Vérification et synthèse de systèmes réactifs

David Lesens

THÈSE

présentée par

David LESENS

pour obtenir le grade de DOCTEUR
de l'INSTITUT NATIONAL POLYTECHNIQUE DE GRENOBLE
(Arrêté ministériel du 30 mars 1992)

(Spécialité : Informatique)

VÉRIFICATION ET SYNTHÈSE DE SYSTÈMES RÉACTIFS

Date de soutenance : 5 septembre 1997

Composition du jury :	Président	Paul Jacquet
	Rapporteurs	Patrick Cousot
		Pierre Wolper
	Examinateurs	Nicolas Halbwachs
		Pascal Raymond

Thèse préparée au sein de l'Unité Mixte de Recherche VÉRIMAG.

Remerciements

Je remercie :

Monsieur Paul Jacquet, Professeur à l'Institut National Polytechnique de Grenoble, pour m'avoir fait l'honneur d'accepter de présider le jury de cette thèse.

Messieurs Patrick Cousot, Professeur à l'Ecole Normale Supérieure de Paris, et Pierre Wolper, Professeur à l'Université de Liège, pour avoir accepté de juger ce travail et d'en être rapporteurs.

Messieurs Nicolas Halbwachs, Directeur de Recherche au CNRS, et Pascal Raymond, Chargé de Recherche au CNRS, qui ont dirigé cette thèse et sans lesquels ce travail n'aurait pu aboutir. Je les remercie en particulier pour leur grande compétence, leur disponibilité, et le soutien qu'ils m'ont apporté durant ces trois années.

Cette thèse a été effectué dans l'Unité Mixte de Recherche VÉRIMAG. Je tiens à remercier Monsieur Joseph Sifakis, Directeur de Recherche au CNRS, de m'avoir accueilli dans cette équipe.

Je tiens également à remercier l'ensemble des membres de VÉRIMAG pour leur soutien amical et pour le plaisir que j'ai eu à travailler avec eux.

Table des matières

Introduction

Systèmes réactifs

Les systèmes réactifs sont des systèmes informatiques qui réagissent continûment à leur environnement physiqueΓà une vitesse déterminée par cet environnement. Par celaΓils s'opposent d'une part aux systèmes transformationnels (les programmes classiquesΓqui disposent de leurs entrées dès leur initialisationΓet délivrent leurs résultats lors de leur terminaison)Γet d'autre part aux systèmes interactifs (qui interagissent continûment avec leur environnementΓmais à leur vitesse propreΓcomme les systèmes d'exploitation).

Ce type de système est particulièrement utilisé dans le cadre du contrôle des systèmes critiques : transportΓnucléaireΓcommandes de processus industrielsΓetc ... De par leur fonctionΓ ces systèmes doivent impérativement satisfaire des contraintes strictes de fonctionnement. Leur défaillance peut entraîner des coûts considérablesΓaussi bien en terme financier (arrêt d'un serviceΓ accident entraînant la détérioration de matérielΓetc ...) qu'en terme humain (par exemple dans le transport aérien ou dans le contrôle des centrales nucléaires).

Les propriétés qu'on considère sur ce type de systèmes consistentΓle plus souventΓen des relations logiques simples entre les entrées qu'ils reçoivent de leur environnement et les sorties qu'ils lui transmettent. De plusΓles propriétés de l'environnement doivent être fortement prises en compte pour vérifier que le système satisfait une propriété donnée. Cet environnement peut être soit une modélisation d'un environnement physique (le "monde réel")Γsoit une abstraction d'un autre programme informatique. Dans le cas des réseaux de processusΓchaque processus voit les autres comme faisant partie d'un environnement global.

De nombreuses recherches ont été entreprises pour vérifier de manière formelle ce genre de propriétés (voir [QS82ΓCES86] pour les plus anciennes) : une technique formelle consiste à baser une preuve sur un raisonnement mathématiqueΓplutôt que sur un raisonnement en langage naturel (qui risque souvent d'être ambigu). Dans le cadre du langage Lustre [HCRP91] (sur lequel se base le présent travail)Γl'outil Lesar [HLR92bΓRat92] permet ainsi de vérifier qu'un programme booléen (ou que l'abstraction booléenne d'un programme plus complexe) de taille raisonnable satisfait une propriété de sûreté.

L'ensemble de ces techniques se heurtent au problème de la taille des modèles générés. Les techniques actuelles ne sont guère capables de construire des preuves pour des systèmes dépassant quelques dizaines de millions d'étatsΓce qui est souvent largement insuffisant. En particulierΓ ces techniques de preuves ne peuvent pas être appliquées à des réseaux de processus distribuésΓ dès que ces réseaux dépassent la centaineΓvoir la dizaine de processus (10 processus à 10 états forment un système à 10 milliards d'états !). Différentes techniques sont explorées pour pallier à ce problème qui empêche leur exploitation industrielle à grande échelle. Dans ce documentΓnous nous intéresserons à la preuve modulaire.

Preuve modulaire

La preuve modulaire consiste à décomposer un programme en plusieurs sous-programmes de taille plus petiteΓsur lesquels on espère pouvoir utiliser les techniques précédentes. Cette technique

nous amène à nous poser un certain nombre de questions :

- Ayant prouvé une propriété d'un sous-programmeΓcomment en tirer parti pour vérifier le programme complet?

  Cette question est à la base de la vérification modulaire. Ayant vérifié une propriété d'un sous-programmeΓcelui-ci peut être considéré comme faisant partie de l'environnement des autres processusΓenvironnement quiΓdès lorsΓpeut être garanti posséder la propriété prouvée.

- InversementΓcomment trouver une propriété suffisante d'un sous-programme pour que le programme complet satisfasse les propriétés désirées?

  La propriété à démontrer sur un sous-programme doit être suffisante pour prouver une propriété sur le système global. Cette propriétéΓdifficile à trouver manuellementΓpeut être générée automatiquement.

- Est-il possible de déduire le code du sous-programme à partir des propriétés qu'il doit satisfaire?

  Si un programme est spécifié de manière suffisamment strictеΓil semble plus efficace (en terme de coût financierΓainsi que de sûreté du logiciel) de générer automatiquement l'ensemble des sous-programmes satisfaisant cette propriétéΓplutôt que de construire manuellement un tel programmeΓet de vérifier à posteriori qu'il satisfait la spécification désirée.

- Comment étendre la preuve modulaire à un nombre arbitraire de processus?

  Un processus d'un réseau voit les autres processus comme son environnement. Il est alors possible de généraliser la synthèse de spécification pour générer un invariant d'un réseau régulierΓet ainsi prouver que le réseau complet satisfait une propriété.

Hypothèse synchrone et le langage LUSTRE

Notre travail se place dans le cadre de la vérification de systèmes réactifs décrits à l'aide du langage LUSTRE [HCRP91]. LUSTRE est un langage déclaratif synchrone dédié à la programmation de systèmes temps réel. LUSTRE a servi de base à des ateliers industriels de conception de programmesΓcomme l'atelier SAGAΓdéveloppé et utilisé par Merlin-GerinΓet la prochaine version de l'atelier CSAOΓdéveloppé par Vérilog pour l'Aérospatiale.

Il repose sur deux concepts originaux : une approche flot de donnée et une interprétation synchrone.

- L'approche flot de donnée consiste à considérer chaque sortie du système comme une fonction du temps discret. Les opérateurs du langage opèrent alors sur ces fonctions.

- L'approche synchrone consiste à supposer que toutes les variables du système évoluent simultanémentΓi.e.Γque le système complet est synchronisé sur une horloge globale.

L'approche suivie pour la vérification consiste à spécifier des propriétés dans le même langageΓà l'aide d'un programme "observateur"Γdétectant les anomalies de fonctionnement du programme à vérifier. On étudie alors le programme construit en exécutant en parallèle le programme à vérifier et son observateurΓen tentant de montrer qu'aucune anomalie n'est détectéeΓquelles que soient les suites d'entrées soumises au programme.

Dans ce documentΓnous étendons la notion de processus observateur d'une propriété de sûretéΓ

- d'une part aux propriétés de vivacité (exprimables par un automate de Büchi).

- d'autre part aux réseaux de processus.

Plan

Dans un premier chapitreΓnous présentons le cadre de notre étude. Les automates à entrée/sortie synchrones sont définisΓles techniques classiques de vérification sont rappelées. Une problématique est alors proposée.

Ce document est ensuite divisé en trois parties :

Synthèse de processus : Dans cette première partieΓnous étudions la synthèse de programmes. Le chapitre 2 présente l'état de l'art dans le domaine. Nous concluons en posant le problème de la synthèse dans le cas des programmes synchrones : il s'agit de générer automatiquement l'ensemble des programmes quiΓdans un environnement correctΓsatisfont une propriété donnée. Le chapitre 3 propose de décrire ces solutions à l'aide d'un observateur. L'observateur synthétisé est capable à tout moment de fournir les comportements possibles de notre programmeΓde façon à ce qu'il puisse satisfaire la propriété désirée. Le chapitre 4 finit de résoudre le problème de la synthèse. L'ensemble des solutions du problème de la synthèse est décrit : ces solutions sont alors des programmes exécutables par un ordinateur. Dans le chapitre 5Γnous décrivons quelques exemples de synthèses; des mesures de performances sont présentées et commentées.

Synthèse d'invariants : Dans cette deuxième partieΓnous étendons la synthèse d'un seul processusΓà la synthèse d'un invariant pour un ensemble arbitraire de processusΓorganisés en réseau régulier. Le chapitre 6 présente l'état de l'art sur la spécification et la vérification des systèmes paramétrés. Dans le chapitre 7Γnous décidons de décrire un réseau constitué d'un nombre arbitraire de processus à l'aide d'une grammaire de réseau [CGJ95]. Nous proposons une nouvelle technique générale de vérification en définissant un invariant du réseau comme étant un plus petit point fixe. Un exemple d'application de cette technique est donné. Les deux chapitres suivants étendent cette technique au calcul de plus grands points fixes. En effetΓalors que la taille du plus petit point fixe dépend uniquement de celle du système (qui est lui-même généralement infini)Γ la taille du plus grand point fixe dépend fortement de celle de la propriété à démontrer (qui est généralement de taille réduite). Le chapitre 8 propose ainsi d'exprimer un invariant d'un réseau linéaire (ou en anneau) comme un plus grand point fixe. Une série d'heuristiques est proposée de façon à permettre le calcul d'une approximation inférieure de ce plus grand point fixe. Le chapitre 9 étend ce résultat aux réseaux arborescentsΓen décomposant l'invariant selon le nombre de branches de l'arbre. Le chapitre 10 présente un certain nombre d'exemples utilisant cette technique de preuve. Le chapitre 11 propose enfin d'utiliser cette technique comme une aide au calcul d'invariants. A l'aide d'exemplesΓplusieurs extensions combinant différentes méthodes d'abstraction sont proposéesΓpermettant de résoudre des problèmes complexes.

Implémentation : La troisième partie décrit l'implémentation qui a été réalisée et qui applique l'ensemble des algorithmes proposés dans ce document. Le chapitre 12 consiste en un petit manuel d'utilisation de l'outil qui a été réalisé : BANG (pour "Boolean Automata Network Grammars checker"). Certains choix d'implémentation sont discutés. Le chapitre 13 propose des algorithmes pour calculer l'abstraction de signaux. Ce point est déterminant pour l'efficacité de notre calcul d'invariant. Le chapitre 14 décrit enfin quelques algorithmes de calcul de la post et de la précondition. Des tests expérimentaux ont été effectués et sont présentés pour choisir le meilleur algorithme.

Chapitre 1

Problématique et modèles

Dans ce chapitre nous présentons les objectifs de ce document et le formalisme utilisé. Après la présentation des modèles de langage et de processus synchrones utilisés dans la suite nous décrivons la problématique de la vérification de propriétés sur les langages synchrones. Nous proposons deux solutions pour éviter l'explosion du nombre des états : la modularité et la preuve par induction.

1.1 Langages et automates

Nous considérons dans ce document une sémantique de traces pour décrire des processus. Le comportement d'un processus est une séquence de pas chaque pas correspondant à la réception et à l'émission de signaux.

Le modèle utilisé est celui des langages synchrones [Hal93] tels LUSTRE [HCRP91] ESTEREL [BG92] ARGOS [Mar92] SIGNAL [GGB87] ou STATECHARTS [Har87]. Il est également le modèle sous-jacent classique des circuits synchrones.

1.1.1 Mots et langages

Nous considérons des processus communicants à l'aide de signaux. Chaque signal est représenté par un symbole. On appelle alphabet un ensemble fini de symboles représentant des signaux. Soit $X = (x^0, \ldots, x^{n-1})$ un alphabet de taille n. Un événement synchrone x sur X est un ensemble de signaux i.e. un sous-ensemble de X ($x \in 2^X$). La dénomination synchrone signifie que tous les signaux d'un événement sont supposés survenir au même instant.

Une exécution d'un processus synchrone est vue par un observateur extérieur comme une suite d'événements composés par des signaux de son interface. On définit ainsi une exécution d'un processus comme étant une trace (ou mot synchrone) sur X. Un mot synchrone τ sur X est une suite finie ou infinie d'événements synchrones sur X ($\tau = (x_0, x_1, \ldots)$ avec $\forall i, x_i \in 2^X$). La longueur d'un mot τ est notée $|\tau|$ ($0 \leq |\tau| \leq \infty$). On note ε le mot de longueur nulle.

Exemple 1.1 :

Soit $X = \{a, b, c\}$. Alors $\{a\}$ $\{a, c\}$ $\{a, b, c\}$ et $\emptyset$ sont des événements synchrones sur X. $(\{a\}, \{b, c\}, \emptyset, \emptyset, \{a, b\}, \ldots)$ est un mot synchrone sur X. □

Un ensemble fini ou infini de mots est appelé un langage. Un processus est décrit par l'ensemble de ses traces donc par un langage.

Si $\tau = (x_0, \ldots, x_n)$ est un mot fini et $\tau' = (x'_0, \ldots, x'_n, \ldots)$ un mot de taille quelconque on note $\tau.\tau'$ la concaténation de ces deux mots i.e.

$$\tau.\tau' = (x_0, \ldots, x_n, x'_0, \ldots, x'_n, \ldots)$$

Notons X^∞ l'ensemble des mots (finis ou infinis)ΓX^ω l'ensemble des mots infinisΓX^* l'ensemble des mots finisΓet X^n l'ensemble des mots de taille n sur X.

