alors la transition $q_p \overset{[b]\ x:=e}{\longrightarrow}_p q'_p$ est changée dans $q_p \overset{[b]\ \tau}{\nrightarrow_o}_p q'_p$.

   Comme ci-dessus, parce que $(\theta, \sigma_1) \overset{v}{\equiv} (\theta, \sigma_2)$ et $vars(b) \subseteq Rlv(\theta)$ on a $\sigma_2(b) = \sigma_1(b) = \mathbf{t}$ et la règle $3.19_{[p.50]}$ peut être appliquée pour $\widehat{SP \backslash_o \Upsilon^s}(L)$:

$$\frac{p_2 \doteqdot (\theta, \sigma_2, \rho_2) \in Q_2 \quad q_p \overset{[b]\ \tau}{\nrightarrow_o}_p q'_p \quad \sigma_2(b) = \mathbf{t}}{q_2 \doteqdot (\theta', \sigma_2, \rho_2) \in Q_2 \quad p_2 \xrightarrow{\tau}_2 q_2} .$$

En plus, parce que $x \notin Rlv(\theta')$ on a $Rlv(\theta') \subseteq Rlv(\theta)$ et donc $(\theta', [v/x]\sigma_1) \overset{\text{v}}{\equiv} (\theta', \sigma_2)$ et $q_1 \sim_{\text{o}} q_2$.

($3.14_{[\text{p.49}]}$). La règle appliquée est :

$$\frac{p_1 \doteqdot (\theta, \sigma_1, \rho_1) \in Q_1 \quad \sigma_1(e) = v \quad c \in C^{\text{int}} \quad q_p \overset{[b]\ c!s(e)}{\longrightarrow}_p q'_p \quad \sigma_1(b) = \mathbf{t} \quad \rho_1(c) = w_1}{q_1 \doteqdot (\theta', \sigma_1, [w_1.s(v)/c]\rho_1) \in Q_1 \quad p_1 \xrightarrow{\tau}_1 q_1} \ .$$

À cause du fait que $c \in C^{\text{int}}$ le prédicat $Use(c\ !\ s)$ se réduit à $vars(e) \subseteq Rlv(q_p)$.

On a deux cas :

(a) Si $Use(c\ !\ s)$ est vrai alors la transition $q_p \overset{[b]\ c\ !\ s(e)}{\longrightarrow}_p q'_p$ est est conservée par $\setminus_{\text{o}}$ conformément à la règle $5.8_{[\text{p.128}]}$ et comme on a $vars(b) \subseteq Rlv(q_p)$ on obtient $vars(b) \cup vars(e) \subseteq Rlv(\theta)$ et donc $\sigma_2(b) = \sigma_1(b) = \mathbf{t}$ et $\sigma_2(e) = \sigma_1(e)$.
Donc la règle $3.14_{[\text{p.49}]}$ pourrait être appliquée dans $\widehat{SP\ \setminus_{\text{o}} \Upsilon^s}(L)$:

$$\frac{p_2 \doteqdot (\theta, \sigma_2, \rho_2) \in Q_2 \quad \sigma_2(e) = v \quad c \in C^{\text{int}} \quad q_p \overset{[b]\ c!s(e)}{\underset{\setminus_{\text{o}}}{\longrightarrow}}_p q'_p \quad \sigma_2(b) = \mathbf{t} \quad \rho_2(c) = w_2}{q_2 \doteqdot (\theta', \sigma_2, [w_2.s(v)/c]\rho_2) \in Q_2 \quad p_2 \xrightarrow{\tau}_2 q_2} \ .$$

On se trouve dans les conditions de la lemme $5.20_{[\text{p.131}]}$ et donc $(\theta', \sigma_1) \overset{\text{v}}{\equiv} (\theta', \sigma_2)$ et parce que $\rho_1 \overset{\text{q}}{\equiv} \rho_2$ on a $\rho_1[w_1.s(v)/c] \overset{\text{q}}{\equiv} \rho_2[w_2.s(v)/c]$ et donc $q_1 \sim_{\text{o}} q_2$.

(b) Si $Use(c\ !\ s)$ est faux alors la transition $q_p \overset{[b]\ c!s(e)}{\longrightarrow}_p q'_p$ est transformée par $\setminus_{\text{o}}$ dans $q_p \overset{[b]\ c!s(\oslash}{\underset{\setminus_{\text{o}}}{\longrightarrow}}_p q'_p$ et maintenant on peut appliquer la règle $3.20_{[\text{p.50}]}$ dans $\widehat{SP\ \setminus_{\text{o}} \Upsilon^s}(L)$:

$$\frac{p_2 \doteqdot (\theta, \sigma_2, \rho_2) \in Q_2 \quad \sigma_2(b) = \mathbf{t} \quad c \in C^{\text{int}} \quad q_p \overset{[b]\ c!s(\oslash)}{\underset{\setminus_{\text{o}}}{\longrightarrow}}_p q'_p \quad \rho_2(c) = w_2}{q_2 \doteqdot (\theta', \sigma_2, [w_2.s(\oslash)/c]\rho_2) \in Q_2 \quad p_2 \xrightarrow{\tau}_2 q_2} \ .$$

On a aussi $q_1 \sim_{\text{o}} q_2$ (parce que les prémisses de la lemme $5.20_{[\text{p.131}]}$ sont accomplies et $[w_1.s(v)/c]\rho_1 \overset{\text{q}}{\equiv} [w_2.s(\oslash)/c]\rho_2$).

($3.16_{[\text{p.49}]}$). La règle appliquée est :

$$\frac{p_1 \doteqdot (\theta, \sigma_1, \rho_1) \in Q_1 \quad \sigma_1(b) = \mathbf{t} \quad c \in C^{\text{int}} \quad q_p \overset{[b]\ c?s(x)}{\longrightarrow}_p q'_p \quad v_1 \in D \setminus \{\oslash\} \quad \rho_1(c) = s(v_1).w_1}{q_1 \doteqdot (\theta', [v_1/x]\sigma_1, [w_1/c]\rho_1) \in Q_1 \quad p_1 \xrightarrow{\tau}_1 q_1} \ .$$

On a deux cas :

(a) Si $x \in Rlv(q'_p)$ alors la transition $q_p \overset{[b]\ c?s(x)}{\longrightarrow}_p q'_p$ est conservée par $\setminus_{\text{o}}$ conformément à la règle $5.6_{[\text{p.127}]}$. Parce que $(\theta, \sigma_1) \overset{\text{v}}{\equiv} (\theta, \sigma_2)$ et $vars(b) \subseteq Rlv(\theta)$ on obtient $\sigma_2(b) = \sigma_1(b) = \mathbf{t}$ et donc la règle $3.16_{[\text{p.49}]}$ peut aussi être appliquée dans $SP\widehat{\setminus_{\text{o}} \Upsilon^s}(L)$:

$$\frac{p_2 \doteqdot (\theta, \sigma_2, \rho_2) \in Q_2 \quad \sigma_2(b) = \mathbf{t} \quad c \in C^{\text{int}} \quad q_p \overset{[b]\ c?s(x)}{\underset{\setminus_{\text{o}}}{\longrightarrow}}_p q'_p \quad v_2 \in D \setminus \{\oslash\} \quad \rho_2(c) = s(v_2).w_2}{q_2 \doteqdot (\theta', [v_2/x]\sigma_2, [w_2/c]\rho_2) \in Q_2 \quad p_2 \overset{\tau}{\longrightarrow}_2 q_2}\ .$$

Parce que $\rho_1 \overset{\text{q}}{\equiv} \rho_2$ et $v_1, v_2 \in D \setminus \{\oslash\}$ on obtient $v_1 = v_2$ et maintenant les prémisses de la lemme $5.20_{[\text{p.131}]}$ sont vraies et donc $(\theta', \sigma_1[v_1/x]) \overset{\text{v}}{\equiv} (\theta', \sigma_2[v_2/x])$. À cause du fait que $\rho_1 \overset{\text{q}}{\equiv} \rho_2$ on obtient $\rho_1[w_1/c] \overset{\text{q}}{\equiv} \rho_2[w_2/c]$ et donc $q_1 \sim_{\text{o}} q_2$.