Les langages de mots finis peuvent être utilisés dans la spécification de propriété de terminaison : lorsqu'une tâche est terminéeΓon peut s'assurer que le processus est dans un état correct. Dans un système réactifΓnous nous intéresserons plus particulièrement aux mots infinis : le système ne s'arrête jamais.

1.1.2 Automates

Soit X un ensemble fini de signauxΓet soit 2^X l'ensemble des événements sur X. Nous posons ici une définition des automates de BüchiΓadaptée au modèle synchrone.

Définition 1.1 :

Un automate à entrée/sortie A sur X est un tuple $(Q_A, q_A^0, I_A, O_A, \delta_A)$ où

- Q_A est un ensemble fini d'états contenant l'état initial q_A^0.
- $I_A \subseteq X$ et $O_A \subseteq X$ sont les ensembles disjointsΓrespectivement des signaux d'entrée et de sortie.
  On appelle configuration un couple formé d'un état et d'un événement d'entrée.
  $$(q, i) \in Q_A \times 2^{I_A}$$
- $\delta_A \subseteq Q_A \times 2^{I_A} \times 2^{O_A} \times Q_A$ est la relation de transition.
  Si l'automate est dans la configuration $(q, i) \in Q_A \times 2^{I_A}$ Γil peut passer dans l'état q' en émettant la sortie o si et seulement si $(q, i, o, q') \in \delta_A$.
  Quand il n'y aura pas d'ambiguïté sur la relation de transitionΓon notera $q \xrightarrow[o]{i} q'$ à la place de $(q, i, o, q') \in \delta_A$.

Le mot $\tau = (x_0, x_1, \dots, x_n, \dots)$ sur $X = I_A \cup O_A$ est reconnu (ou généré) par l'automate $A = (Q_A, q_A^0, I_A, O_A, \delta_A)$ si et seulement s'il existe une suite infinie $(q_0, q_1, \dots, q_n, \dots)$ d'étatsΓavec $q_0 = q_A^0$ et pour tout $n \geq 1, q_{n-1} \xrightarrow[x_{n-1} \cap O_A]{x_{n-1} \cap I_A} q_n$. □

Un tel automate peut donc être interprété comme un reconnaisseur de langage sur X (ou éventuellement un générateur de langageΓvoir [RW89]).

De façon à décrire des propriétés de vivacitéΓnous introduisons un ensemble d'états marqués $F_A \subseteq Q_A$. Un automate à entrée/sortie A sur X est alors un tuple $(Q_A, q_A^0, I_A, O_A, \delta_A, F_A)$. En posant $F_A = Q_A\Gamma$on retrouve le modèle initial.

IntuitivementΓpour les mots finisΓen réponse à une séquence $(i_0, i_1, \dots, i_{n-1})$ d'événements d'entréeΓun tel automate retourne une séquence $(o_0, o_1, \dots, o_{n-1})$ d'événements de sortieΓtelle qu'il existe une séquence $(q_0, q_1, \dots, q_n)$ d'étatsΓavec $q_0 = q_A^0 \Gamma q_n \in F_A$ (l'exécution commence dans l'état initial et finit dans un état marqué) et pour tout $n \geq 1, q_{n-1} \xrightarrow[o_{n-1}]{i_{n-1}} q_n$. Dans ce cas$\Gamma$l'ensemble F_A est également appelé l'ensemble des états finals (toute exécution finie doit se terminer dans un état final).

La séquence $(i_0 \cup o_0, i_1 \cup o_1, \dots, i_{n-1} \cup o_{n-1})$ est appelée une trace de l'automate A. Une telle trace est un mot sur X.

Définition 1.2 :

- Le mot fini $\tau = (x_0, x_1, \dots, x_{n-1})$ sur $X = I_A \cup O_A$ est reconnu (ou généré) par l'automate $A = (Q_A, q_A^0, I_A, O_A, \delta_A, F_A)$ si et seulement s'il existe une suite $(q_0, q_1, \dots, q_n)$ d'étatsΓavec $q_0 = q_A^0 \Gamma q_n \in F_A$ et pour tout $n \geq 1, q_{n-1} \xrightarrow[x_{n-1} \cap O_A]{x_{n-1} \cap I_A} q_n$.

– Le mot infini $\tau = (x_0, x_1, \ldots, x_n, \ldots)$ sur $X = I_A \cup O_A$ est reconnu (ou généré) par l'automate $A = (Q_A, q_A^0, I_A, O_A, \delta_A, F_A)$ si et seulement s'il existe une suite infinie $(q_0, q_1, \ldots, q_n, \ldots)$ d'états avec $q_0 = q_A^0$ et pour tout $n \geq 1, q_{n-1} \xrightarrow[x_{n-1} \cap O_A]{x_{n-1} \cap I_A} q_n$ et telle qu'il existe un nombre infini d'indices i tels que $q_i \in F_A$.
Intuitivement, comme le mot τ est infini, on n'exige plus qu'il se termine dans un état final, mais qu'il passe infiniment souvent par un état final.

□

L'ensemble des traces d'un automate A est un langage sur X qui est noté T_A ($T_A \subseteq X^\infty$).

Définition 1.3 :

Un langage T est dit régulier s'il existe un automate A reconnaissant ce langage ($T = T_A$). □

Les langages réguliers ont été très étudiés. De nombreux algorithmes ont été proposés pour effectuer les opérations classiques (intersection, union, complémentation, etc ...). Un résumé de l'état de l'art sur les langages réguliers et les automates peut être trouvé dans [Arn92, KG95].

Pour qu'un automate modélise un processus réel, il est souvent nécessaire qu'il soit réactif et déterministe.

Définition 1.4 :

Un automate *réactif* ne peut pas refuser un événement d'entrée. Il possède donc la propriété

$$\forall q \in Q_A, \forall i \in 2^{I_A}, \exists o \in 2^{O_A}, \exists q' \in Q_A, q \xrightarrow[o]{i} q'$$

□

Cette définition signifie que la vitesse du processus est déterminée par son environnement (le processus ne peut pas ralentir son environnement). Nous proposerons dans la suite un algorithme pour rendre réactif un automate (voir chapitre 3, page 33).

Définition 1.5 :

Un automate *déterministe* a au plus une réaction possible à un événement d'entrée donné. Il possède donc la propriété

$$\begin{aligned} &\forall q \in Q_A, \forall i \in 2^{I_A}, \forall o_1, o_2 \in 2^{O_A}, \forall q_1', q_2' \in Q_A, \\ &\quad q \xrightarrow[o_1]{i} q_1' \text{ et } q \xrightarrow[o_2]{i} q_2' \implies o_2 = o_1 \text{ et } q_2' = q_1' \end{aligned}$$

Dans ce cas, on préfère remplacer la relation de transition par une fonction de transition et par une fonction de sortie. Un automate déterministe A est alors défini par un tuple $(Q_A, q_A^0, I_A, O_A, \delta_A^Q, \delta_A^O)$ où

– $\delta_A^Q \in Q_A \times 2^{I_A} \to Q_A$ est la fonction de transition.
Si l'automate est dans la configuration $(q, i) \in Q_A \times 2^{I_A}$, l'état suivant est $\delta_A^Q(q, i) \in Q_A$.
– $\delta_A^O \in Q_A \times 2^{I_A} \to 2^{O_A}$ est la fonction de sortie.
Si l'automate est dans la configuration $(q, i) \in Q_A \times 2^{I_A}$, l'automate émet l'événement de sortie $\delta_A^O(q, i) \in 2^{O_A}$.

En d'autres termesΓ

$$\forall q, q' \in Q_A, \forall i \in 2^{I_A}, \forall o \in 2^{O_A}, \quad \left(\ (q', o) = \left(\ \delta_A^Q(q, i), \delta_A^O(q, i) \ \right) \ \right) \quad \Longleftrightarrow \quad q \xrightarrow[o]{i} q'$$

□

Les programmes réels sont normalement déterministes. On s'attend à ce qu'ils répondent toujours de la même manière à des suites d'événements extérieurs identiques. Il est généralement admis queΓdans les systèmes critiquesΓtout forme de non déterminisme est à proscrire.

Les automates non déterministes sont cependant très utilisés pour spécifier des propriétés ou des comportements qui ne sont pas entièrement connus (ils peuvent par exemple représenter une abstraction d'un autre automateΓou un modèle d'un environnement physique).

1.1.3 Opérations sur les automates

1.1.3.1 Pré et postcondition

Soit $A = (Q_A, q_A^0, I_A, O_A, \delta_A)$ un automate. Soit $\mathcal{Q} \subseteq Q_A$ un ensemble d'étatsΓet $\Gamma \subseteq Q_A \times 2^{I_A}$ un ensemble de configurations.

Définition 1.6 :

La fonction de précondition *pre* calcule l'ensemble des configurations prédécesseurs d'un ensemble d'états donné.

$$pre(\mathcal{Q}) \quad = \quad \{ \ (q, i) \in Q_A \times 2^{I_A} \mid \exists q' \in \mathcal{Q}, \exists o \in 2^{O_A}, \ q \xrightarrow[o]{i} q' \ \}$$

La fonction de postcondition *post* calcule l'ensemble des états successeurs d'un ensemble de configurations donné.

$$post(\Gamma) \quad = \quad \{ \ q' \in Q_A \mid \exists (q, i) \in \Gamma, \exists o \in 2^{O_A}, \ q \xrightarrow[o]{i} q' \ \}$$

□

1.1.3.2 Projection

Soient $A = (Q_A, q_A^0, I_A, O_A, \delta_A)$ un automateΓet $O' \subseteq O_A$.

Définition 1.7 :

La projection $A \downarrow O'$ de l'automate A est l'automate $(Q_A, q_A^0, I_A, O', \delta'_A)$ où

$$\forall (q, i, o', q') \in Q_A \times 2^{I_A} \times 2^{O'} \times Q_A,$$
$$(q, i, o', q') \in \delta'_A \quad \Longleftrightarrow \quad \exists o \in 2^{O_A \setminus O'}, \ (q, i, o' \cup o, q') \in \delta_A$$

□

La projection conserve la réactivité et le déterminisme.

IntuitivementΓun système peut posséder des signaux internesΓinvisibles de l'extérieur. La fonction de projection permet de calculer les traces "externes" du processus (dans lesquels les signaux internes n'interviennent pas).

1.1.3.3 Produit synchrone

Le produit synchrone est l'opération de base pour faire communiquer des automates.

Définition 1.8 :

Soient $A_1 = (Q_1, q_1^0, I_1, O_1, \delta_1)$ et $A_2 = (Q_2, q_2^0, I_2, O_2, \delta_2)$ deux automates. Leur produit synchrone $A_1 \times A_2$ est un automate

$$A_1 \times A_2 = (Q_1 \times Q_2, (q_1^0, q_2^0), (I_1 \setminus O_2) \cup (I_2 \setminus O_1), (O_1 \setminus I_2) \cup (O_2 \setminus I_1), \delta_{12})$$

où

$$(q_1, q_2) \xrightarrow[o]{i} (q_1', q_2') \quad - \quad \exists l_1 \in I_1 \cap O_2, \quad \exists l_2 \in I_2 \cap O_1,$$

$$q_1 \xrightarrow[(l_2 \cup o) \cap O_1]{l_1 \cup (i \cap I_1)} q_1'$$

$$q_2 \xrightarrow[(l_1 \cup o) \cap O_2]{l_2 \cup (i \cap I_2)} q_2'$$

□

Intuitivement, ce produit synchrone est équivalent à "connecter" les entrées/sorties de A_1 avec celles de A_2 portant le même nom. Dans la section 7.2.4, page 78, nous présenterons une technique de renommage permettant de généraliser le produit synchrone et de modéliser des "connections" quelconques.

1.1.4 Circuits booléens

1.1.4.1 Méthodes énumérative et symbolique

La première technique d'implémentation des automates est dite énumérative. Elle consiste à coder la fonction de transition explicitement dans un tableau (ou équivalent). De nombreux algorithmes efficaces ont été développés et implémentés sur ce genre de structure (par exemple dans les outils ALDEBARAN [FGK+96] ou CONCURRENCY WORKBENCH [CPS89]). La partie III décrit une telle implémentation (BANG [Les97a]).

Le principal défaut de cette implémentation est l'espace nécessaire pour stocker explicitement un grand nombre d'états et de transitions. Pour pallier à ce problème, des méthodes symboliques ont été proposées, qui permettent de représenter la fonction de transition par une formule. Si on se limite au cas booléen, cette formule peut se coder de façon efficace à l'aide de BDD ("Binary Decision Diagram"). Par contre, certains algorithmes, comme l'abstraction (voir chapitre 13, page 151) ou la minimisation, peuvent être moins performants en symbolique qu'en énumératif.

Dans la section suivante, nous présentons un modèle de circuits booléens [Hal94]. Les langages synchrones LUSTRE, ESTEREL et ARGOS peuvent être aisément compilés dans ce format.

1.1.4.2 Définition

Soit $\mathbb{B} = \{0, 1\}$ l'ensemble des booléens. Nous utiliserons par la suite les équivalences classiques entre les ensembles, les fonctions booléennes et les formules booléennes. A toute formule booléenne B sur un ensemble $\{b_0, b_1, \ldots, b_{n-1}\}$ de variables booléennes sont associés :

1. une fonction booléenne (notée également B) de $\mathbb{B}^n$ dans $\mathbb{B}$ définie par

   $$\forall \mathbf{b} \in \mathbb{B}^n, \; B(\mathbf{b}) \; = \; B[b_0/\mathbf{b}_0, \ldots, b_{n-1}/\mathbf{b}_{n-1}]$$

   où $B[b_0/\mathbf{b}_0, \ldots, b_{n-1}/\mathbf{b}_{n-1}]$ est égale à la valeur de la formule B dans laquelle on a remplacé, pour tout i, la variable booléenne b_i par la valeur $\mathbf{b}_i$ de la $i^{\text{ème}}$ composante du vecteur $\mathbf{b}$.

2. un sous-ensemble de $\mathbb{B}^n$ (noté également B) défini par

$$\forall \mathbf{b} \in \mathbb{B}^n,\ \mathbf{b} \in B \iff B(\mathbf{b}) = 1$$

On peut maintenant définir les circuits booléens.

Définition 1.9 :

Un circuit booléen C à n variables d'étatΓm variables d'entrée et k variables de sortie est un tuple $(Init_C, \delta_C^Q, \delta_C^O)$ où

- Une configuration du circuit est un couple $(q, i) \in \mathbb{B}^n \times \mathbb{B}^m$.
  $q \in \mathbb{B}^n$ est un vecteur de valeurs booléennes appelé état du circuit. Nous dirons que le circuit se trouve dans l'état q siΓpour tout j tel que $0 \leq j \leq n-1\Gamma$la $j^{\text{ème}}$ variable d'état du circuit a pour valeur la $j^{\text{ème}}$ composante du vecteur q.
  $i \in \mathbb{B}^m$ est un vecteur de valeurs booléennes appelé événement d'entrée du circuit. IntuitivementΓchaque composante i_j de i correspond à un signal d'entrée. Le signal est dit présent si la valeur de cette composante est à vraiΓet absent dans le cas contraire. Nous dirons que i est l'entrée courante du circuit siΓpour tout j tel que $0 \leq j \leq m-1\Gamma$la $j^{\text{ème}}$ variable d'entrée du circuit a pour valeur la $j^{\text{ème}}$ composante du vecteur i.
- $Init_C \subseteq \mathbb{B}^n$ est une formule booléenne décrivant l'ensemble des états initiaux. On se limite en général à un seul état initial : $Init_C$ est alors un monôme complet des variables d'état.
- $\delta_C^Q \in \mathbb{B}^{n+m} \to \mathbb{B}^n$ est la fonction de transition.
  Si $(q, i) \in \mathbb{B}^{n+m}$ est une configuration du circuitΓl'état suivant est $\delta_C^Q(q, i)$.
- $\delta_C^O \in \mathbb{B}^{n+m} \to \mathbb{B}^k$ est la fonction de sortie.
  Si $(q, i) \in \mathbb{B}^{n+m}$ est une configuration du circuitΓle circuit émet l'événement de sortie $\delta_C^O(q, i)$. L'événement de sortie est interprété de la même manière qu'un événement d'entrée.
  Par la suiteΓil sera également utile de considérer δ_C^Q et δ_C^O comme un seul vecteur de $n+k$ fonctions de $\mathbb{B}^{n+m}$ dans $\mathbb{B}$. On note $\delta_C^j\Gamma$pour $0 \leq j \leq n-1\Gamma$la $j^{\text{ème}}$ composante du vecteur $\delta_C^Q = [\delta_C^j]_{0 \leq j \leq n-1}\Gamma$et pour $n \leq j \leq n+k-1\Gamma$la $(j-n)^{\text{ème}}$ composante du vecteur $\delta_C^O = [\delta_C^j]_{n \leq j \leq n+k-1}$. Si $(q, i) \in \mathbb{B}^{n+m}\Gamma$on peut donc écrire $\delta_C(q, i) = (\delta_C^0(q, i), \ldots, \delta_C^{n+k-1}(q, i))$.

Comme précédemmentΓnous pouvons décrire des propriétés de vivacité en introduisant la formule booléenne $F_C \subseteq \mathbb{B}^n\Gamma$décrivant l'ensemble des états finals. □

Notation : Un ensemble de configurations du circuit est un ensemble de couples formés d'un état et d'un événement d'entrée. Il peut être représenté par une formule booléenne des variables d'état et des variables d'entréeΓou de façon équivalente par une fonction booléenne. Par la suiteΓ nous utiliserons indifféremment ces trois notations. AinsiΓsi Γ est un ensemble de configurations et (q, i) une configurationΓla proposition

(q, i) est élément de Γ.

peut s'écrire

$$\left(\ (q,i) \in \Gamma\ \right) \iff \left(\ (q \wedge i) \Rightarrow \Gamma\ \right) \iff \left(\ \Gamma(q,i) = 1\ \right)$$

Un circuit booléen est une implémentation d'un automate déterministe. Les définitions concernant les automates s'appliquent donc de la même manière.

La plupart des fonctions de base sur les circuits booléens ont été implémentées dans l'outil BAC de Halbwachs ([Hal94]) : les pré et postconditionsΓl'accessibilitéΓetc ... Dans la partie IIIΓune amélioration de l'algorithme de calcul de précondition sera présentée au chapitre 14Γpage 159. Un nouvel algorithme de calcul des abstractions de signaux (voir section 7.2.3Γpage 77) sera présenté chapitre 13Γpage 151.

1.2 Expression d'une propriété par un observateur synchrone

1.2.1 Expression d'une propriété

Il existe plusieurs façons d'exprimer formellement des propriétés sur des systèmes de transitions.

1. Par comparaison de structure :

   Les propriétés sont exprimées à l'aide d'un automate (non déterministe). On définit alors des relations de simulation sur les structures des automates pour vérifier qu'une implémentation satisfait la propriété.

2. Par inclusion de traces :

   Ce cas est un sous-cas du précédent. Une propriété est modélisée par l'ensemble des traces d'un automate. Un processus satisfait cette propriété si le langage de ses traces y est inclus.

   On rappelle que l'inclusion des traces est équivalente à la simulation forte lorsque les automates considérés sont déterministes.

3. Par utilisation d'une logique :

   Les traces d'un processus sont exprimées à l'aide de formules de logique temporelle sur le modèle. Les logiques les plus connues sont PTL (logique linéaire) et CTL (logique arborescente).

Nous allons plus particulièrement nous intéresser à l'inclusion des traces. Nous pouvons distinguer deux types de propriétés : les propriétés de sûreté et les propriétés de vivacité.

Une propriété de sûreté exprime que "quelque chose de mauvais" ne se produit jamais durant l'exécution du système. Par exempleΓl'exclusion mutuelle (voir chapitre 10Γpage 109) ou l'absence de blocage sont des propriétés de sûreté. Une propriété de vivacité exprime que "quelque chose de bon" se produit immanquablement durant l'exécution du système. Par exempleΓla terminaisonΓ l'absence de famine ou la garantie de service sont des propriétés de vivacité.

Définition 1.10 :

- Une propriété de sûreté ("safety property") φ sur l'ensemble de signaux X est un sous-ensemble préfixe-clos de X^∞Γi.e.Γtelle que la condition suivante est vérifiée

  $$\tau \in \varphi \quad - \quad (\ \tau' \in \varphi \text{ pour tout préfixe fini } \tau' \text{ de } \tau.\ )$$

- Une propriété φ sur X est une propriété de vivacité ("liveness property") si et seulement si

  Pour tout trace finie τΓil existe τ' telle que $\tau.\tau' \in \varphi$

□

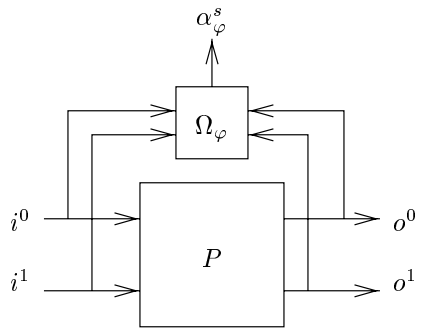


FIG. 1.1 – L'observateur d'une propriété

1.2.2 Vérification

Soit A un automate et φ une propriété. φ est donc modélisée par un ensemble de traces T_φ. Vérifier que l'automate A satisfait la propriété φ revient donc à montrer que l'ensemble des traces de A est inclus dans T_φ.

$$T_A \subseteq T_\varphi$$

Notons $\overline{T_\varphi} = X^\infty \backslash T_\varphi$ le complémentaire de T_φ. Classiquement l'inclusion précédente est remplacée par

$$T_A \cap \overline{T_\varphi} = \emptyset$$

1.2.3 Observateur d'une propriété

Cette section montre comment une propriété peut être spécifiée à l'aide d'un observateur synchrone [HLR93]. Un observateur synchrone est un processus capable d'observer le comportement d'un autre processus sans modifier ce comportement. Il prend en entrée les signaux d'entrée et de sortie du processus observé (appelés signaux de communication) et émet un ou des signaux d'alarme caractérisant la propriété (voir figure 1.1).

1.2.3.1 Observateur d'une propriété de sûreté

L'observateur d'une propriété de sûreté φ sur X peut être modélisé par un automate

$$\Omega_\varphi = (Q_\varphi, q^0_\varphi, X, \{\alpha^s_\varphi\}, \delta_\varphi)$$

où α^s_φ est un nouveau signal ($\alpha^s_\varphi \notin X$) appelé signal d'alarme émis lorsque la trace des événements d'entrée ne vérifie pas la propriété φ.

Propriété 1.1 :

Un automate A satisfait la propriété de sûreté φ si et seulement si la composition synchrone de A et de l'observateur de φ n'émet jamais le signal d'alarme α^s_φ. □

Remarque : Ces deux approches sont bien entendu équivalentes.

$$T_A \cap \overline{T_\varphi} = \emptyset \iff T_A \cap T_{\Omega_\varphi} \subseteq X^\infty$$

1.2.3.2 Observateur d'une propriété de vivacité

L'observateur d'une propriété de vivacité φ sur X peut être modélisé par un automate

$$\Omega_\varphi \ = \ (Q_\varphi, q^0_\varphi, X, \{\alpha^l_\varphi\}, \delta_\varphi)$$

où α^l_φ est un nouveau signal $(\alpha^l_\varphi \notin X)$Γappelé signal de vivacitéΓémis lorsque la trace des événements d'entrée conduit à un but de la propriété φ.

Propriété 1.2 :

> Un automate A satisfait la propriété de vivacité φ si et seulement si la composition synchrone de A et de l'observateur de φ émet infiniment souvent le signal de vivacité α^l_φ. □

1.2.3.3 Observateur d'une propriété : cas général

On utilise la propriété suivante

Propriété 1.3 :

> Toute propriété correspondant à un langage régulier est l'intersection d'une propriété de sûreté et d'une propriété de vivacité. □

Une propriété φ peut donc toujours être écrite sous la forme $\varphi = \varphi^s \cap \varphi^l$Γ où φ^s est une propriété de sûreté et φ^l est une propriété de vivacité. Un observateur est donc un automate $(Q_\varphi, q^0_\varphi, X, \{\alpha^s_\varphi, \alpha^l_\varphi\}, \delta_\varphi)$ où α^s_φ et α^l_φ sont respectivement le signal d'alarme et de vivacité.

On a l'habitude de considérer le produit de l'automate et de son observateur plutôt que l'automate seul. En d'autres termesΓon considère un programme intégrant une spécificationΓplutôt qu'un programme et une spécification séparée.

Ces définitions sontΓbien entenduΓéquivalentes à celle des automates de Büchi.

1.2.3.4 Observateur et circuit booléen

Un observateur peut être implémenté par un circuit booléen à deux sorties. Ces sorties peuvent être définies par une formule booléenne des états et des entrées du circuit. TraditionnellementΓon appelle invariant (noté $\mathcal{I}$) la négation de la formule définissant le signal d'alarme. Une propriété de sûreté est vraie si le signal d'alarme n'est jamais émisΓou inversement si l'invariant est toujours vrai.

Un observateur implémenté par un circuit booléen est donc un tuple $(Init_C, \delta_C, \mathcal{I}_C, F_C)$ où α^s_φ et α^l_φ sont respectivement émis dans les configurations de $\overline{\mathcal{I}_C}$ et F_C.

Remarque : Un circuit booléen est par définition une implémentation d'un automate déterministe. Les observateurs nous permettent de modéliser des comportements non déterministes.

1.2.4 Environnement et vérification

1.2.4.1 Pourquoi définir un environnement ?

La caractéristique principale des systèmes réactifs est qu'ils interagissent fortement avec leur environnement physique. Les propriétés de l'environnement doivent donc être fortement prises en compte pour concevoir et vérifier de tels systèmes.

AinsiΓle problème de la vérification peut souvent se poser de la façon suivante :

> Vérifier que dans l'environnement EΓle processus P satisfait la propriété φ.

$$E \times P \models \varphi$$

L'environnement peut être défini par

- Un autre processus: il suffit de vérifier que le produit synchrone de P et de ce processus satisfait la propriété.
- L'abstraction d'un processus complexe: il est rarement nécessaire de considérer l'environnement dans son ensemble. Seuls quelques aspects de son comportement suffisent à la satisfaction de la propriété.
- Un environnement physique: il est alors nécessaire de modéliser son comportement de manière formelle. Ce travail est en général une tâche complexe à laquelle on ne s'intéressera pas ici.

Dans les 2 derniers cas, l'environnement peut être décrit par un processus non déterministe, traduisant le fait qu'on ne s'intéresse qu'à certains comportements de cet environnement.

1.2.4.2 Modélisation de l'environnement par un observateur

Nous proposons ici de modéliser un environnement non déterministe par un observateur déterministe. Dans la suite, nous nous limiterons aux propriétés de sûreté sur l'environnement.

Considérons le produit synchrone $\Omega_E \times P \times \Omega_\varphi$.

Propriété 1.4 :

> Le processus P satisfait la propriété de sûreté φ dans l'environnement E si et seulement si pour toute exécution de $\Omega_E \times P \times \Omega_\varphi$ n'émettant pas le signal α^s_E, le signal α^s_φ n'est jamais émis. □

Il est à noter que cette propriété n'est pas équivalente à la proposition suivante:

> Le processus P satisfait la propriété de sûreté φ dans l'environnement E si et seulement si pour toute exécution de $\Omega_E \times P \times \Omega_\varphi$, dès que le signal α^s_φ est émis, le signal α^s_E a été déjà été émis au moins une fois dans le passé.

En effet, Ω_E peut être dans un état où le signal α^s_E n'est pas émis, mais où il le sera inéluctablement dans le futur. Un tel observateur est dit non extensible.

Définition 1.11 :

> Un observateur déterministe $\Omega_\varphi = (Q_\varphi, q^0_\varphi, X, \{\alpha^s_\varphi\}, \delta^Q_\varphi, \delta^O_\varphi)$ est dit extensible (ou "causal") si et seulement si
>
> $$\forall (q,x) \in Q_\varphi \times 2^X,\ \delta^O_\varphi(q,x) = \emptyset \quad \Longrightarrow \quad \exists x' \in 2^X, \delta^O_\varphi\left(\ \delta^Q_\varphi(q,x), x'\ \right) = \emptyset$$
>
> □

Un observateur est donc extensible si et seulement si, à partir de tout état n'émettant pas le signal α^s_φ, il existe une exécution infinie de l'automate ne l'émettant pas. Un algorithme permettant de rendre extensible un observateur sera proposé section 3.2.1.1, page 35.

1.2.4.3 Les environnements des circuits booléens

Soient un automate A et les observateurs d'un environnement Ω_E et d'une propriété Ω_φ, implémentés par des circuits booléens. On se limite aux propriétés de sûreté sur l'environnement.

En considérant le produit de ces trois circuits, on obtient un nouveau circuit C possédant trois sorties supplémentaires:

- Le signal de sûreté de l'observateur de la propriété, dont l'absence est codé par l'invariant $\mathcal{I}_C$.