(b) Si $x \notin Rlv(q'_p)$ alors la transition $q_p \overset{[b]\ c?s(x)}{\longrightarrow}_p q'_p$ est changée dans $q_p \overset{[b]\ c?s(\Omega)}{\underset{\setminus_{\text{o}}}{\longrightarrow}}_p q'_p$. Comme ci-dessus, les variables de $vars(b)$ appartiennent à $Rlv(\theta)$ et donc $\sigma_2(b) = \sigma_1(b) = \mathbf{t}$ et la règle $3.18_{[\text{p.50}]}$ peut être appliquée dans $SP\widehat{\setminus_{\text{o}} \Upsilon^s}(L)$:

$$\frac{p_2 \doteqdot (\theta, \sigma_2, \rho_2) \in Q_2 \quad \sigma_2(b) = \mathbf{t} \quad c \in C^{\text{int}} \quad q_p \overset{[b]\ c?s(\top)}{\underset{\setminus_{\text{o}}}{\longrightarrow}}_p q'_p \quad v_2 \in D \quad \rho_2(c) = s(v_2).w_2}{q_2 \doteqdot (\theta', \sigma_2, [w_2/c]\rho_2) \in Q_2 \quad p_2 \overset{\tau}{\longrightarrow}_2 q_2}\ .$$

À cause du fait que $x \notin Rlv(\theta')$ on a $Rlv(\theta') \subseteq Rlv(\theta)$ et donc $(\theta', [v_1/x]\sigma_1) \overset{\text{v}}{\equiv} (\theta', \sigma_2)$ et du au fait que $\rho_1 \overset{\text{q}}{\equiv} \rho_2$ on obtient $[w_1/c]\rho_1 \overset{\text{q}}{\equiv} [w_2/c]\rho_2$ et donc $q_1 \sim_{\text{o}} q_2$.

2. Soit $a = c?s(v) \in \Sigma$. La transition $p_1 \overset{c?s(v)}{\longrightarrow}_1 q_1$ a été obtenue par l'utilisation de la règle $3.15_{[\text{p.49}]}$:

$$\frac{p_1 \doteqdot (\theta, \sigma_1, \rho_1) \in Q_1 \quad q_p \overset{[b]\ c?s(x)}{\longrightarrow}_p q'_p \quad \sigma_1(b) = \mathbf{t} \quad c \in C^{\text{ext}} \quad v \in D \setminus \{\oslash\}}{q_1 \doteqdot (\theta', [v/x]\sigma_1, \rho_1) \in Q_1 \quad p_1 \overset{c?s(v)}{\longrightarrow}_1 q_1}\ .$$

Ce cas est similaire à un cas précédent (celui quand la transition $p_1 \overset{\tau}{\longrightarrow}_1 q_1$ a été obtenue par l'utilisation de la règle $3.16_{[\text{p.49}]}$). Dans ce cas, quand $x \notin Rlv(q'p)$, on utilise la règle $3.17_{[\text{p.50}]}$ au lieu de la règle $3.18_{[\text{p.50}]}$ (car maintenant on a $c \in C^{\text{ext}}$).

3. Soit $a = c\,!\,s(v) \in \Sigma$. La transition $p_1 \xrightarrow{c!s(v)}_1 q_1$ a été obtenue en utilisant la règle $3.13_{[\text{p.49}]}$:

$$\frac{p_1 \doteqdot (\theta, \sigma_1, \rho_1) \in Q_1 \quad q_p \xrightarrow{[b]\ c!s(e)}_p q'_p \quad \sigma_1(b) = \mathbf{t} \quad \sigma_1(e) = v \quad c \in C^{\text{ext}}}{q_1 \doteqdot (\theta', \sigma_1, \rho_1) \in Q_1 \quad p_1 \xrightarrow{c!s(v)}_1 q_1} \ .$$

Ce cas est partiellement similaire à un cas précédent (celui dont la transition $p_1 \xrightarrow{\tau}_1 q_1$ a été obtenue en utilisant la règle $3.14_{[\text{p.49}]}$ et $Use(c!s)$ est vrai).

Si $Use(c!s)$ est faux et parce que $c \in C^{\text{ext}}$ on obtient $c\,!\,s(\oslash) \notin \Upsilon^s(L)$ et $(\forall \ell)(\ell \in L \wedge c\,!\,s(\ell) \notin \Upsilon^s(L))$. Donc l'utilisation de la règle $5.10_{[\text{p.128}]}$ est *obligatoire* et la transition $p_1 \xrightarrow{c!s(v)}_1 q_1$ devient $p_1 \xrightarrow{c!s(\oslash)}_1 q_1$.

Conformément à la règle $5.9_{[\text{p.128}]}$ la transition $q_p \xrightarrow{[b]\ c!s(e)}_p q'_p$ est transformée dans $q_p \xrightarrow[\setminus_{\text{o}}]{[b]\ c!s(\oslash)}_p q'_p$. Maintenant la règle $3.19_{[\text{p.50}]}$ peut être appliquée dans $SP\ \widehat{\setminus_{\text{o}} \Upsilon^s}(L)$:

$$\frac{p_2 \doteqdot (\theta, \sigma_2, \rho_2) \in Q_2 \quad q_p \xrightarrow[\setminus_{\text{o}}]{[b]\ c!s(\oslash)}_p q'_p \quad \sigma_2(b) = \mathbf{t} \quad c \in C^{\text{ext}}}{q_2 \doteqdot (\theta', \sigma_2, \rho_2) \in Q_2 \quad p_2 \xrightarrow{c\ !\ s(\oslash)}_2 q_2} \ .$$

On obtient aussi $q_1 \ \sim_{\text{o}} \ q_2$.

4. Soit $a = c\,!\,s(\oslash) \in \Sigma$. La transition $p_1 \xrightarrow{c!s(\oslash)}_1 q_1$ a été obtenue en utilisant les règles $3.13_{[\text{p.49}]}$ et $5.10_{[\text{p.128}]}$:

$$\frac{p_1 \doteqdot (\theta, \sigma_1, \rho_1) \in Q_1 \quad q_p \xrightarrow{[b]\ c!s(e)}_p q'_p \quad \sigma_1(b) = \mathbf{t} \quad \sigma_1(e) = v \quad c \in C^{\text{ext}}}{q_1 \doteqdot (\theta', \sigma_1, \rho_1) \in Q_1 \quad p_1 \xrightarrow{c!s(v)}_1 q_1}$$

$$\frac{p_1 \xrightarrow{c!s(v)}_1 q_1 \quad c!s(\oslash) \ \notin \ \Upsilon^s(L)}{p_1 \xrightarrow{c!s(\oslash)}_1 q_1} \ .$$

Par transitivité on obtient la règle :

$$\frac{p_1 \doteqdot (\theta, \sigma_1, \rho_1) \in Q_1 \quad q_p \xrightarrow{[b]\ c!s(e)}_p q'_p \quad \sigma_1(b) = \mathbf{t} \quad \sigma_1(e) = v \quad c \in C^{\text{ext}} \quad c!s(\oslash) \ \notin \ \Upsilon^s(L)}{p_1 \xrightarrow{c!s(\oslash)}_1 q_1} \ .$$

On utilise à nouveau le raisonnement du cas dont $a = c\,!\,s(v) \in \Sigma$ et $Use(c!s)$ est faux.

□

Lemme 5.22[p.131]

Soient $a \in \Sigma$, $p_1 \sim_{\mathrm{o}} p_2$ et $q_2 \in Q_2$ t.q. $p_2 \xrightarrow{a}_2 q_2$.

À cause du fait que $(\theta, \sigma_2, \rho_2) = p_2 \sim_{\mathrm{o}} p_1$ on a $p_1 = (\theta, \sigma_1, \rho_1)$ et $(\theta, \sigma_1) \overset{\mathrm{v}}{\equiv} (\theta, \sigma_2)$ et $\rho_1 \overset{\mathrm{q}}{\equiv} \rho_2$.