– Le signal de vivacité de l'observateur de la propriétéΓcodé par la finalité F_C.

– Le signal de sûreté de l'observateur de l'environnementΓdont l'absence est codée par l'assertion $\mathcal{A}_C$.

Si on se limite aux propriétés de sûretéΓon peut dire que

> L'automate A satisfait la propriété φ de sûreté dans l'environnement E si et seulement si pour toute exécution infinie du circuit C pour laquelle $\mathcal{A}_C$ reste toujours vraieΓ$\mathcal{I}_C$ reste également toujours vrai.

1.3 Vérification de propriétés

1.3.1 Principe

1.3.1.1 Ensemble des états accessibles

Soit $\Omega_\varphi = (Q_\varphi, q^0_\varphi, X, \{\alpha^s_\varphi\}, \delta_\varphi)$ la composition d'un programme et d'un observateur d'une propriété de sûreté φ. La vérification que Ω_φ n'émet jamais le signal d'alarme α_φ repose sur le calcul des états accessibles de l'automate.

Définition 1.12 :

> Un état $q \in Q_\varphi$ est dit accessible si et seulement s'il existe une exécution finie $(q^0_\varphi, q_1, q_2, \ldots, q)$ atteignant q. □

Soit $\mathcal{R} \subseteq Q_\varphi$ l'ensemble des états accessibles de Ω_φ. Cet ensemble est défini par un plus petit point fixe.

$$q^0_\varphi \in \mathcal{R}$$
$$\left(q \in \mathcal{R} \quad \text{et} \quad \exists i, o,\ q \xrightarrow[o]{i} q' \right) \quad \Rightarrow \quad q' \in \mathcal{R}$$

ou de manière équivalente en utilisant la fonction de postcondition définie section 1.1.3.1Γpage 16Γ

$$\begin{aligned} q^0_\varphi &\in \mathcal{R} \\ post(\mathcal{R}) &\subseteq \mathcal{R} \end{aligned}$$

L'ensemble des états accessibles $\mathcal{R}$ est donc calculé par itération

$$\begin{aligned} \mathcal{R}_0 &= \{q^0_\varphi\} \\ \mathcal{R}_{i+1} &= \mathcal{R}_i \cup post(\mathcal{R}_i) \\ \mathcal{R} &= \lim_{i \to \infty} \mathcal{R}_i \end{aligned}$$

Pour les systèmes à états finisΓauxquels nous nous sommes jusqu'à présent limitésΓce calcule converge. Pour les systèmes infinisΓil ne converge en général pas ($\mathcal{R}$ est souvent lui-même infiniΓ voir chapitre 6Γpage 59). Mais même dans le cas finiΓcette technique est limitée par l'explosion du nombre des états.

1.3.1.2 Méthode symbolique

Les circuits booléens permettent de calculer une représentation symbolique de l'ensemble des états accessiblesΓsans avoir à les énumérer explicitement. Chaque état est en effet codé par un monôme complet des variables d'étatΓet tout ensemble d'états par une formule booléenne des

variables d'état (voir section 1.1.4Γpage 17). Un algorithme de calcul de la postcondition a été développé dans [Rat92] et implémenté dans [Hal94] (voir chapitre 14Γpage 159). Il permet d'effectuer le calcul de point fixe précédent sur des formules booléennes : $\mathcal{R} \in \mathbb{B}^n \to \mathbb{B}$.

$$\begin{aligned} \mathcal{R}_0 &= q_\varphi^0 \\ \mathcal{R}_{i+1} &= \mathcal{R}_i \vee post(\mathcal{R}_i) \\ \mathcal{R} &= \lim_{i\to\infty} \mathcal{R}_i \end{aligned}$$

Pour des systèmes plus généraux Γdont la partie donnée contient par exemple des entiers (et plus seulement des booléens)Γle calcul exact des états accessibles est en général impossible. Différentes méthodes d'approximation ont été étudiées permettant de calculer un sur-ensemble des états accessibles. Dans le cas des contraintes linéaires Γl'outil Polka [HMP95ΓHPR94] calcule ainsi Γà l'aide de polyèdres Γun sur-ensemble du domaine que peuvent prendre ces valeurs.

Un autre axe de recherche consiste à utiliser des systèmes d'aide à la démonstration ("theorem prover")Γafin de construire un système abstrait Γsur lequel des propriétés de sûreté peuvent être vérifiées. Cette technique consiste à exprimer des relations entre des prédicats de façon à déterminer un sur-ensemble des états accessibles. Dans la section 11.4Γpage 132Γnous combinerons le calcul d'invariant Γqui est le sujet principal de ce document Γavec les techniques développées par Saïdi et Graf [Saï96ΓGS96]Γpour vérifier des réseaux paramétrés de processus infinis.

1.3.2 Vérification modulaire

Afin d'éviter l'explosion du nombre des états due au produit synchrone Γil semble naturel de décomposer le problème de la vérification en plusieurs sous problèmes plus simples Γet d'effectuer une preuve modulaire. Pour fixer les idées Γconsidérons deux processus P_1 et P_2 et une propriété φ. Afin d'éviter le calcul du produit synchrone $P_1 \times P_2$ Γune preuve modulaire consisterait à chercher Γ connaissant P_2 Γune propriété ψ nécessaire et suffisante que doit vérifier P_1 pour que $P_1 \times P_2$ satisfasse φ. En pratique Γil est rarement possible de trouver une propriété à la fois nécessaire et suffisante. Pour les problèmes de vérification Γon s'intéresse plutôt à une propriété suffisante. En notation tout à fait informelle Γcette propriété pourrait être notée

$$\psi \;=\; (P_2 \Rightarrow \varphi)$$

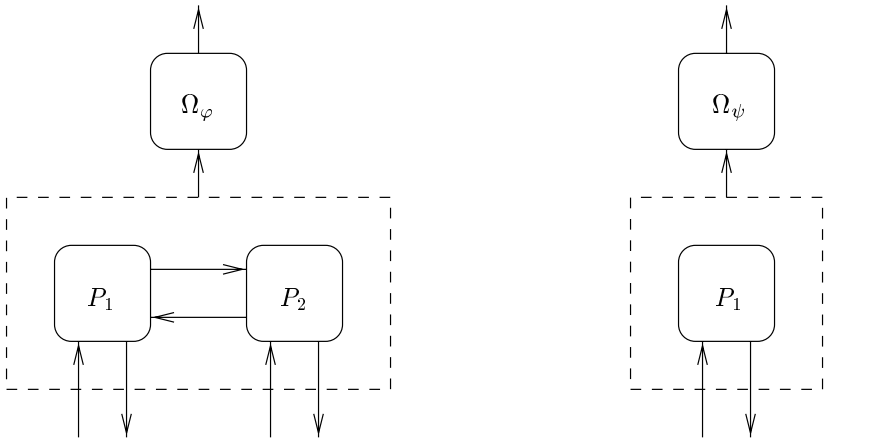
Il suffit alors de vérifier que $P_1 \models (P_2 \Rightarrow \varphi)$ Γpour que $P_1 \times P_2 \models \varphi$ (voir figure 1.2). Bien entendu Γ on espère que le coût du calcul de $P_2 \Rightarrow \varphi$ sera plus faible que celui du calcul direct.

L'intérêt des observateurs est de pouvoir exprimer les processus et les propriétés sur ces processus dans un même formalisme : aussi bien les processus que les propriétés sont modélisés par des automates. Dans le cas de la synthèse modulaire Γcela signifie que P_1 ΓP_2 Γφ et surtout $P_2 \Rightarrow \varphi$ sont des automates. La vérification modulaire peut donc se ramener au problème de la synthèse de spécification. Ce problème sera étudié dans la partie I de ce document.

1.3.3 Cas des réseaux réguliers

La vérification modulaire peut être aisément étendue à plus de 2 processus. Soient n processus P_1 ΓP_2 Γ... ΓP_n communicants et φ une propriété. Le problème de la vérification modulaire est alors le suivant :

> Connaissant les $n-1$ processus P_2 Γ... ΓP_n Γtrouver une propriété ψ_{n-1} nécessaire et/ou suffisante que doit vérifier P_1 pour que le produit synchrone
>
> $$P_1 \times P_2 \times \ldots \times P_n$$
>
> vérifie la propriété φ.



(a) Vérification du produit synchrone (b) Vérification modulaire

FIG. 1.2 – Vérification de propriétés sur des processus communicants

A la limiteΓcette idée peut être utilisée pour vérifier des propriétés sur des réseaux paramétrés de processusΓi.e.Γde vérifier une propriété sur n processusΓquelle que soit la valeur de n. AinsiΓsi on fait tendre n vers l'infini et qu'on définisse ψ par

$$\psi \;=\; \bigcup_n \psi_n$$

on obtient le résultat suivant :

$$(\ \forall n,\ P_n \ \models \psi\ ) \quad \Longrightarrow \quad (\ \forall n,\ P_1 \times \ldots \times P_n \ \models\ \varphi\ )$$

Ce résultat est appelé principe d'inductionΓet ψ est appelé un invariant du réseau.

Plus formellementΓconsidérons un ensemble de k processus $\{P_1, \ldots, P_k\}$ et la famille de processus $\mathcal{F}$ définie par

$$\forall i = 1 \ldots k, \quad (\ P_i \in \mathcal{F}\ ) \quad \text{et} \quad (\ P', P'' \in \mathcal{F} \ \Rightarrow\ P' \times P'' \in \mathcal{F}\ )$$

$\mathcal{F}$ est appelé un réseau binaire de processus (voir chapitre 9Γpage 99). Soit une propriété φ et soit $\preceq$ une relation d'ordre sur les processusΓcompatible avec φΓi.e.Γtelle que

$$(\ P' \ \preceq\ P'' \quad \text{et} \quad P'' \ \models\ \varphi\ ) \quad \Longrightarrow \quad (\ P' \ \models\ \varphi\ )$$

La preuve par induction consiste à déterminer un processus IΓappelé invariant du réseauΓtel que

$$\begin{array}{rrcl} \forall i = 1 \ldots k, & P_i & \preceq & I \\ & I \times I & \preceq & I \end{array}$$

Proposition 1.1 :

(Principe d'induction)

$$(\ I \ \models\ \varphi\ ) \quad \Longrightarrow \quad (\ \forall P \in \mathcal{F},\ P \ \models\ \varphi\ )$$

□

Toute la difficulté de cette technique réside à déterminer l'invariant I. La partie II de ce document présentera l'état de l'art dans le domaine et proposera une nouvelle technique de calcul d'invariants à base de points fixes.

Première partie

Synthèse de processus

Chapitre 2

Introduction à la synthèse de processus

2.1 Problème de la synthèse

Certains aspects de la synthèse des processus synchrones ont été traités dans [Les94].

2.1.1 Vérification modulaire

Revenons sur le problème de la vérification de propriétés sur un processusΓtel qu'il a été posé section 1.2.4Γpage 21.

> Le processus P satisfait-il la propriété φ dans l'environnement E?
>
> $$E \times P \models \varphi \quad ?$$

Le processus P est généralement généré par un compilateurΓl'environnement E peut être soit un autre processusΓsoit une modélisation d'un environnement physique.

Les algorithmes actuelsΓqui fonctionnent selon le principe du calcul des états accessiblesΓne permettent pas de vérifier des propriétés sur des programmes de grande taille. Dans la section 1.3.2Γ page 24Γnous proposonsΓpour pallier à ce problèmeΓd'effectuer des preuves modulaires .

Supposons que le processus P puisse se décomposer en deux processus P_1 et P_2 :

$$P = P_1 \times P_2$$

La preuve modulaire consiste à

1. Déterminer une propriété φ_2 qu'il est nécessaire et suffisant que P_2 satisfasseΓpour que le produit $E \times P_1 \times P_2$ satisfasse la propriété φ.
2. Vérifier que le processus P_2 satisfait φ_2.

Le chapitre 3 va proposer des algorithmes symboliques permettant le calcul d'une propriété φ_2.

2.1.2 Synthèse de processus

Plus généralementΓle problème de la synthèse est posé de la manière suivante: un environnement E précis est définiΓune propriété φ assez stricte est spécifiée. Au lieu d'écrire une implémentation de P et de vérifier qu'elle satisfait φ dans l'environnement EΓon cherche à synthétiser le processus le plus général vérifiant cette propriété:

> Étant donnés un environnement E et une propriété φΓquel est le processus P le plus généralΓsatisfaisant φ sous l'environnement E?

La synthèse permet ainsi de déterminer des stratégies gagnantes dans des problèmes de jeuxΓ ou de dériver un programme à partir d'une spécification.

Si on suppose que les processus E et P peuvent générer respectivement les signaux x_E et x_PΓ ce problème peut être envisagé de plusieurs manières :

- Dans la théorie des jeuxΓE et P sont vus comme deux joueurs choisissant chacun à leur tour une valeur pour x_E et x_PΓrespectivement. E joue le premier. Le but du joueur E est de parvenir à la violation de $\varphi(x_E, x_P)$. Celui du joueur P est de contrer les ambitions de E : à chaque coup de EΓP doit déterminer une stratégie pour ne pas perdreΓi.e.Γpour préserver $\varphi(x_E, x_P)$.
- Le paradigme de Skolem est basé sur l'observation que la formule $(\forall x_E)(\exists x_P)\varphi(x_E, x_P)$ est équivalente à la formule du second ordre $(\exists f)(\forall x_E)\varphi(x_E, f(x_E))$. La synthèse de P peut donc se ramener à la détermination de la fonction f.
- Dans la théorie des superviseurs (voir [RW87ΓRW89] ainsi que l'implémentation qui en a été faite dans [HW92])ΓP n'est pas un programme du même type que E. E est décrit par un automate recevant des événements discrets (DES pour "Discrete Event Systems"). P est un superviseur qui peut autoriser ou interdire certaines transitions de E. Le problème est alors de synthétiser un superviseur P tel que E supervisé par P satisfait la spécification φ.

Ce dernier point est devenu un classique de la synthèse. Dans la section suivanteΓla théorie des superviseurs va être rappelée.

2.2 Générateur et superviseur (théorie de Ramadge et Wonham)

Cette section présente les bases de la théorie des processus à événements discrets (ou DES pour "Discrete Event Systems")Γdécrite dans [RW87ΓRW89].

Les processus à événements discrets sont des systèmes discretsΓasynchrones (ils n'entrent donc pas dans le cadre synchrone de notre modèle)Γet éventuellement non déterministes. Les signaux d'entrée et de sortie ne sont pas distingués.