On montre l'existence de $q_1 \in Q_1$ t.q. $p_1 \xrightarrow{a}_1 q_1$ et $q_1 \sim_{\mathrm{o}} q_2$ en utilisant une induction sur la forme de l'étiquette a et les règles 3.11[p.49] $\div$ 3.21[p.50] et 5.4[p.127] $\div$ 5.9[p.128].

On raisonne seulement dans le cas des règles 3.17[p.50] $\div$ 3.21[p.50] parce que dans les cas des règles 3.11[p.49] $\div$ 3.16[p.49] on peut utiliser les implications inverses de la lemme 5.21[p.131].

1. Soit $a = \tau \in \Sigma$. La transition $p_2 \xrightarrow{\tau}_2 q_2$ a été obtenue en utilisant une des règles 3.12[p.49], 3.14[p.49], 3.16[p.49], 3.18[p.50], 3.20[p.50], 3.21[p.50].

   (3.18[p.50]). La règle appliquée est :

$$\frac{p_2 \doteqdot (\theta, \sigma_2, \rho_2) \in Q_2 \quad \sigma_2(b) = \mathbf{t} \quad c \in C^{\mathrm{int}} \quad q_p \overset{[b]\ c\ ?\ s(\Omega)}{\xrightarrow[\circ]{}_p} q'_p \quad v_2 \in D \quad \rho(c) = s(v_2).w_2}{q_2 \doteqdot (\theta', \sigma_2, [w_2/c]\rho_2) \in Q_2 \quad p_2 \xrightarrow{\tau}_2 q_2} .$$

   La transition $q_p \overset{[b]\ c\ ?\ s(\Omega)}{\xrightarrow[\circ]{}_p} q'_p$ a été obtenue en utilisant la règle 5.7[p.128]. Donc on peut appliquer la règle 3.16[p.49] dans $\widehat{SP} \downarrow \Upsilon^s(L)$:

$$\frac{p_1 \doteqdot (\theta, \sigma_1, \rho_1) \in Q_1 \quad \sigma_1(b) = \mathbf{t} \quad c \in C^{\mathrm{int}} \quad q_p \overset{[b]\ c?s(x)}{\longrightarrow}_p q'_p \quad v_1 \in D \setminus \{\oslash\} \quad \rho_1(c) = s(v_1).w_1}{q_1 \doteqdot (\theta', [v_1/x]\sigma_1, [w_1/c]\rho_1) \in Q_1 \quad p_1 \xrightarrow{\tau}_1 q_1} .$$

   Mais parce que la règle 5.7[p.128] a été appliquée, $x \notin Rlv(\theta')$ et donc $Rlv(\theta') \subseteq Rlv(\theta)$ et donc on a $(\theta', \sigma_2) \overset{\mathrm{v}}{\equiv} (\theta', [v_1/x]\sigma_1)$. À cause du fait que $\rho_1 \overset{\mathrm{q}}{\equiv} \rho_2$ on a aussi $[w_1/c]\rho_1 \overset{\mathrm{q}}{\equiv} [w_2/c]\rho_2$ et donc $q_1 \sim_{\mathrm{o}} q_2$.

   (3.20[p.50]). La règle appliquée est :

$$\frac{p_2 \doteqdot (\theta, \sigma_2, \rho_2) \in Q_2 \quad \sigma_2(b) = \mathbf{t} \quad c \in C^{\mathrm{int}} \quad q_p \overset{[b]\ c!s(\oslash)}{\xrightarrow[\circ]{}_p} q'_p \quad \rho_2(c) = w_2}{q_2 \doteqdot (\theta', \sigma_2, [w_2.s(\oslash)/c]\rho_2) \in Q_2 \quad p_2 \xrightarrow{\tau}_2 q_2} .$$

La transition $q_p \overset{[b]\ c!s(\oslash)}{\underset{\backslash o}{\longrightarrow}}_p q'_p$ a été obtenue en utilisant la règle $5.9_{[p.128]}$. Maintenant on peut appliquer la règle $3.14_{[p.49]}$ dans le cas de $\widehat{SP} \downarrow \Upsilon^s(L)$:

$$\frac{p_1 \doteqdot (\theta, \sigma_1, \rho_1) \in Q_1 \quad \sigma_1(e) = v_1 \quad c \in C^{\text{int}} \quad q_p \overset{[b]\ c!s(e)}{\longrightarrow}_p q'_p \quad \sigma_1(b) = \mathbf{t} \quad \rho_1(c) = w_1}{q_1 \doteqdot (\theta', \sigma_1, [w_1.s(v_1)/c]\rho_1) \in Q_1 \quad p_1 \xrightarrow{\tau}_1 q_1} \ .$$

On a $[w_1.s(v_1)/c]\rho_1 \overset{\text{q}}{\equiv} [w_2.s(\oslash)/c]\rho_2$ et donc on obtient $q_1 \sim_{\text{o}} q_2$.

$(3.21_{[p.50]})$. La règle appliquée est :

$$\frac{p_2 \doteqdot (\theta, \sigma_2, \rho_2) \in Q_2 \quad q_p \overset{[b]\ \tau}{\underset{\backslash o}{\longrightarrow}}_p q'_p \quad \sigma_2(b) = \mathbf{t}}{q_2 \doteqdot (\theta', \sigma_2, \rho_2) \in Q_2 \quad p_2 \xrightarrow{\tau}_2 q_2} \ .$$

La transition $q_p \overset{[b]\ \tau}{\underset{\backslash o}{\longrightarrow}}_p q'_p$ a été obtenue en utilisant la règle $5.5_{[p.127]}$. Maintenant on peut appliquer la règle $3.12_{[p.49]}$ dans le cas de $\widehat{SP} \downarrow \Upsilon^s(L)$:

$$\frac{p_1 \doteqdot (\theta, \sigma_1, \rho_1) \in Q_1 \quad q_p \overset{[b]\ x:=e}{\longrightarrow}_p q'_p \quad \sigma_1(b) = \mathbf{t} \quad \sigma_1(e) = v_1}{q_1 \doteqdot (\theta', [v_1/x]\sigma_1, \rho_1) \in Q_1 \quad p_1 \xrightarrow{\tau}_1 q_1} \ .$$

Mais parce que la règle $5.5_{[p.127]}$ a été appliqué, $x \notin Rlv(\theta')$ et donc on a $(\theta', \sigma_2) \overset{\text{v}}{\equiv} (\theta', [v_1/x]\sigma_1)$ et donc $q_1 \sim_{\text{o}} q_2$.

2. Soit $a = c\ ?\ s(v) \in \Sigma$. La transition $p_2 \xrightarrow{c?s(v)}_2 q_2$ a été obtenue en utilisant une des règles $3.15_{[p.49]}$, $3.17_{[p.50]}$.

   $(3.17_{[p.50]})$. La règle appliquée est :

$$\frac{p_2 \doteqdot (\theta, \sigma_2, \rho_2) \in Q_2 \quad q_p \overset{[b]\ c?s(\Omega)}{\underset{\backslash o}{\longrightarrow}}_p q'_p \quad \sigma_2(b) = \mathbf{t} \quad c \in C^{\text{ext}} \quad v \in D}{q_2 \doteqdot (\theta', [v/x]\sigma_2, \rho_2) \in Q_2 \quad p_2 \xrightarrow{c?s(v)}_2 q_2} \ .$$

La transition $q_p \overset{[b]\ \tau}{\underset{\backslash o}{\longrightarrow}}_p q'_p$ a été obtenue en utilisant la règle $5.7_{[p.128]}$. Maintenant on peut appliquer la règle $3.15_{[p.49]}$ dans le cas de $\widehat{SP} \downarrow \Upsilon^s(L)$:

$$\frac{p_1 \doteqdot (\theta, \sigma_1, \rho_1) \in Q_1 \quad q_p \overset{[b]\ c\ ?\ s(x)}{\longrightarrow}_p q'_p \quad \sigma_1(b) = \mathbf{t} \quad c \in C^{\text{ext}} \quad v \in D}{q_1 \doteqdot (\theta', [v/x]\sigma_1, \rho_1) \in Q_1 \quad p_1 \xrightarrow{c?s(v)}_1 q_1} \ .$$

À cause de l'application de la règle $5.7_{[p.128]}$, $x \notin Rlv(\theta')$ et donc on a $(\theta', [v/x]\sigma_2) \overset{v}{\equiv} (\theta', [v/x]\sigma_1)$ et donc $q_1 \sim_o q_2$.