2.2.1 Contrôle des processus à événements discrets

Soit $A = (Q_A, q_A^0, X_A, \emptyset, \delta_A^Q, F_A)$ un automate (qui est plutôt appelé générateur dans cette théorie) sans signaux de sortie et où δ_A^Q est une fonction non totale. On dit que dans l'état $q \in Q_A$Γ le signal x est accepté si $\delta_A^Q(q, x)$ est défini. Cette vision est équivalente à celle des observateurs : un signal d'alarme est émis lorsque la fonction de transition n'est pas définie.

L'ensemble X des signaux est partitionné en signaux contrôlables et signaux incontrôlables.

$$X = X_c \cup X_{nc} \qquad X_c \cap X_{nc} = \emptyset$$

Une fonction de contrôle ("control pattern") est une fonction

$$\gamma : X \to \mathbb{B}$$

Un signal x est autorisé si $\gamma(x) = 1$Γet interdit si $\gamma(x) = 0$. On a donc $\forall x \in X_{nc}$, $\gamma(x) = 1$ (tous les signaux incontrôlables sont autorisés). γ permet de définir une nouvelle fonction de transitionΓ dite fonction de transition contrôlée

$$\delta_c^Q : \left(X \to \mathbb{B} \right) \times Q_A \times X \to Q_A$$

avec

$$\delta_c^Q(\gamma, q, x) = \begin{cases} \delta_A^Q(q, x) & \text{si } \delta_A^Q(q, x) \text{ est définie et } \gamma(x) = 1 \\ \text{indéfini} & \text{sinon} \end{cases}$$

Alors l'automate

$$A_c \quad = \quad (Q_A, q_A^0, X, \emptyset, \delta_c^Q, F_A)$$

est une version de A admettant un contrôle externe. Il est appelé processus à événements discrets contrôlés (où CDEP pour "Controlled Discrete Event Process"). Le contrôle consiste alors à déterminer pour chaque état q une fonction de contrôle γ telle que le processus contrôlé A_c vérifie la propriété recherchée.

2.2.2 Superviseur

Un superviseur ("supervisor") est un couple

$$\mathcal{S} \quad = \quad (S, \Phi)$$

où

$$S = (Q_S, q_S^0, X, \emptyset, \delta_S^Q, F_S)$$

est un automate déterministe et où

$$\Phi \quad : \quad Q_S \rightarrow (X \rightarrow \mathbb{B})$$

est une fonction de contrôle appelée "state feedback map". Pour tout $q \in Q_S$Γ

$$\gamma \quad = \quad \Phi(q) \in \left(\; X \rightarrow \mathbb{B} \; \right)$$

est une fonction de contrôle. On peut alors coupler $\mathcal{S}$ et A_c. On définit ainsi le processus à événements discrets supervisé (SDEP pour "Supervised Discrete Event Process") comme étant

$$\mathcal{S}/A_c \quad = \quad (Q_S \times Q_A, (q_S^0, q_A^0), X, \emptyset, \delta_S^Q \times \delta_c^Q, F_S \times F_A)$$

2.2.3 Problème de la synthèse

Étant donné deux langages T_1 et T_2 sur XΓle problème de la synthèse revient à déterminer un superviseur $\mathcal{S}$ tel que

$$T_1 \; \subseteq \; T_{\mathcal{S}/A_c} \; \subseteq \; T_2$$

Dans [RW87ΓRW89]Γdes conditions nécessaires et suffisantes sont données sur T_1 et T_2 pour que le problème soit soluble.

Le principal défaut de cette technique est d'utiliser des algorithmes énumératifsΓqui peuvent provoquer une explosion du nombre des états. Selon le modèle de la section 1.3.1.2Γpage 23Γnous allons présenter dans la suite des algorithmes symboliques qui permettent d'éviter la construction du graphe des états.

2.2.4 Commentaires

Dans notre approcheΓnous utilisons des communications synchrones et des processus à entrée/sortie. AinsiΓles signaux incontrôlables et contrôlables correspondent respectivement aux signaux d'entrée et de sortie de nos processus. La fonction de transition contrôlée correspond à la fonction de sortie du processus (i.e.Γla fonction qui calcule les valeurs à donner aux sorties)Γet le SDEP correspond au processus synthétisé lui-même.

Ramadge et Wonham donnent des résultats d'existence sur les langages. Les preuves de ces résultats fournissent des algorithmes énumératifs. Nous nous proposons dans la suite de définir des algorithmes ne raisonnant plus sur les langages mais sur les ensembles de configurations. Ces algorithmes peuvent alors être implémentés de façon efficaces pour les circuits booléens.

2.3 Observateur et synthèse

Nous proposons d'utiliser les observateurs synchrones (voir section 1.2Γpage 19) pour spécifier et résoudre le problème de la synthèse. Par la suiteΓces observateurs seront implémentés de façon efficace à l'aide de circuits booléens.

Soient Ω_E et Ω_φ des observateurs respectivement d'une assertion sur l'environnement et d'une propriété. Notre but est de générer l'ensemble des programmes P vérifiant la propriété φ lorsqu'ils fonctionnent dans un environnement E vérifiant l'assertionΓet tels que

- Le programme P est réactif: comme on s'intéresse aux exécutions infiniesΓnotre solution ne doit pas se bloquer. A chaque fois que P émet un événement de sortieΓil doit être sûr de pouvoir satisfaire la propriété φ dans le futur (à conditionΓbien sûrΓde fonctionner dans un environnement correct).
- Le programme P est déterministe: P doit être un programme réelΓqui réagit toujours de la même manière à des séquences d'événements d'entrée identiques.

Dans le chapitre 3Γnous proposerons un algorithme pour calculer l'observateur le plus général de la solution du problème de synthèse. Nous en déduirons un programme exécutable (i.e.Γ déterministe) dans le chapitre 4.

Chapitre 3

Synthèse d'observateurs

Le problème de la synthèse de systèmes réactifs va être résolu en deux étapes. Dans ce chapitre nous allons calculer un observateur décrivant la solution non déterministe la plus générale. Dans le chapitre suivantΓnous calculerons explicitementΓà l'aide d'oraclesΓl'ensemble des programmes déterministes solution de ce problème.

Nous nous limitons à des propriétés de sûreté sur l'environnement : l'observateur de l'environnement Ω_E ne peut émettre qu'un seul signal d'alarmeΓα^s_EΓlorsque l'hypothèse (ou assertion) sur l'environnement est violée. Nous supposerons de plus que les observateurs sont extensibles (voir section 1.2.4.2Γpage 22).

3.1 Problème et solution de la synthèse

3.1.1 Spécification

Soient Ω_E et Ω_φ respectivement un observateur d'un environnement (donc d'une propriété de sûreté) et un observateur d'une propriété φ.

Définition 3.1 :

On appelle spécification le produit synchrone $\Omega_E \times \Omega_\varphi$Γnoté $\mathcal{S}_{E,\varphi}$. □

Quand il n'y aura pas d'ambiguïté sur la spécification considéréeΓnous noterons souvent $\mathcal{S}$ à la place de $\mathcal{S}_{E,\varphi}$. Notre but étant d'implémenter la synthèse de processus de façon symbolique (i.e.Γ pour les circuits booléens)Γnous ne nous intéresserons qu'aux spécifications reconnues par un observateur déterministe. Bien entenduΓcela n'implique pas que l'environnement et la propriété soient déterministes. Dans le reste de ce chapitreΓnous considérerons la spécification

$$\mathcal{S} = (Q, q^0, I \cup O, \{\alpha^s_E, \alpha^s_\varphi, \alpha^l_\varphi\}, \delta^Q, \delta^O)$$

Une spécification prend en entrée les signaux d'entrée et de sortie du systèmeΓet émet en sortie les trois signaux d'alarme α^s_EΓα^s_φ et α^l_φ : ces signaux sont émis respectivement lorsque l'hypothèse sur l'environnement est violéeΓla propriété de sûreté est violéeΓun état marqué est atteint (voir section 1.2.1Γpage 19Γet figure 3.1). Ces trois signaux permettent de définir respectivement 3 ensembles de configurations :

– L'assertion $\mathcal{A}$ sur l'environnement où le signal d'alarme α^s_E de l'environnement n'est pas émis.

$$\mathcal{A} \;=\; \{\; (q,i,o) \in Q \times 2^I \times 2^O \mid \alpha^s_E \notin \delta^O(q,i,o) \;\}$$

$\mathcal{A}$ représente l'ensemble des configurations décrivant un comportement correct de l'environnement.

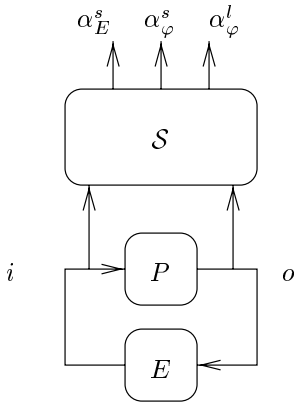


FIG. 3.1 – Une spécification

– L'invariant $\mathcal{I}$ où le signal d'alarme α_φ^s de la propriété n'est pas émis.

$$\mathcal{I} \;=\; \{\; (q,i,o) \in Q \times 2^I \times 2^O \mid \alpha_\varphi^s \notin \delta^O(q,i,o) \;\}$$

– La finalité F qui émet le signal de vivacité α_φ^l de la propriété.

$$F \;=\; \{\; (q,i,o) \in Q \times 2^I \times 2^O \mid \alpha_\varphi^l \in \delta^O(q,i,o) \;\}$$

Un circuit booléen implémentant la spécification $\mathcal{S}$ possède donc trois sortiesΓdéfinies par les formules booléennes $\mathcal{A}\Gamma\mathcal{I}$ et F.

3.1.2 Observateur solution

En généralΓla synthèse produit un automate non déterministeΓi.e.Γla spécification n'est pas assez forte pour déterminer complètement le processus solution. Dans ce chapitreΓnous nous proposons de représenter ce processus non déterministe à l'aide d'un observateur.

L'observateur Ω_PΓde la solution du problème de synthèseΓest construit à partir de $\mathcal{S}$ (il hérite de sa structure de contrôleΓi.e.Γ$\delta_P^Q = \delta^Q$). La synthèse de l'observateur de P revient alors à déterminer la fonction de sortie δ_P^OΓi.e.Γles deux ensembles de configurations $\mathcal{I}_P$ (qui n'émet pas le signal α_P^s) et F_P (qui émet le signal α_P^l).

La propriété φ va être récrite comme l'intersection d'une propriété de sûreté et d'une propriété de vivacité (voir section 1.2.1Γpage 19).

3.1.2.1 Propriété de sûreté

IntuitivementΓΩ_P émet son signal d'alarme dès que $\mathcal{S}$ émet le signal α_φ^s sans le signal α_E^s. En terme d'ensemble de configurationsΓtoute exécution de Ω_P peut

– soit rester dans l'ensemble de configurations $\mathcal{I}$ (la propriété φ est alors satisfaite)Γ

– soit sortir de l'ensemble $\mathcal{A}$ (auquel cas l'assertion sur l'environnement est violéeΓle comportement du processus devenant quelconque).

3.1.2.2 Propriété de vivacité

De même que précédemmentΓΩ_P émet le signal de vivacité α_P^l si $\mathcal{S}$ émet son signal de vivacité α_φ^l sans le signal α_E^s. Infiniment souventΓΩ_P doit donc être forcé à atteindre un état émettant le signal de vivacité. Cela nous amènera à définir une notion d'accessibilité et une notion d'accessibilité forte.

Nous allons ainsi définir deux ensembles de configurations :

- F_P^s est l'ensemble de configurations maximal depuis lequel il est toujours possible de joindre une configuration de FΓquel que soit le comportement de l'environnement. F_P^s doit vérifier la propriété

  $$\forall\Box\exists\Diamond F$$

  Si une exécution du processus sort de F_P^sΓil existe un comportement de l'environnement empêchant le processus d'atteindre un état final. La propriété de vivacité peut alors être violée.

- F_P^l est un ensemble de configurations conduisant dans une configuration de FΓpour tout comportement de l'environnement. Si les sorties du processus sont choisies de façon à demeurer dans F_P^lΓle processus atteindra un état final en un temps fini. F_P^l doit vérifier la propriété

  $$\forall\Diamond F$$

IntuitivementΓtant que nous restons dans F_P^sΓil est possible de forcer le système à émettre le signal de vivacité. Si nous restons dans F_P^lΓle système émettra inéluctablement le signal de vivacité.

Les sections suivantes proposent des algorithmes pour déterminer les trois ensembles de configurationsΓ$\mathcal{I}_P$ΓF_P^s et F_P^l.

3.2 Propriétés des ensembles de configurations

Les trois ensembles de configurations $\mathcal{I}_P$Γ F_P^s et F_P^l peuvent être définis comme étant les ensembles de configurations de taille maximale satisfaisant certaines propriétés :

- $\mathcal{I}_P$ doit vérifier une propriété d'invariance.
- F doit être fortement accessible depuis F_P^l.
- F doit être accessible depuis F_P^s.

Ces trois ensembles doivent de plus satisfaire des propriétés de compatibilité vis-à-vis de l'assertion.

3.2.1 Propriétés de l'environnement

3.2.1.1 Assertion sur l'environnement extensible

Il est tout d'abord nécessaire que l'assertion sur l'environnement soit extensible. La définition de l'extensibilité (voir section 1.2.4.2Γpage 22) s'écrit simplement à l'aide de la fonction de précondition preΓdéfinie respectivement pour les ensembles d'états et les ensembles de configurationsΓ par

$$pre(\mathcal{Q}) = \{\ (q,i,o) \in Q \times 2^I \times 2^O \mid \exists q' \in \mathcal{Q},\ \text{tel que } q \xrightarrow[o]{i} q'\ \}$$

$$pre(\Gamma) = \{\ (q,i,o) \in Q \times 2^I \times 2^O \mid \exists (q',i',o') \in \Gamma,\ \text{tel que } q \xrightarrow[o]{i} q'\ \}$$

$\mathcal{A}$ est extensible si et seulement si

$$\mathcal{A} \subseteq pre(\mathcal{A})$$

Propriété 3.1 :

Il existe un ensemble maximal de configurations inclus dans $\mathcal{A}$ qui est extensible. Il est égal au plus grand point fixe de la fonction monotone des ensembles de configurations dans les ensembles de configurations définie par

$$\lambda\Gamma.\mathcal{A} \cap pre(\Gamma)$$

□

Nous supposerons dorénavant que l'assertion $\mathcal{A}$ est extensible.