3. Soit $a = c!s(v) \in \Sigma$. La transition $p_2 \xrightarrow{c!s(v)}_2 q_2$ a été obtenue en utilisant la règle $3.13_{[p.49]}$.

4. Soit $a = c\,!\,s(\oslash) \in \Sigma$. La transition $p_2 \xrightarrow{c!s(\oslash)}_2 q_2$ a été obtenue en utilisant la règle $3.19_{[p.50]}$:

$$\frac{p_2 \doteqdot (\theta, \sigma_2, \rho_2) \in Q_2 \quad q_p \overset{[b]}{\xrightarrow[\setminus o]{}}{}_p\ q'_p \quad \sigma_2(b) = \mathbf{t} \quad c \in C^{\text{ext}}}{q_2 \doteqdot (\theta', \sigma_2, \rho_2) \in Q_2 \quad p_2 \xrightarrow{c!s(\perp)}_2 q_2} .$$

La transition $q_p \overset{[b]}{\xrightarrow[\setminus o]{}}{}_p\ q'_p$ a été obtenue en utilisant la règle $5.9_{[p.128]}$.

Maintenant on peut appliquer la règle combinée $3.13_{[p.49]}$ et $5.10_{[p.128]}$ dans le cas de $\widehat{SP} \downarrow \Upsilon^s(L)$:

$$\frac{\begin{array}{l} p_1 \doteqdot (\theta, \sigma_1, \rho_1) \in Q_1 \quad q_p \overset{[b]}{\xrightarrow{c\ !\ s(e)}}{}_p\ q'_p \quad \sigma_1(b) = \mathbf{t} \quad \sigma_1(e) = v_1 \quad c \in C^{\text{ext}} \\ c!s(\oslash) \notin \Upsilon^s(L) \wedge (\forall \ell)(\ell \in L \wedge c\,!\,s(\ell) \notin \Upsilon^s(L)) \end{array}}{q_1 \doteqdot (\theta', \sigma_1, \rho_1) \in Q_1 \quad p_1 \xrightarrow{c!s(\oslash)}_1 q_1} .$$

Les prémisses de la lemme $5.20_{[p.131]}$ sont vraies et donc on obtient $q_1 \sim_o q_2$.

□

Théorème $5.6_{[p.134]}$

Cette preuve est similaire à la preuve du théorème $4.1_{[p.90]}$ On montre que l'égalité entre les états est aussi une bisimulation et donc c'est la bisimulation forte.

L'application des règles $3.35_{[p.82]}$ pour $\prod(\widehat{SP}, TP, \Sigma_f)$ et $3.35_{[p.82]}$ et $5.11_{[p.134]}$ pour $\prod(\widehat{SP \setminus_c \Sigma_f}, TP, \Sigma_f)$ conduit à l'égalité entre les états initiaux de $\prod(\widehat{SP}, TP, \Sigma_f)$ et $\prod(\widehat{SP \setminus_c \Sigma_f}, TP, \Sigma_f)$.

Dans la suite on montre qu'une transition existe dans $\prod(\widehat{SP}, TP, \Sigma_f)$ ssi elle existe dans $\prod(\widehat{SP \setminus_c \Sigma_f}, TP, \Sigma_f)$:

1. Supposons-nous qu'une transition de $\prod(\widehat{SP}, TP, \Sigma_f)$ a été obtenue en appliquant la règle $3.36_{[p.82]}$.

Cela a été possible à cause de l'application d'une des règles $3.12_{[p.49]}$, $3.14_{[p.49]}$ ou $3.16_{[p.49]}$ dans $\widehat{SP}$.

Mais, à cause de la lemme $5.7_{[p.107]}$, les mêmes règles peuvent être appliquées dans $\widehat{SP \setminus_c \Sigma_f}$ et donc l'application de la règle $3.36_{[p.82]}$ est possible dans $\prod(\widehat{SP \setminus_c \Sigma_f}, TP, \Sigma_f)$.

Les implications inverses sont aussi vraies.

2. Supposons-nous qu'une transition de $\prod(\widehat{SP}, TP, \Sigma_f)$ a été obtenue en appliquant les règles $3.37_{[p.82]}$ ou $3.38_{[p.82]}$.

   Cela a été possible à cause de l'application de la règle $3.15_{[p.49]}$ dans $\widehat{SP}$.

   Mais, à cause de la lemme $5.7_{[p.107]}$, la même règle peut être appliquée dans $\widehat{SP \setminus_c \Sigma_f}$ et donc l'application des règles $3.37_{[p.82]}$ ou $3.38_{[p.82]}$ est possible dans $\prod(\widehat{SP \setminus_c \Sigma_f}, TP, \Sigma_f)$.

   Observons-nous que la prémisse $v \in D \setminus \{\oslash\}$ de la règle $3.15_{[p.49]}$ est *filtrée* par les prémisses des règles $3.37_{[p.82]}$ et $3.38_{[p.82]}$ pour obtenir $v \in \phi(\ell)$ et donc les implications inverses sont aussi vraies.

3. Supposons-nous qu'une transition de $\prod(\widehat{SP}, TP, \Sigma_f)$ a été obtenue en appliquant les règles $3.39_{[p.82]}$ ou $3.40_{[p.82]}$.

   Cela a été possible à cause de l'application de la règle $3.13_{[p.49]}$ dans $\widehat{SP}$.

   Mais, à cause de la lemme $5.7_{[p.107]}$, la même règle peut être appliquée dans $\widehat{SP \setminus_c \Sigma_f}$ et donc l'application des règles $3.39_{[p.82]}$ ou $3.40_{[p.82]}$ est possible dans $\prod(\widehat{SP \setminus_c \Sigma_f}, TP, \Sigma_f)$. Les implications inverses sont aussi vraies.

□

Bibliographie

[ABB+01] Tobias Amnell, Gerd Behrmann, Johan Bengtsson, Pedro R. D'Argenio, Alexandre David, Ansgar Fehnker, Thomas Hune, Bertrand Jeannet, Kim G. Larsen, M. Oliver Möller, Paul Pettersson, Carsten Weise, and Wang Yi. UPPAAL — now, next, and future. In *[CJRR01]*, page 99, 2001.

[ABD02] Paul Ammann, Paul E. Black, and Wei Ding. Model checkers in software testing. NIST-IR 6777, National Institute of Standards and Technology, 2002.

[ACD93] R. Alur, C. Courcoubetis, and D.L. Dill. Model Checking in Dense Real Time. *Information and Computation*, 104(1), 1993.

[ACY95] Rajeev Alur, Costas Courcoubetis, and Mihalis Yannakakis. Distinguishing tests for nondeterministic and probabilistic machines. In *Proceedings of the Twenty-Seventh Annual ACM Symposium on the Theory of Computing*, pages 363–372, Las Vegas, Nevada, May 1995.

[AD94] Rajeev Alur and David L. Dill. A theory of timed automata. *Theoretical Computer Science*, (126):183–235, 1994.

[Ann01] A. Annichini. *Vérification d'automates étendus : algorithmes d'analyse symbolique et mise en oeuvre.* Thèse de doctorat, Université Joseph Fourier, Grenoble I, 2001.

[Apt99] Krzysztof R. Apt. The essence of constraint propagation. *Theoretical Computer Science*, 221(1–2):179–210, June 1999.

[ASU89] Alfred Aho, Ravi Sethi, and Jeffrey Ullman. *COMPILATEURS. Principes, techniques et outils.* InterEditions, Paris, 1989.

[BBF+00] S. Bensalem, M. Bozga, J.-C. Fernandez, L. Ghirvu, and Y. Lakhnech. A Transformational Approach for Generating Non-Linear Invariants. In *[Pal00]*, 2000.

[BCM+92] J.R. Burch, E.M. Clarke, K.L. McMillan, D.L. Dill, and L.J. Hwang. Symbolic Model Checking: 10^{20} States and Beyond. *Information and Computation*, 98(2):142–170, 1992.