3.2.1.2 Compatibilité avec l'environnement

Soit $q \in Q$ un état.

Définition 3.2 :

L'événement d'entrée/sortie (i, o) est dit compatible avec l'assertion $\mathcal{A}$ dans l'état q si $(q, i, o) \in \mathcal{A}$. L'événement d'entrée i est dit compatible avec l'assertion $\mathcal{A}$ dans l'état $q\Gamma$s'il existe un événement de sortie o (une réaction du système)Γtel que (i, o) soit compatible avec l'assertion en q (i.e.$\Gamma(q, i, o) \in \mathcal{A}$). □

Notons $in^{\mathcal{A}}(q)$ l'ensemble des événements d'entrée compatibles avec l'assertion en q.

$$in^{\mathcal{A}}(q) \;=\; \{\; i \in 2^I \mid \exists o \in 2^O,\; (q, i, o) \in \mathcal{A} \;\}$$

Ces définitions sont étendues aux séquences. Une séquence $(\; (i_0, o_0), (i_1, o_1), \ldots, (i_n, o_n), \ldots \;)$ d'événements d'entrée/sortie est dite compatible avec l'assertion $\mathcal{A}$ en $q\Gamma$s'il existe une séquence $(q_0 = q, q_1, \ldots, q_n, \ldots)$ d'états telle que pour tout $n \geq 0\Gamma\, q_n \xrightarrow[o_n]{i_n} q_{n+1}$ et $(q_n, i_n, o_n) \in \mathcal{A}$. Une séquence $(i_0, i_1, \ldots, i_n, \ldots)$ d'événements d'entrée est dite compatible avec l'assertion $\mathcal{A}$ en $q\Gamma$ s'il existe une séquence $(o_0, o_1, \ldots, o_n, \ldots)$ d'événements de sortie telle que la séquence $(\; (i_0, o_0), (i_1, o_1), \ldots, (i_n, o_n), \ldots \;)$ d'événements d'entrée/sortie soit compatible avec l'assertion $\mathcal{A}$.

Notons de même $In^{\mathcal{A}}(q)$ l'ensemble des séquences d'événements d'entrée compatibles avec l'assertion en q.

$$\begin{aligned} In^{\mathcal{A}}(q_0) \;=\; \{\; (i_0, i_1, \ldots) \in 2^I \mid\; & \exists q_1, q_2, \ldots \in Q,\; \exists o_0, o_1, \ldots \in 2^O, \\ & \forall i \in \mathbb{N},\; (q_i, i_i, o_i) \in \mathcal{A} \text{ et } q_i \xrightarrow[o_i]{i_i} q_{i+1} \;\} \end{aligned}$$

3.2.1.3 Réactivité vis-à-vis de l'environnement

Soit Γ un ensemble de configurationsΓet soit $(q, i, o) \in \Gamma \cap \mathcal{A}$ une configuration de Γ vérifiant l'assertion sur l'environnement. Il peut exister un événement d'entrée i' compatible avec l'environnementΓtel que toute réaction o' de l'automate viole Γ :

$$\exists i' \in in^{\mathcal{A}}(q),\; \forall o' \in 2^O,\; (q, i', o') \notin \Gamma$$

L'état q est alors dit non réactif par rapport à l'assertion $\mathcal{A}$ sur l'environnement. IntuitivementΓ un ensemble de configurations Γ est dit réactif vis-à-vis de l'assertion $\mathcal{A}$ sur l'environnementΓs'il possède au moins une réaction pour toute entrée acceptée par l'assertion.

Définition 3.3 :

La fonction de réaction vis-à-vis de l'assertion $\mathcal{A}$ sur l'environnement est la fonction $react_Q^{\mathcal{A}}\Gamma$des ensembles de configurations dans les ensembles d'états définie par

$$react_Q^{\mathcal{A}}(\Gamma) \;=\; \{\; q \in Q \mid in^{\mathcal{A}}(q) \neq \emptyset \text{ et } \forall i \in in^{\mathcal{A}}(q),\; \exists o \in 2^O,\; (q, i, o) \in \mathcal{A} \cap \Gamma \;\}$$

Cette fonction est étendue pour renvoyer un ensemble de configurations :

$$react^{\mathcal{A}}(\Gamma) \;=\; (\Gamma \cap \mathcal{A}) \cap \left(\; react^{\mathcal{A}}_{Q}(\Gamma) \times 2^{I} \times 2^{O} \; \right)$$

□

Pour tout $q \in react^{\mathcal{A}}_{Q}(\Gamma)\Gamma$

1. il existe une action de l'environnement vérifiant l'assertion.
2. pour toute action de l'environnement permettant de vérifier l'assertionΓil existe un événement de sortie vérifiant l'assertion et l'inclusion dans Γ.

Propriété 3.2 :

Soit Γ un ensemble de configurations. Alors $react^{\mathcal{A}}(\Gamma)$ est l'ensemble maximal de configurations compatible avec l'assertion inclus dans Γ. □

Preuve :

Il suffit de montrer que $react^{\mathcal{A}}(react^{\mathcal{A}}(\Gamma)) = react^{\mathcal{A}}(\Gamma)$.

$$\begin{array}{rcl}
react^{\mathcal{A}}(react^{\mathcal{A}}(\Gamma)) & = & \{ \quad (q,i,o) \in \mathcal{A} \cap react^{\mathcal{A}}(\Gamma) \mid \; in^{\mathcal{A}}(q) \neq \emptyset \text{ et} \\
& & \quad \forall i' \in in^{\mathcal{A}}(q),\; \exists o' \in 2^{O},\; (q,i',o') \in \mathcal{A} \cap react^{\mathcal{A}}(\Gamma) \quad \} \\
& = & \{ \quad (q,i,o) \in \mathcal{A} \cap \Gamma \mid in^{\mathcal{A}}(q) \neq \emptyset, \\
& & \quad \forall i' \in in^{\mathcal{A}}(q),\; \exists o' \in 2^{O},\; (q,i',o') \in \mathcal{A} \cap \Gamma, \text{ et} \\
& & \quad \forall i' \in in^{\mathcal{A}}(q),\; \exists o' \in 2^{O},\; (q,i',o') \in \mathcal{A} \cap react^{\mathcal{A}}(\Gamma) \quad \} \\
& = & \{ \quad (q,i,o) \in \mathcal{A} \cap \Gamma \mid in^{\mathcal{A}}(q) \neq \emptyset, \\
& & \quad \forall i' \in in^{\mathcal{A}}(q),\; \exists o' \in 2^{O},\; (q,i',o') \in \mathcal{A} \cap \Gamma \quad \} \\
& = & react^{\mathcal{A}}(\Gamma)
\end{array}$$

□

La compatibilité est une propriété d'un étatΓsignifiant que pour toute action de l'environnement (un événement d'entrée)Γil existe une stratégie du processus (un événement de sortie) permettant de vérifier l'assertion. La réactivité est une propriété d'un ensemble de configurationsΓsignifiant que pour toute action de l'environnementΓil existe une stratégie du processus permettant de vérifier l'assertion et cet ensemble de configurations.

3.2.2 Stabilité et accessibilité

Les notions suivantes sont similaires aux modalités de CTLΓmais sont définies en tenant compte de la contrôlabilité.

Soient $\mathcal{Q}_1$ et $\mathcal{Q}_2$ deux ensembles d'étatsΓet soient Γ_1 et Γ_2 deux ensembles de configurations compatibles avec l'assertionΓi.e.Γ

$$\Gamma_1 \;=\; react^{\mathcal{A}}(\Gamma_1) \quad \text{et} \quad \Gamma_2 \;=\; react^{\mathcal{A}}(\Gamma_2)$$

Définition 3.4 :

Invariance : Γ_1 est invariant si toute configuration de Γ_1 mène dans Γ_1.

$$\Gamma_1 \subseteq pre(\Gamma_1)$$

IntuitivementΓl'invariance est équivalente à la préservation sous assertion.

Accessibilité forte : Γ_2 est fortement accessible (ou inévitable) dans Γ_1 si toute séquence infinie de configurations de Γ_1 conduit inévitablement dans Γ_2.

$$\begin{array}{c} \forall (q_0,i_0,o_0),(q_1,i_1,o_1),(q_2,i_2,o_2),\ldots \in \Gamma_1 \\ \text{tels que} \quad \forall i, \quad q_i \xrightarrow[o_i]{i_i} q_{i+1}, \\ \exists n \in \mathbb{N}, \quad \text{tel que} \quad (q_n,i_n,o_n) \in \Gamma_2 \end{array}$$

Stabilisabilité : $\mathcal{Q}_1$ est stabilisable si pour toute séquence d'événements d'entrée compatible avec l'environnementΓil existe une séquence d'événements de sortie compatible avec l'environnement qui reste dans $\mathcal{Q}_1$.

$$\forall q \in \mathcal{Q}_1,\ \forall i \in in^{\mathcal{A}}(q),\ \exists o \in 2^O, (q,i,o) \in \mathcal{A} \cap pre(\mathcal{Q}_1)$$

IntuitivementΓun ensemble d'états est stabilisable s'il existe une stratégie des événements de sortie le rendant invariant.

Accessibilité : $\mathcal{Q}_2$ est accessible depuis $\mathcal{Q}_1$ si de tout état de $\mathcal{Q}_1$ et pour toute séquence d'événements d'entrée compatible avec l'assertionΓil existe une séquence d'événements de sortie menant dans $\mathcal{Q}_2$.

$$\begin{array}{c} \forall q_0 \in \mathcal{Q}_1, \forall (i_0,i_1,\ldots) \in In^{\mathcal{A}}(q_0) \\ \exists n \in \mathbb{N}, \exists q_1,\ldots q_n \in Q, \exists o_0,\ldots o_{n-1} \in 2^O, \\ \forall i = 0\ldots n-1,\ (q_i,i_i,o_i) \in \mathcal{A}, \quad q_i \xrightarrow[o_i]{i_i} q_{i+1} \quad \text{et} \quad q_n \in \mathcal{Q}_2 \end{array}$$

IntuitivementΓun ensemble d'états est accessible s'il existe une stratégie des événements de sortie le rendant fortement accessible.

□

Ces propriétés vont nous permettre de caractériser les ensembles $\mathcal{I}_P\Gamma F_P^s$ et F_P^l.

- $\mathcal{I}_P$ est le plus grand ensemble de configurations invariant inclus dans $\mathcal{I}$.
- F_P^l est un ensemble de configurations inclus dans $\mathcal{I}_P$ à partir duquel F est fortement accessible.
- F_P^s est le plus grand ensemble de configurations invariant inclus dans $\mathcal{I}_P$ à partir duquel F est accessible.

Les sections suivantes proposent des algorithmes pour calculer ces ensembles de configurations.

3.3 Plus grand ensemble de configurations invariant

D'après la section précédente$\Gamma\Gamma$ est invariant si et seulement si

$$\Gamma \ = \ react^{\mathcal{A}}(\Gamma \cap pre(\Gamma))$$

Théorème 3.1 :

Il existe un plus grand ensemble de configurations invariant inclus dans Γ. Il est égal au plus grand point fixe de la fonction f_1 des ensembles de configurations dans les ensembles de configurations définie par

$$f_1 \ = \ \lambda X.react^{\mathcal{A}}(\Gamma \cap pre(X))$$

□

Preuve :

Cette fonction est croissante dans le treillis complet des ensembles de configurations. Elle admet donc un plus grand point fixe. □

3.4 Ensembles d'états accessible et fortement accessible

Soit Γ un ensemble de configurations et $\mathcal{Q}$ l'ensemble des états de Γ.

$$\mathcal{Q} \;=\; \{\; q \in Q \mid \exists i \in 2^I, \exists o \in 2^O,\; (q,i,o) \in \Gamma \;\}$$

Dans cette section nous nous proposons de calculer

1. L'ensemble maximal $\mathcal{Q}'$ d'états à partir duquel $\mathcal{Q}$ est accessible.
2. Un ensemble de configurations Γ' dont l'ensemble des états est $\mathcal{Q}'$ et à partir duquel Γ est fortement accessible.

De façon à satisfaire une propriété de vivacité nous cherchons un chemin pouvant atteindre les états finals mais pas forcément tous les chemins pouvant atteindre ces états finals. Ainsi nous ne demandons pas que Γ' soit maximal.

Propriété 3.3 :

Il n'existe pas de plus grand ensemble de configurations dans lequel Γ est fortement accessible. □

Preuve :

La figure 3.2 présente un contre-exemple. □

L'ensemble de configurations à partir duquel Γ est fortement accessible en *au plus* un pas est par définition

$$react^{\mathcal{A}}\,(\;pre(\Gamma)\;)$$

Donc l'ensemble des états à partir duquel $\mathcal{Q}$ est accessible en *au plus* un pas est

$$\{\; q \in Q \mid \exists i \in 2^I, \exists o \in 2^O,\; (q,i,o) \in react^{\mathcal{A}}\,(\;pre(\Gamma)\;)\;\}$$

De façon à calculer l'ensemble d'états à partir duquel $\mathcal{Q}$ est accessible en *exactement* un pas il est nécessaire de supprimer de l'ensemble précédent tous les états de $\mathcal{Q}$ (voir figure 3.3). On obtient ainsi l'ensemble de configurations Γ_1 et l'ensemble d'états $\mathcal{Q}_1$ définis par

$$\begin{aligned}
\Gamma_1 &= react^{\mathcal{A}}\,(\;pre(\Gamma) \cap (\;(Q \setminus \mathcal{Q}) \times 2^I \times 2^O\;)\;) \\
\mathcal{Q}_1 &= \{\; q \in Q \setminus \mathcal{Q} \mid \exists i \in 2^I, \exists o \in 2^O,\; (q,i,o) \in react^{\mathcal{A}}\,(\;pre(\Gamma)\;)\;\}
\end{aligned}$$

Nous définissons ainsi les deux fonctions

$$\begin{aligned}
\widehat{pre}_\Gamma &= \lambda\Gamma.react^{\mathcal{A}}\,(\;pre(\Gamma) \cap (\{\; q \in Q \mid \forall i \in 2^I, \forall o \in 2^O,\; (q,i,o) \notin \Gamma\;\} \times 2^I \times 2^O)\;) \\
\widehat{pre}_Q &= \lambda\mathcal{Q}.\{\; q \in Q \setminus \mathcal{Q} \mid \exists i \in 2^I, \exists o \in 2^O,\; (q,i,o) \in react^{\mathcal{A}}\,(\;pre(\mathcal{Q})\;)\;\}
\end{aligned}$$

Théorème 3.2 :