[BDA] Chourouk Bourhfir, Rachida Dssouli, and El Mostapha Aboulhamid. Automatic test generation for EFSM-based systems. Université de Montréal.

[BDHS00] D. Bošnački, D. Dams, L. Holenderski, and N. Sidorova. Model Checking SDL with SPIN. In *[GS00]*, 2000.

[BdS92] A. Bouali and R. de Simone. Symbolic bisimulation minimisation. In *Proc. 4th International Computer Aided Verification Conference*, pages 96–108, 1992.

[Bei90] Boris Beizer. *Software Testing Techniques*. Electrical-computer science and engineering series. Van Nostrand Reinhold, 1990.

[BFG99a] M. Bozga, J.-C. Fernandez, and L. Ghirvu. State Space Reduction based on Live Variables Analysis. In *[CF99]*, 1999.

[BFG+99b] Marius Bozga, Jean-Claude Fernandez, Lucian Ghirvu, Susanne Graf, Jean-Pierre Krimm, Laurent Mounier, and Joseph Sifakis. If: An intermediate representation for sdl and its applications. In *SDL Forum Proceedings*, Montreal, Canada., June 1999.

[BFG+99c] Marius Bozga, Jean-Claude Fernandez, Lucian Ghirvu, Susanne Graf, Jean-Pierre Krimm, Laurent Mounier, and Joseph Sifakis. If: An intermediate representation and validation environment for timed asynchronous systems. In *[WWD99]*, 1999.

[BFG00a] M. Bozga, J.-C. Fernandez, and L. Ghirvu. State Space Reduction based on Live Variable Analysis. *Science of Computer Programming*, 2000.

[BFG00b] M. Bozga, J.-C. Fernandez, and L. Ghirvu. Using Static Analysis To Improve Automatic Test Generation. In *[GS00]*, 2000.

[BFG+00c] M. Bozga, J.-C. Fernandez, L. Ghirvu, S. Graf, J.-P. Krimm, and L. Mounier. If: A validation environment for timed asynchronous systems (category b: tool presentation). In *[ES00]*, 2000.

[BFG+00d] M. Bozga, J.-C. Fernandez, L. Ghirvu, C. Jard, T. Jéron, A. Kerbrat, P. Morel, and L. Mounier. Verification and test generation for the SSCOP protocol. *Journal of Science of Computer Programming, special issue on Formal Methods in Industry*, 36(1):27–52, January 2000.

[BFGed] M. Bozga, J.-C. Fernandez, and L. Ghirvu. Using Static Analysis To Improve Automatic Test Generation. *International Journal on Software Tools for Technology Transfer*, Submitted,.

[BFKM98] Marius Bozga, Jean-Claude Fernandez, Alain Kerbrat, and Laurent Mounier. Protocol Verification with the Aldebaran Toolset. *International Journal on Software Tools for Technology Transfer*, 1(1+2):166–184, 1998.

[BGM02] M. Bozga, S. Graf, and L. Mounier. IF 2.0 : A Validation Environment for Component-Based Real-Time Systems. In *Computer Aided Verification, 14th International Conference, CAV 2002*, Copenhagen, DK, July 2002.

[BHR84] Stephen D. Brookes, C. A. R. Hoare, and A. W. Roscoe. A Theory of Communicating Sequential Processes. *Journal of the ACM*, 31(3):560–599, 1984.

[BJR00] L. du Bousquet, T. Jéron, and V. Rusu. An Approach to Symbolic Test Generation. Technical report, IRISA/INRIA, Project PAMPA, Rennes, FR, 2000.

[BK85] J.A. Bergstra and J.W. Klop. Algebra of communicating processes with abstraction. *Theoretical Computer Science*, 37:77–121, 1985.

[BL99] Saddek Bensalem and Yassine Lakhnech. Automatic generation of invariants. *Formal Methods in Systems Design*, 15:75–92, 1999.

[BLM01] M. Bozga, D. Lesens, and L. Mounier. Model-checking ariane-5 flight program. In *Proceedings of FMICS'01*, pages 211–227, Paris, FR, 2001.

[BLO98] Saddek Bensalem, Yassine Lakhnech, and Sam Owre. Invest: A tool for the verification of invariants. In *[HV98]*, pages 505–510, 1998.

[Boi98] Bernard Boigelot. *Symbolic Methods for Exploring Infinite State Spaces.* PhD thesis, University of Liège, May 1998.

[Bou85] G. Boudol. Le calcul Meije. In J.P. Verjus and G. Roucairo, editors, *Parallélisme, Communication, Synchronisation.* CNRS, 1985.

[Bou92] François Bourdoncle. *Sémantiques des Langages Impératifs d'Ordre Supérieur et Interprétation Abstraite.* Thèse de doctorat, École Polytechnique, 1992.

[Bou93] François Bourdoncle. Efficient chaotic iteration strategies with widenings. In *International Conference on Formal Methods in Programming and their Applications*, volume 735 of *Lecture Notes in Computer Science*, pages 128–141. Springer Verlag, 1993.

[Boz99] D.M. Bozga. *Vérification symbolique pour les protocoles de communication.* Thèse de doctorat, Université Joseph Fourier, Grenoble I, 1999.

[Bri88] E. Brinskma. A theory for derivation of tests. In S. Aggarwal and K. Sabnani, editors, *Protocol Specification, Testing and Verification VIII*, pages 63–74. North-Holland, 1988.

[BS97] Sébastien Bornot and Joseph Sifakis. Relating time progress and deadlines in hybrid systems, 1997.

[BST97a] S. Bornot, J. Sifakis, and S. Tripakis. Modelling Urgency in Timed Systems. In *International Symposium: Compositionality - The Significant Difference*, volume 1536 of *Lecture Notes in Computer Science*, September 1997.

[BST97b] Sébastien Bornot, Joseph Sifakis, and Stavros Tripakis. Modeling Urgency in Timed Systems. In *Compositionality: the significant difference, COMPOS'97*, volume 1536 of *Lecture Notes in Computer Science*, 1997.

[BSW69] K. A. Bartlett, R. A. Scantlebury, and P. T. Wilkinson. A note on reliable full-duplex transmission over half-duplex links. *Communications of the ACM*, 12(5):260–261, May 1969.

[BZ83] Daniel Brand and Pitro Zafiropulo. On communicating finite-state machines. *Journal of the ACM*, 30(2):323–342, April 1983.

[CC92a] Patrick Cousot and Radhia Cousot. Abstract interpretation frameworks. Rapport de recherche LIX-RR-92-10, Ecole Polytechnique, Palaiseau, France, May 1992.

[CC92b] Patrick Cousot and Radhia Cousot. Comparing the Galois connection and widenning/narrowing approaches to abstract interpretation. Rapport de recherche LIX-RR-92-09, Ecole Polytechnique, Palaiseau, France, June 1992.

[CDRV96] Pierre-Yves Calland, Alain Darte, Yves Robert, and Frédéric Vivien. On the removal of anti and output dependences. RR 2800, INRIA, February 1996.

[CES83] E. M. Clarke, E. A. Emerson, and A. P. Sistla. Automatic verification of finite state concurrent systems using temporal logic specifications. In *Conference Record of the Tenth Annual ACM Symposium on Principles of Programming Languages*, pages 117–126. ACM, ACM Press, January 1983.

[CES86] E. M. Clarke, E. A. Emerson, and A. P. Sistla. Automatic verification of finite state concurrent systems using temporal logic specifications. *ACM Transactions on Programming Languages and Systems*, 8(2), 1986.

[CF99] A. Cortesi and G. Filé, editors. *Static Analysis, Sixth International Symposium, SAS '99*, volume 1694 of *Lecture Notes in Computer Science*, Venezia. IT, September 1999. Springer.

[CFR+99] E.M. Clarke, M. Fujita, S.P. Rajan, T. Reps, S. Shankar, and T. Teitelbaum. Program Slicing for Design Automation : An Automatic Technique for Speeding-up Hardware Design, Simulation, Testing, and Verification. Technical report, CMU, 1999.

[CH78] P. Cousot and N. Halbwachs. Automatic discovery of linear restraints among variables of a program. In *Conference Record of the Fifth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 84–97, Tucson, Arizona, 1978. ACM Press, New York, NY.

[Che93] Jingde Cheng. Slicing concurrent programs - a graph-theoretical approach. In *Proceedings of the* 1^{st} *International Workshop on Automated and Algorithmic Debugging*, number 749 in Lecture Notes in Computer Science, pages 223–240. Springer-Verlag, 1993.

[Che96] C. Chevrier. *Test de conformité de protocoles de communication modèle de fautes et génération automatique de séquences de test.* PhD thesis, Université Bordeaux I, 1996.

[CJRR01] Franck Cassez, Claude Jard, Brigitte Rozoy, and Mark Dermot Ryan, editors. *Modeling and Verification of Parallel Processes, Fourth Summer School, MOVEP 2000*, volume 2067 of *Lecture Notes in Computer Science*, Nantes, France, June 2001. Springer.

[Cle90] R. Cleaveland. Tableau-based model checking in the propositional mu-calculus. *Acta Informatica*, 27(8):725–748, 1990.

[Con96] WAND Consortium. Magic WAND - Wireless ATM Network Demonstrator. URL : http://www.tik.ee.ethz.ch/wand, 1996.

[Cou78] Patrick M. Cousot. *Methodes iteratives de construction et d'approximation de points fixes d'operateurs monotones sur un treillis, analyse semantique des programmes.* Thèse de doctorat, Université Scientifique et Médicale de Grenoble, Institut National Polytechnique de Grenoble, 1978.

[Cou98] P. Cousot. Introduction to Abstract Interpretation. In *International Summer School Marktoberdorf on Calculational System Design*, 1998.

[CSE96] J. Callahan, F. Schneider, and S. Easterbrook. Specification-Based Testing using Model Checking. Technical Report NASA-IVV-96-022, NASA Software Research Laboratory, 1996.

[CZ93] S. Chanson and J. Zhu. A unified approach to protocol test sequence generation. In *Proceedings IEEE INFOCOM*, San Francisco, US, 1993.

[DC96] Matthew B. Dwyer and Lori A. Clarke. A flexible architecture for building data flow analyzers. In *Proceedings of the Eighteenth International Conference on Software Engineering*, pages 554–564, Berlin, Germany, March 1996.

[DC97] Matthew B. Dwyer and Lori A. Clarke. Data flow analysis frameworks for concurrent programs. KSU CIS TR 97-6, Kansas State University, Manhattan, KS, US, 1997.

[DGG97] Dennis Dams, Rob Gerth, and Orna Grumberg. Abstract Interpretation of Reactive Systems. *ACM Transactions on Programming Languages and Systems*, 19(2):253–291, March 1997.

[DHZ99] Matthew B. Dwyer, John Hatcliff, and Hongjun Zheng. Slicing software for model construction. *Journal of Higher-order and Symbolic Computation*, June 1999.

[DNH84] R. De Nicola and M.C.B. Hennessy. Testing equivalences for processes. *Theoretical Computer Science*, (34):83–133, 1984.

[DO91] R.A. DeMillo and A.J. Offutt. Constraint-based automatic test data generation. *IEEE Transactions on Software Engineering*, 17(9):900–910, September 1991.

[DPDea98] I. Dravapoulos, N. Pronios, and S. Denazis et al. The Magic WAND - Deliverable 3D5, Wireless ATM MAC, Final Report, August 1998.

[dT94] Union Internationale des Télécommunications. Couche d'adaptation du mode de transfert asynchrone du rnis à large bande - protocole en mode connexion propre au service. Recomandation UIT-T Q.2110, July 1994.

[Dwy95] Matthew Dwyer. *Data Flow Analysis for Verifying Correctness Properties of Concurrent Programs*. PhD thesis, Computer Science Department, University of Massachusetts at Amherst, September 1995.

[Dwy01] M.B. Dwyer, editor. *Model Checking Software, Eighth International Spin Workshop*, volume 2057 of *Lecture Notes in Computer Science*, Toronto, CA, May 2001. Springer.

[DY96] Conrado Daws and Sergio Yovine. Reducing the number of clock variables of timed automata. In *Proc. 1996 IEEE Real-Time Systems Symposium, RTSS'96*, Washington DC, USA, December 1996. IEEE Computer Society Press.

[EFM97] A. Engels, L.M.G. Feijs, and S. Mauw. Test generation for intelligent networks using model checking. In E. Brinksma, editor, *Proceedings of the Third International Workshop on Tools and Algorithms for the Construction and Analysis of Systems*, number 1217 in Lecture Notes in Computer Science, pages 384–398. Springer Verlag, 1997.

[ES00] E. Allen Emerson and A. Prasad Sistla, editors. *Computer Aided Verification, Twelfth International Conference, CAV 2000*, volume 1855 of *Lecture Notes in Computer Science*, Chicago, IL, US, July 2000. Springer.

[Fer88] Jean-Claude Fernandez. *ALDEBARAN - un système de vérification par réduction de processus communicants*. Thèse de doctorat, Université Joseph Fourier, Grenoble I, 1988.

[Fer96] Jean-Claude Fernandez. *Validation par évaluation sur un modèle : méthodes et algorithmes*. Document de présentation des travaux pour obtenir l'habilitation à diriger des recherches en informatique, Université Joseph Fourier, Grenoble I, 1996.

[FGK+96] Jean-Claude Fernandez, Hubert Garavel, Alain Kerbrat, Radu Mateescu, Laurent Mounier, and Mihaela Sighireanu. CADP:A Protocol Validation and Verification Toolbox. In Rajeev Alur and Thomas A. Henzinger, editors, *Proceedings of the 8th Conference on Computer-Aided Verification*, New Brunswick, New Jersey, USA, August 1996.

[FGM00] L.M.G. Feijs, N. Goga, and S. Mauw. Probabilities in the TorX test derivation algorithm. In *The 2nd Workshop of the SDL Forum Society on SDL and MSC - SAM 2000*, 2000.

[FJJM92] Jean-Claude Fernandez, Claude Jard, Thierry Jéron, and Laurent Mounier. "On the Fly" Verification of Finite Transition Systems. *Formal Methods in System Design*, 1:251–273, 1992.

[FJJV96] Jean-Claude Fernandez, Claude Jard, Thierry Jéron, and César Viho. Using on-the-fly verification techniques for the generation of test suites. In Rajeev Alur and Thomas A. Henzinger, editors, *Proceedings of the 8th International Conference on Computer-Aided Verification*, volume 1102 of *Lecture Notes in Computer Science*, Rutgers University, New Brunswick, NJ, USA, 1996. Springer Verlag. Also available as INRIA Research Report RR-2987.

[FJJV97] Jean-Claude Fernandez, Claude Jard, Thierry Jéron, and César Viho. An experiment in automatic generation of test suites for protocols with verification technology. *Science of Computer Programming*, 29(1-2), 1997. Special issue on Industrially Relevant Applications of Formal Analysis Techniques. Also available as INRIA Research Report RR-2923.

[FM95] Jean-Claude Fernandez and Laurent Mounier. A local checking algorithm for Boolean Equation Systems. Spectre 95-07, Verimag, Grenoble, France, 1995.

[fS88] International Organisation for Standardization. ISO/IEC 9074 Information Processing Systems. Open Systems Interconnection. ESTELLE: A Formal Description Technique based on an Extended State Transition Model, 1988.

[fS89] International Organisation for Standardization. ISO/IEC 8807 LOTOS: a Formal Description Technique based on the Temporal Ordering of Observational Behaviour, 1989.

[fS92a] International Organization for Standardization. ISO/IEC 9646-3 Information technology. Open Systems Interconnection. Conformance testing methodology and framework. Part 3: The Tree and Tabular Combined Notation (TTCN), 1992.

[fS92b] International Organization for Standardization. ISO/IEC 9646 Information technology. Open Systems Interconnection. Conformance testing methodology and framework, 1992.