1. Il existe un plus grand ensemble d'états $\mathcal{Q}'$ à partir duquel $\mathcal{Q}$ est accessible. Il est égal à la limite de la suite croissante $(u_n^{\mathcal{Q}})_{n\geq 1}$ définie par

$$u_n^{\mathcal{Q}} \;=\; \bigcup_{0\leq i\leq n-1} \widehat{pre}_Q^{(i)}(\mathcal{Q})$$

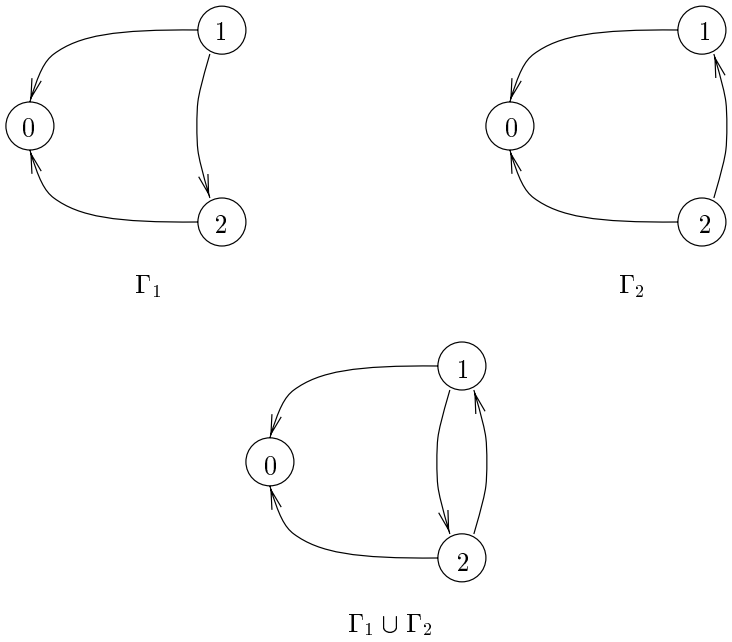


FIG. 3.2 – Contre-exemple prouvant la non unicité d'un ensemble de configurations à partir duquel Γ est fortement accessible. L'état 0 est accessible depuis l'ensemble d'états $\{1, 2\}$. Il est fortement accessible depuis Γ_1 et $\Gamma_2\Gamma$mais pas depuis $\Gamma_1 \cup \Gamma_2$ car il s'est formé une boucle.

2. Il existe un ensemble de configurations $\Gamma'\Gamma$dont les états sont ceux de $\mathcal{Q}'\Gamma$à partir duquel Γ est fortement accessible. On peut choisirΓpar exemple$\Gamma\Gamma'$ égal à la limite de la suite croissante $(u_n^\Gamma)_{n\geq 1}$ définie par

$$u_n^\Gamma \;=\; \bigcup_{0\leq i\leq n-1} \widehat{pre}_\Gamma^{(i)}(\Gamma)$$

□

Preuve :

1. $\widehat{pre}_Q^{(i)}(\mathcal{Q})$ est l'ensemble des états à partir duquel $\mathcal{Q}$ est accessible en exactement i pas. $(u_n^\mathcal{Q})_{n\geq 1}$ est croissante dans le treillis des ensembles d'étatsΓdonc elle converge.
2. $\widehat{pre}_\Gamma^{(i)}(\Gamma)$ est un ensemble de configurations à partir duquel Γ est fortement accessible en exactement i pas. $(u_n^\Gamma)_{n\geq 1}$ est croissante dans le treillis *fini* des ensembles de configurationsΓdonc elle converge en un temps *fini*.

□

3.5 Solution de la synthèse

3.5.1 Calcul d'invariance

L'invariant $\mathcal{I}_P$ de l'observateur synthétisé doit satisfaire la propriété de sûreté $\mathcal{I}$. Il doit donc être inclus dans l'ensemble de configurations invariant $\mathcal{I}'$ défini par

$$\mathcal{I}' \;=\; gfp(f_1(\mathcal{I}))$$

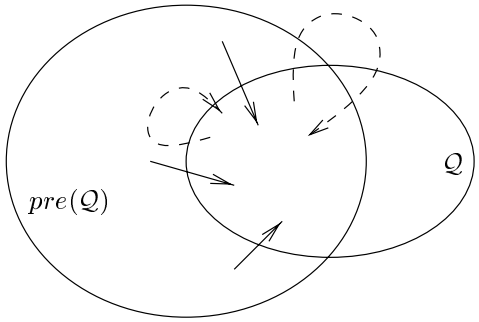


FIG. 3.3 – L'ensemble $\mathcal{Q}$ et l'ensemble de ses prédécesseurs. Les états sources des transitions représentées en trait plein atteignent $\mathcal{Q}$ en exactement un pas. Les états sources des transitions représentées en trait pointillé atteignent $\mathcal{Q}$ en 0 ou 1 pas.

Celui-ci est calculé itérativement :

$$\begin{aligned} \mathcal{I}'^{0} &= react^{\mathcal{A}}(\mathcal{I}) \\ \mathcal{I}'^{n+1} &= react^{\mathcal{A}}\left(\mathcal{I}'^{n} \cap pre(\mathcal{I}'^{n}) \right) \\ \mathcal{I}' &= \lim_{n\to\infty} \mathcal{I}'^{n} \end{aligned}$$

3.5.2 Calcul d'accessibilité

F^s_P est le plus grand ensemble de configurations invariant inclus dans $\mathcal{I}'$Γà partir des états duquel F est accessible. Il est calculé en deux temps :

1. On calcule tout d'abord le plus grand ensemble d'états $\mathcal{Q}^s$ à partir duquel F est accessible. Ce calcul est effectué comme précédemmentΓen tenant compte que l'invariant $\mathcal{I}'$ doit être préservé. La fonction $\widehat{pre}_Q$ est donc redéfinie de la manière suivante

$$\widehat{pre}_Q = \lambda\mathcal{Q}.\left\{ q \in Q \setminus \mathcal{Q} \mid \exists i \in 2^I, \exists o \in 2^O,\ (q,i,o) \in react^{\mathcal{A}}\left(pre(\mathcal{Q}) \cap \mathcal{I}' \right) \right\}$$

   $\mathcal{Q}^s$ est défini par

$$\begin{aligned} \mathcal{Q}^s &= \lim_{n\to\infty} u^{\mathcal{Q}}_n \\ \text{avec}\quad u^{\mathcal{Q}}_n &= \bigcup_{0\le i\le n-1} \widehat{pre}^{(i)}_Q(\mathcal{Q}) \\ \text{et}\quad \mathcal{Q} &= \left\{ q \mid \exists i \in 2^I, \exists o \in 2^O,\ (q,i,o) \in F \right\} \end{aligned}$$

   et est calculé itérativement.

2. F^s_P est le plus grand ensemble de configurations invariant inclus dans $\mathcal{I}'$ dont les états sont inclus dans $\mathcal{Q}^s$. Il est défini par

$$F^s_P = gfp\left(f_1\left(\mathcal{I}' \cap (\mathcal{Q}^s \times 2^I \times 2^O) \right) \right)$$

   et est calculé itérativement :

La solution du problème de synthèse doit satisfaire $\mathcal{I}'$ et F^s_P. Comme $F^s_P \subseteq \mathcal{I}'$Γon pose

$$\mathcal{I}_P = F^s_P$$

3.5.3 Calcul d'accessibilité forte

F_P^l est un ensemble de configurations inclus dans $\mathcal{I}'$ dont les états sont inclus dans $\mathcal{Q}^s$ et depuis lequel F est fortement accessible. Pour tenir compte de $\mathcal{I}'$Γon redéfinit comme précédemment $\widehat{pre}_\Gamma$ en

$$\widehat{pre}_\Gamma \quad = \quad \lambda\Gamma.react^{\mathcal{A}} \left(\ pre(\Gamma) \cap (\{q \in Q \mid \forall i \in 2^I, \forall o \in 2^O, \ (q,i,o) \notin \Gamma\} \times 2^I \times 2^O) \cap \mathcal{I}' \ \right)$$

F_P^l est défini par

$$\begin{aligned} F_P^l &= \lim_{n\to\infty} u_n^\Gamma \\ \text{avec} \quad u_n^\Gamma &= \bigcup_{0\le i\le n-1} \widehat{pre}_\Gamma^{(i)}(F) \end{aligned}$$

et est calculé itérativement.

3.5.4 Synthèse

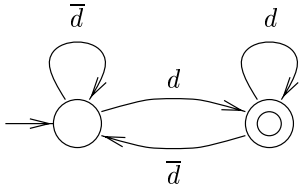
Afin de respecter la propriété de sûretéΓnotre processus solution doit infiniment souvent atteindre une configuration finale.

Ce problème est résolu en introduisant un nouveau signal d'entréeΓappelé signal "oracle"Γet noté d.

1. Tant que le signal oracle d n'est pas reçuΓle processus ne cherche pas à atteindre un état final. Il n'est contraint que par l'invariant $\mathcal{I}_P$.

2. Dès que le signal oracle est émisΓnotre processus choisit ses événements de sortie de façon à atteindre un état final. Cela est toujours possibleΓpuisque les états finals sont accessibles depuis les états de $\mathcal{I}_P$.

De façon à atteindre infiniment souvent un état finalΓle signal oracle d doit être infiniment souvent présent. d peut être ainsi généré par l'automate de Büchi suivant :



Le chapitre suivant va proposer plusieurs algorithmes calculantΓpour tout état et tout événement d'entréeΓl'ensemble des événements de sortie permettant de demeurer dans $\mathcal{I}_P$ (ou plus généralement dans n'importe quel ensemble de configurations invariant).

Chapitre 4

Synthèse d'automates

4.1 Problème

Soit $\mathcal{S} = (Q, q^0, I \cup O, \{\alpha_E^s, \alpha_\varphi^s, \alpha_\varphi^l\}, \delta^Q, \delta^O)$ une spécification. A l'aide des algorithmes vus dans le chapitre précédentΓnous avons déterminé deux ensembles de configurations $\mathcal{I}_P$ et F_P^lΓ correspondantΓrespectivementΓà l'invariant que doit vérifier la solution PΓet à un chemin conduisant inévitablement aux états finals. La spécification et ces deux ensembles permettent de calculer l'ensemble des solutions du problème de la synthèse.

Une fois résolu ce problèmeΓnous souhaitons obtenir un programme réel qui puisse être exécuté sur un ordinateur. Cela revient à déterminer explicitement la fonction de sortie δ_P^OΓi.e.Γk fonctionsΓ notées $(o_i^*)_{0 \leq i \leq k-1}$ et calculant dans chaque configuration la valeur de la sortie o_i (i.e.Γabsent ou présent). Ces fonctions vont être déterminisées à l'aide d'entrées supplémentairesΓappelées "oracles".

Les sections suivantes présentent 3 méthodes pour construire toutes les solutions déterministes du problème de la synthèse. Étant donné un ensemble de configurations Γ invariantΓces méthodes vont nous permettre de calculer toutes les valeurs possibles des sortiesΓde façon à ce que toute exécution de P reste dans Γ. Nous proposons trois méthodes :

1. Une méthode ensembliste qui peut être utilisée quel que soit le type d'implémentation.

2. Deux méthodes applicables aux circuits booléens et profitant de la structure de donnée utilisée pour coder les formules booléennes : les BDD.

4.2 Méthode ensembliste

4.2.1 Premier cas : il n'y a qu'une sortie

Soit Γ un ensemble de configurations invariant. Supposons dans un premier temps qu'il n'y ait qu'un seul signal de sortieΓi.e.ΓO est un singleton ($O = \{o_0\}$). Dans certaines configurationsΓ toute stratégie dans le choix de o_0 permet de rester dans Γ. Dans d'autres configurationsΓla seule stratégie possible est de choisir o_0 présent. FinalementΓdans les configurations restantesΓla seule stratégie possible est de choisir o_0 absent.

AinsiΓdéfinissons deux sous-ensembles de Γ : T (pour "True") et F (pour "False").

- T est l'ensemble des configurations dans lesquelles o_0 doit être présentΓi.e.Γtelles que si o_0 est absentΓl'exécution sort de Γ.

- De la même manièreΓF est l'ensemble des configurations dans lesquelles o_0 doit être absentΓ i.e.Γtelles que si o_0 est présentΓl'exécution sort de Γ.

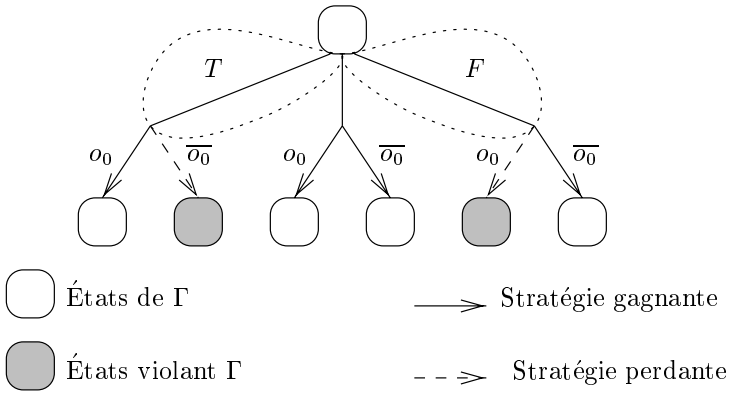


FIG. 4.1 – Ensembles de configurations T et F

T et F sont définis par

$$\begin{aligned} T &= \{\ (q,i) \in Q \times 2^I \mid (q, i \cup \{o_0\}) \in \Gamma \text{ et } (q,i) \notin \Gamma\ \} \\ F &= \{\ (q,i) \in Q \times 2^I \mid (q,i) \in \Gamma \text{ et } (q, i \cup \{o_0\}) \notin \Gamma\ \} \end{aligned}$$

(voir la figure 4.1). T et F sont disjoints. Si la configuration (q,i) est dans TΓle signal o_0 doit être présent. Si elle est dans FΓo_0 doit être absent. SinonΓo_0 est indéterminé. Pour obtenir tous les automates déterministes qui vérifient la spécificationΓintroduisons un nouveau signalΓappelé "signal oracle associé à o_0"Γet noté d_0Γqui choisit une valeur pour o_0 lorsqu'elle n'est pas contrainte. AinsiΓnous avons

$$\begin{aligned} \forall (q,i) \in Q \times 2^I,\ \ o_0^*(q,i) \ = \ & if \ \ (q,i) \in T \ \ then\ present \\ & else \ \ if \ \ (q,i) \in F \ \ then\ absent \\ & else \ \ d_0 \end{aligned}$$

4.2.2 Cas général

Supposons maintenant que la taille de O soit plus grande que 1 (i.e.Γ$O = \{o_0, o_1, \dots\}$). La stratégie gagnante utilisée pour le signal o_i peut dépendre de la stratégie gagnante utilisée pour les autres signaux $(o_j)_{j \neq i}$. Considérons le premier signal o_0 et appliquons le même raisonnement que précédemment.