[Gar98] Hubert Garavel. OPEN/CAESAR: An open software architecture for verification, simulation, and testing. In *Proceedings of the First International Conference on Tools and Algorithms for the Construction and Analysis of Systems TACAS'98*, March 1998. Full version available as INRIA Research Report RR-3352.

[GH99] A. Gargantini and C. Heitmeyer. Using Model Checking to Generate Tests from Requirements Specifications. In *Proceedings of the Joint* 7^{th} *European Software Engineering Conference and the* 7^{th} *ACM SIGSOFT International Symposium on Foundations of Software Engineering (ESEC/FSE99)*, volume 1687 of *Lecture Notes in Computer Science*, Toulouse, FR, September 1999.

[GKT91] Gina Goff, Ken Kennedy, and Chau-Wen Tseng. Practical dependence testing. In *Proceedings of the ACM SIGPLAN'91 Conference on Programming LanguageDesign and Implementation*, Toronto, CA, 1991.

[Gog01] N. Goga. Comparing TorX, Autolink, TGV and UIO Test Algorithms. In *[RR01]*, 2001.

[Gra90] P. Granger. Static Analysis of Linear Congruence Equalities Among Variables of a Program. Rapport de recherche LIX-RR-90-10, Ecole Polytechnique, Palaiseau, France, November 1990.

[Gro99] The Object Management Group. *OMG Unified Modeling Language Specification*, 1.3 edition, June 1999.

[GS00] Susanne Graf and Michael I. Schwartzbach, editors. *Tools and Algorithms for Construction and Analysis of Systems, Sixth International Conference, TACAS 2000, Held as Part of the European Joint Conferences on the Theory and Practice of Software, ETAPS 2000*, volume 1785 of *Lecture Notes in Computer Science*, Berlin, DE, March 2000. Springer.

[Har87] David Harel. Statecharts: A visual formalism for complex systems. *Science of Computer Programming*, 8(3):231–274, June 1987.

[HCD+99] John Hatcliff, James Corbett, Matthew Dwyer, Stefan Sokolowski, and Hongjun Zheng. A formal study of slicing for multi-threaded programs with jvm concurrency primitives. In *[CF99]*, 1999.

[Hec77] Matthew S. Hecht. *Flow Analysis of Computer Programs.* Elsevier North-Holland, Inc., 1977.

[Hen88] Matthew Hennessy. *Algebraic Theory of Processes.* The MIT Press, Cambridge, Mass., 1988.

[HLJ95] C.-M. Huang, Y.-C. Lin, and M.-Y. Jang. Executable data flow and control flow protocol test sequence generation for efsms specified protocol. In *International Workshop on Protocol Test Systems*, Evry, FR, 1995.

[HLSU02] Hyoung Seok Hong, Insup Lee, Oleg Sokolsky, and Hasan Ural. A temporal logic based theory of test coverage and generation. In *[KS02]*, 2002.

[HNSY94] T.A. Henzinger, X. Nicollin, J. Sifakis, and S. Yovine. Symbolic model checking for real-time systems. *Information and Computation*, 111(2), 1994.

[Hoa78] C.A.R. Hoare. Communicating sequential processes. *Communications of the ACM*, 21(8):666–677, August 1978.

[Hoa85] C.A.R. Hoare. *Communicating Sequential Processes.* Prentice Hall, 1985.

[Hol91] Gerard J. Holzmann. *Design and Validation of Computer Protocols.* Prentice Hall, 1991.

[Hol97] G.J. Holzmann. The Model Checker SPIN. *IEEE Transactions on Software Engineering*, 23(5):279–295, May 1997.

[HPR97] N. Halbwachs, Y.E. Proy, and P. Roumanoff. Verification of real-time systems using linear relation analysis. *Formal Methods in System Design*, 11(2):157–185, August 1997.

[HU79] J.E. Hopcroft and J.D. Ullman. *Introduction to automata theory, languages and computation*. Addison Wesley, 1979.

[HU00] O. Henniger and H. Ural. Test generation based on control and data dependencies within multi-process SDL specifications. In *The 2nd Workshop of the SDL Forum Society on SDL and MSC - SAM 2000*, 2000.

[HV98] Alan J. Hu and Moshe Y. Vardi, editors. *Computer Aided Verification, Tenth International Conference, CAV '98, Proceedings*, volume 1427 of *Lecture Notes in Computer Science*, Vancouver, BC, CA, June 1998. Springer.

[JG01] Guoping Jia and Susanne Graf. Verification Experiments on the Mascara Protocol. In *[Dwy01]*, 2001.

[JM97] Thierry Jéron and Pierre Morel. Abstraction, τ-réduction et déterminisation à la volée : application à la génération de test. In G. Leduc, editor, *Colloque Français pour l'Ingénierie des Protocoles*. Hermes, September 1997.

[JM99] Thierry Jéron and Pierre Morel. Test generation derived from model-checking. In *Proceedings of the 11th International Conference on Computer-Aided Verification*, Trento, Italy, July 1999.

[Kea96] Baker R. Kearfott. Interval computations : introduction, uses and resources. *Euromath Bulletin*, 2(1):95–112, 1996.

[Ker94] Alain Kerbrat. *Méthodes symboliques pour la vérification de processus communicants : étude et mise en oeuvre*. Thèse de doctorat, Université Joseph Fourier, Grenoble I, 1994.

[Kil73] G.A. Kildall. A Unified Approach to Global Program Optimization. In *ACM Symposium on Principles of Programming Languages*, pages 194–206. ACM, ACM Press, October 1973.

[KJG99] A. Kerbrat, T. Jéron, and R. Groz. Automated test generation from SDL specifications. In *Proceedings of SDL Forum*. Elsevier Science (North Holland), 1999.

[KM00] J.-P. Krimm and L. Mounier. Compositional State Space Generation with Partial Order Reductions for Asynchronous Communicating Systems. In *[GS00]*, 2000.

[Koh78] Z. Kohavi. *Switching and Finite Automata Theory.* McGraw-Hill, 1978.

[Koz83] D. Kozen. Results on the propositional mu -calculus. *Theoretical Computer Science*, 27(3):333–354, December 1983.

[Kri98] J. Krinke. Static Slicing of Threaded Programs. In *Proceedings of the ACM SIGPLAN/SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*, pages 35–42, 1998.

[KS02] Joost-Pieter Katoen and Perdita Stevens, editors. *Tools and Algorithms for the Construction and Analysis of Systems, Eighth International Conference, TACAS 2002, Held as Part of the Joint European Conference on Theory and Practice of Software, ETAPS 2002, Proceedings*, volume 2280 of *Lecture Notes in Computer Science*, Grenoble, France, April 2002. Springer.

[KVZ98] Hakim Kahlouche, César Viho, and Massimo Zendri. An industrial experiment in automatic generation of executable test suites for a cache coherency protocol. In *International Workshop on Testing of Communicating Systems*, Tomsk, Russia, September 1998.

[LGS+94] Claire Loiseaux, Susanne Graf, Joseph Sifakis, Ahmed Bouajjani, and Saddek Bensalem. Property preserving abstractions for the verification of concurrent systems. *Formal Methods in System Design*, 6:1–35, 1994.

[LT89] N. A. Lynch and M. R. Tuttle. An introduction to input/output automata. *CWI Quarterly*, 2(3):219–246, 1989.

[LY94] D. Lee and M. Yannakakis. Testing finite-state machines: State identification and verification. *IEEE Transactions on Computers*, 43(3), March 1994.

[LY96] David Lee and Mihalis Yannakakis. Principles and methods of testing finite state machines - a survey. *Proceedings of the IEEE*, 84(8):1090–1123, August 1996.

[Mil80] Robin Milner. *A Calculus for Communicating Systems*, volume 92 of *Lecture Notes in Computer Science*. Springer-Verlag, Berlin, 1980.

[Mor00] Pierre Morel. *Une algorithmique efficace pour la génération automatique de tests de conformité*. Thèse de doctorat, Université de Rennes 1, 2000.

[MOSS99] Markus Müller-Olm, David Schmidt, and Bernhard Steffen. Model-checking - a tutorial introduction. In *[CF99]*, 1999.

[Mou92] Laurent Mounier. *Méthodes de vérification de spécifications comportementales : étude et mise en œuvre*. Thèse de doctorat, Université Joseph Fourier, Grenoble I, January 1992.

[MR90] T.J. Marlowe and B.G. Ryder. Properties of data flow frameworks. A unified model. *Acta Informatica*, 28(2):121–163, 1990.

[MRB95] T.J. Marlowe, B.G. Ryder, and M. Burke. Defining Flow Sensitivity for Data Flow Problems. Technical Report LCSR-TR-249, Laboratory of Computer Science Research, July 1995.

[MS96] Tiziana Margaria and Bernhard Steffen, editors. *Tools and Algorithms for Construction and Analysis of Systems, Second International Workshop, TACAS '96, Proceedings*, volume 1055 of *Lecture Notes in Computer Science*, Passau, DE, March 1996. Springer.

[MT98] L.I. Millett and T. Teitelbaum. Slicing Promela and its Applications to Model Checking, Simulation, and Protocol Understanding. In 4^{th} *International SPIN Workshop*, ENST, Paris, FR, November 1998.

[Muc97] Steven S. Muchnick. *Advanced Compiler Design Implementation*. Morgan Kaufmann Publishers, San Francisco, CA, 1997.

[Neu99] Peter G. Neumann. Robust open-source software. *Communications of the ACM*, 42(2):128, February 1999.

[NNH99] F. Nielson, H.R. Nielson, and C. Hankin. *Principles of Program Analysis*. Springer, October 1999.

[NSS94] Esko Nuutila and Eljas Soisalon-Soininen. On Finding the Strongly Connected Components in a Directed Graph. *Information Processing Letters*, 49:9–14, 1994.

[OFMP+94] Anders Olsen, Ove Færgemand, Birger Møller-Pedersen, Rick Reed, and J.R.W. Smith. *Systems engineering using SDL-92*. Elsevier Science, 1994.

[OL99] A.J. Offutt and S. Liu. Generating Test Data from SOFL Specifications. *Journal of Systems and Software*, 49(1):49–62, 1999.

[Pac01] Cyril Pachon. Génération automatique de tests symboliques. Rapport D.E.A., Université Joseph Fourier, Grenoble I, 2001.

[Pal00] J. Palsberg, editor. *Static Analysis, Seventh International Symposium, SAS 2000*, volume 1824 of *Lecture Notes in Computer Science*, Santa Barbara, CA, US, June 2000. Springer.

[Par81] D.M.R. Park. Concurrency and Automata on Infinite Sequences. In *Proceedings of the 5th GI Conference on Theoretical Computer Science*, number 104 in Lecture Notes in Computer Science, Berlin, 1981. Springer-Verlag.

[Pha94a] M. Phalippou. *Relations d'Implantation et Hypothèses de Test sur des Automates à Entrées et Sorties.* Thèse de doctorat, L'Université de Bordeaux I, France, 1994.

[Pha94b] Marc Phalippou. Test sequence generation using Estelle or SDL structure information. In *Formal Description Techniques for Distributed Systems*, Berne, October 1994.

[Pnu77] A. Pnueli. The Temporal Logic of Programs. In *IEEE Symposium on Foundations of Computer Science*, pages 46–57, 1977.

[PT87] Robert Paige and Robert E. Tarjan. Three partition refinement algorithms. *SIAM Journal on Computing*, 16(6):973–989, December 1987.

[PW86] David A. Padua and Michael J. Wolfe. Advanced compiler optimization for supercomputers. *Communications of the ACM*, 29(12):1184–1201, December 1986.

[QS82] J.P. Queille and J. Sifakis. Specification and verification of concurrent systems in CESAR. In *Proceedings of the Fifth International Symposium in Programming*, volume 137 of *Lecture Notes in Computer Science*, pages 337–351, New York, 1982. Springer-Verlag.

[QS83] J.-P. Queille and J. Sifakis. Fairness and related properties in transition systems. *Acta Informatica*, 19:195–220, 1983.

[Ray87] D. Rayner. OSI conformance testing. *Computer Networks and ISDN Systems*, (14):79–98, 1987.

[RBJ00] V. Rusu, L. du Bousquet, and T. Jéron. An Approach to Symbolic Test Generation. In *International Conference on Integrating Formal Methods (IFM'00)*, volume 1945 of *Lecture Notes in Computer Science*, pages 338–357, Dagstuhl, DE, 2000.

[RDT95] T. Ramalingom, A. Das, and K. Thulasiraman. A unified test case generation method for the efsm model using context independent unique sequences. In *International Workshop on Protocol Test Systems*, Evry, FR, 1995.

[RR88] Helmut Ratschek and Jon Rokne. *New computer methods for global optimization*. Ellis Horwood series in mathematics and its applications. Numerical analysis, statistics and operational research. Ellis Horwood, John Wiley, 1988.

[RR01] Rick Reed and Jeanne Reed, editors. *SDL 2001: Meeting UML, Tenth International SDL Forum ,Proceedings*, volume 2078 of *Lecture Notes in Computer Science*, Copenhagen, Denmark, June 2001. Springer.

[RW85] S. Rapps and E.J. Weyuker. Selecting Software Test Data Using Data Flow Information. *IEEE Transactions on Software Engineering*, SE-11(4):367–375, April 1985.

[Sec94] Standardization Sector. ITU-T. Recommendation Z-100. Specification and Description Language (SDL). Technical Report Z-100, International Telecommunication Union, Genève, CH, 1994.

[SEG+98] Michael Schmitt, Anders Ek, Jens Grabowski, Dieter Hogrefe, and Beat Koch. Autolink - Putting SDL-based test generation into practice. In A. Petrenko and N . Yevtuschenko, editors, *Testing of Communicating Systems*, volume 11. Kluwer Academic Publishers, 1998.

[Sif82] J. Sifakis. A unified approach for studying the properties of transition systems. *Theoretical Computer Science*, 18, 1982.

[Sif99] J. Sifakis. Integration, the Price of Success - Extended Abstract. In *[WWD99]*, pages 52–55. Springer-Verlag, 1999.

[SS98] D. Schmidt and B. Steffen. Program Analysis as Model Checking of Abstract Interpretations. In *Proceedings of Static Analysis Symposium*, volume 1503 of *Lecture Notes in Computer Science*, pages 351–380. Pisa, IT, Springer–Verlag, September 1998.

[TB99] Jan Tretmans and Axel Belinfante. Automatic Testing with Formal Methods. Technical Report TR-CTIT-17, CTIT, University of Twente, 1999.

[Tel] Telelogic Technologies Toulouse. *ObjectGEODE*. URL : http://www.telelogic.com/products/objectgeode/.

[Tip94] Frank Tip. A Survey of Program Slicing Techniques. CS R9438, Centrum voor Wiskunde en Informatica, Amsterdam, NL, 1994.

[Tre92] Jan Tretmans. *A Formal Approach to Conformance Testing*. PhD thesis, Twente University, 1992.

[Tre96] Jan Tretmans. Test generation with inputs, outputs and quiescence. In *[MS96]*, pages 127–146, 1996.

[Vir] European Community's Fourth Framework Project VIRES (Esprit LTR 23498). http://radon.ics.ele.tue.nl/~vires/.

[Wei84] Mark Weiser. Program Slicing. *IEEE Transactions on Software Engineering*, SE-10(4):352–357, July 1984.

[WWD99] J.M. Wing, J. Woodcock, and J. Davies, editors. *FM'99 - Formal Methods, World Congress on Formal Methods in the Development of Computing Systems*, volume 1708-1709 of *Lecture Notes in Computer Science*, Toulouse, FR, September 1999. Springer.

[WZ91] Mark N. Wegman and F. Kenneth Zadeck. Constant propagation with conditional branches. *ACM Transactions on Programming Languages and Systems*, 13(2):181–210, April 1991.

[XRK99] Spyros Xanthakis, Pascal Regnier, and Constantin Karapoulios. *Test et contrôle des logiciels*. Hermes, Paris, FR, 1999.

[Yov97] S. Yovine. Kronos: A verification tool for real-time systems. *Springer International Journal of Software Tools for Technology Transfer*, 1(1-2), October 1997.