Dans certaines configurationsΓtoute stratégie dans le choix de o_0 permet de rester dans ΓΓsi les stratégies des autres signaux de sortie sont bien choisies. Dans d'autres configurationsΓla seule stratégie possible est de choisir o_0 présent. Cela signifie que si o_0 est absentΓtoutes stratégies dans le choix des autres signaux de sortie conduit à sortir de Γ. FinalementΓdans les configurations restantesΓla seule stratégie possible est de choisir o_0 absent. AinsiΓon peut définirΓcomme à la section précédenteΓ2 ensembles de configurations: T_0 et F_0.

- T_0 est l'ensemble de configurations dans lesquelles o_0 doit être présentΓi.e.Γtelles que si o_0 est absentΓtoute exécution sort de ΓΓquels que soient les choix des autres signaux.
- De la même manièreΓF_0 est l'ensemble des configurations dans lesquelles o_0 doit être absentΓ i.e.Γtelles que si o_0 est présentΓtoute exécution sort de ΓΓquels que soient les choix des autres signaux.

T_0 et F_0 sont définis par

$$\begin{aligned} T_0 &= \{\ (q,i) \in Q \times 2^I \mid \exists o \in 2^O,\ (q,i,o) \in \Gamma \text{ et } \forall o \in 2^{O \setminus \{o_0\}},\ (q,i,o) \notin \Gamma\ \} \\ F_0 &= \{\ (q,i) \in Q \times 2^I \mid \exists o \in 2^O,\ (q,i,o) \in \Gamma \text{ et } \forall o \in 2^{O \setminus \{o_0\}},\ (q,i,\{o_0\} \cup o) \notin \Gamma\ \} \end{aligned}$$

et nous introduisons un signal oracle $d_0\Gamma$

$$o_0^*(q,i) \;=\; if \;\; (q,i) \in T_0 \;\; then\ present \\ else \;\; if \;\; (q,i) \in F_0 \;\; then\ absent \\ else \;\; d_0$$

Le calcul se continue de même pour les autres signaux de sortie. Le choix de o_0 est maintenant déterministe. AinsiΓle signal o_0 peut être considéré comme un signal d'entrée. Soit Γ^1 la nouvelle spécification définie par

$$\Gamma^1 \quad = \quad \Gamma[o_0/o_0^*]$$

Γ^1 ne dépend plus que des signaux de sortie $o_1\Gamma\ldots$ (et éventuellement de d_0). Nous pouvons appliquer l'algorithme précédent avec $O = \{o_1, \ldots\}$. AinsiΓla sortie o_i dépend des signaux oracles $d_0\Gamma\ldots\ \Gamma d_i$.

Le désavantage de cette technique est qu'elle nécessite le calcul de k substitutions (pour déterminer les Γ^i)Γqui est une opération particulièrement coûteuse. Dans la suiteΓnous nous intéressons à la synthèse de programmes pour les circuits booléens. Les deux méthodes de synthèse suivantes profitent de la structure des données utilisée : les BDD.

4.3 Méthode symbolique utilisant des tests

Soit Γ un ensemble de configurations invariantΓi.e.Γune formule booléenne des variables d'étatΓ d'entrée et de sortie. L'arbre de Shannon de Γ porte sur chacune de ses feuilles soit un 1Γsoit un 0. L'idée intuitive est de parcourir une et une seule fois cet arbre afin d'interdire toutes les configurations conduisant à une feuille portant un 0.

On numérote toutes les variables de l'automate de 0 à $n+m+k-1$ (on rappelle qu'on note respectivement $n\Gamma m$ et k le nombre de variables d'étatΓd'entrée et de sortie) et on les classe dans l'ordre suivant :

1. Les variables d'état et d'entrée $(x_i)\Gamma$pour $0 \leq i < n+m$.
2. Les variables de sortie $(x_i)\Gamma$pour $n+m \leq i < n+m+k$.

La racine de l'arbre de Shannon de Γ est donc une variable d'état ou une variable d'entréeΓalors que chacune de ses feuilles est reliée à une variable de sortie. Cet ordre est primordial pour l'efficacité de l'algorithme : il évite de parcourir plusieurs fois certaines portions de l'arbre.

On utilise les avantages des BDD par rapport aux arbres de Shannon pour ne pas parcourir plusieurs fois des branches similaires. Pour celaΓon va marquer chaque noeud du BDD lors du parcours et on va étiqueter ces noeuds par l'information correspondant à ce parcours.

4.3.1 Étiquetage du BDD de Γ

Soient $\mathcal{N}$ un noeud du BDD de Γ et i le numéro de la variable associée à ce noeud. A ce noeud est attachée une formule booléenne (notée $\mathcal{F}_\mathcal{N}$) des variables $(x_j)_{j \geq i}$. AinsiΓsi $i = n+m+k-1\Gamma$ (i.e.Γla variable considérée est classée dernière)Γon a $\mathcal{F}_\mathcal{N} = x_i$ ou $\mathcal{F}_\mathcal{N} = \overline{x_i}$.

L'idée est de considérer cette formule comme une "mini-spécification" ne portant que sur les variables $(x_j)_{j \geq i}$. Elle introduit donc des contraintes sur chacune de ces variables.

On attache donc à ce noeud un tableau de $n+m+k-i$ formules booléennes contenant chacune de ces contraintes. Plus précisémentΓla case numérotée j pour $0 \leq j < min(k, n+m+k-i)$ contient une formule booléenne définissant la contrainte sur la variable $x_{n+m+j}\Gamma$calculée à partir de la formule $\mathcal{F}_\mathcal{N}$. La première case contient donc la contrainte devant être imposée à x_i si x_i est une variable de sortie (voir figure 4.2).

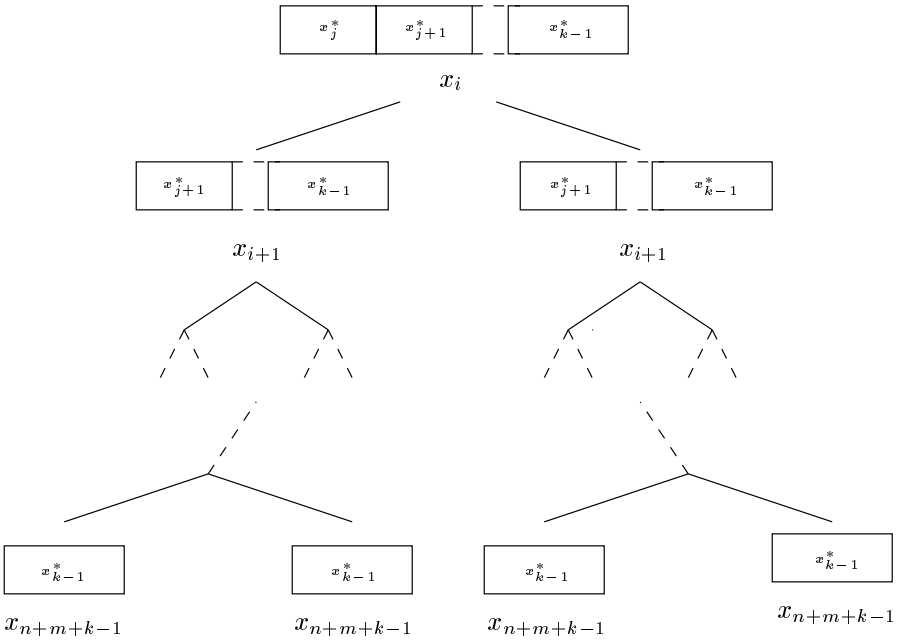


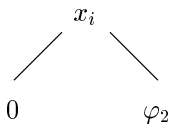
FIG. 4.2 – Étiquetage du BDD de Γ en vue de la synthèse de programmes

4.3.2 Parcours du BDD de Γ

Soient $\mathcal{N}$ un noeud du BDD de Γ et i le numéro de la variable associée à ce noeud. Plusieurs cas peuvent se présenter selon la valeur de $\mathcal{F}_\mathcal{N}$. Pour chacun de ces casΓil faut de plus distinguer le cas où la racine est une variable d'état ou une variable d'entréeΓet le cas où la racine est une variable de sortie.

1. $\mathcal{F}_\mathcal{N} = 1$

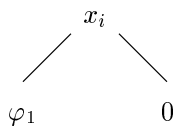
   La formule est déjà vérifiée. Toutes les variables de sortie restantes sont indéterminées (i.e.Γ on remplit le tableau associé au noeud $\mathcal{N}$ par $\forall j \geq 0,\ o_j^* = d_j$).

2. $\mathcal{F}_\mathcal{N}$ est de la forme 

   Si $x_i = 0\Gamma$la spécification est violée. Donc

   (a) Si x_i est une variable d'état ou une variable d'entréeΓil n'existe pas de stratégie des sorties permettant de rester à coup sûr dans l'ensemble de configurations $\Gamma\Gamma$car on ne peut pas empêcher x_i de prendre la valeur vraie. Si Γ est invariantΓx_i est forcée à vraie par l'assertion.

   (b) Si x_i est une variable de sortieΓil faut poser $o_{i-(n+m)}^* = 1$. Les contraintes sur les $(x_j)_{j>i-(m+n)}$ sont alors déterminées à l'aide de la formule φ_2.

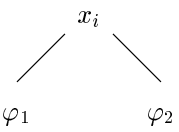
3. $\mathcal{F}_\mathcal{N}$ est de la forme 

Ce cas est symétrique au précédentΓi.e.Γ

(a) Si x_i est une variable d'état ou une variable d'entréeΓil n'existe pas de stratégie des sorties permettant de rester à coup sûr dans l'ensemble de configurations Γ. Comme précédemmentΓsi Γ est invariantΓx_i est forcée à fausse par l'assertion.

(b) Si x_i est une variable de sortieΓil faut imposer $o^*_{i-(n+m)} = 0$. Les contraintes sur les $(x_j)_{j>i-(n+m)}$ sont alors déterminées à l'aide de la formule φ_1.

4. $\mathcal{F}_\mathcal{N}$ est de la forme



Une exécution peut rester dans ΓΓque x_i soit vraie ou fausse. Si x_i est une variable de sortieΓ elle reste indéterminée (on pose $o^*_{i-(n+m)} = d_{i-(n+m)}$). Puis on itère le calcul dans le cas où x_i est vraie et dans celui où x_i est fausse :

(a) En partant de φ_1Γon calcule pour chaque j une formule $(o^*_j)_{\overline{x_i}}$ valable lorsque x_i est fausse.

(b) En partant de φ_2Γon calcule pour chaque j une formule $(o^*_j)_{x_i}$ valable lorsque x_i est vraie.

(c) EnfinΓon remplit le tableau associé au noeud $\mathcal{N}$ par les o^*_jΓcalculées pour chaque j par

$$o^*_j = \text{if } x_i \text{ then } (o^*_j)_{x_i} \text{ else } (o^*_j)_{\overline{x_i}}$$

si x_i est une variable d'état ou d'entréeΓet

$$o^*_j = \text{if } d_{i-(n+m)} \text{ then } (o^*_j)_{x_i} \text{ else } (o^*_j)_{\overline{x_i}}$$

si x_i est une variable de sortie.

Le défaut de cette technique est qu'elle interdit le réordonnancement automatique des variables (voir section 12.1.2.2Γpage 140) pour ne pas perdre les informations associées aux noeuds du BDD de Γ. Si elle est assez efficace sur les petits exemplesΓelle est inutilisable sur les exemples de taille importanteΓqui nécessitent le réordonnancement automatique pour pouvoir être chargés en mémoire (voir chapitre 5Γpage 51).

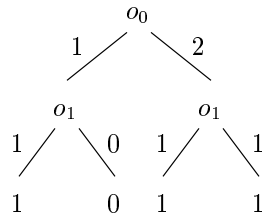
Par contreΓelle peut être aisément étendue de façon à calculer des probabilités sur les oracles.

4.3.3 Synthèse probabiliste

Les signaux oracles servent à modéliser le comportement non déterministe de nos processus synthétisés. Lors de l'exécution de ces programmesΓl'ordinateur doit choisir une valeur aléatoire pour chaque variable oracle. La technique précédente permet de calculer les probabilités d'occurrence qu'il convient de donner à chacune de ces variablesΓde façon à ce que l'ensemble des comportements possibles du processus synthétisé se produisent de façon équiprobable.

Comme l'ensemble de configurations Γ est invariantΓon peut décider à tout moment si un couple (q, i) est acceptable ou non.

Soit une configuration de l'automateΓi.e.Γun couple (q, i) vérifiant l'assertion. En descendant le BDD de Γ selon la valeur de (q, i)Γon obtient un BDD portant sur les seules variables de O. Considérons l'arbre de Shannon de Γ au lieu de son BDD. Il y a autant de choix possibles pour o qu'il y a de feuilles de cet arbre égales à 1 (il y en a au moins une car Γ est invariant). Pour obtenir tous les vecteurs de sortie de façon équiprobableΓil suffit donc d'étiqueter chaque branche par le nombre de 1 auxquels elle permet d'accéder. AinsiΓl'arbre de Shannon suivant



autorise les trois vecteurs $\overline{o_0} \wedge \overline{o_1}$Γ$o_0 \wedge \overline{o_1}$ et $o_0 \wedge o_1$. Il faut donc choisir o_0 avec une probabilité de $\frac{2}{1+2}$ et $\overline{o_0}$ avec une probabilité de $\frac{1}{1+2}$.

4.4 Méthode symbolique utilisant des conjonctions

Soit Γ un ensemble de configurations invariantΓi.e.Γune formule booléenne des variables d'étatΓ d'entrée et de sortie. L'idée de cette méthode va être de partir de $o_i^* = 0$ pour tout i et d'ajouter au fur et à mesure les configurations dans lesquelles o_i doit être vraie. Nous allons donc parcourir le BDD implémentant Γ en retenant le chemin qui y accèdeΓet mettre à jour les o_i^* à chaque noeud du BDD concernant une variable de sortie. Cette technique nécessite comme précédemment que les variables d'état et les variables d'entrée soient rangées avant les variables de sortie (dans l'ordre des variables des BDD). Cette technique peut être étendue à un ordre quelconque (voir plus loin).

Soient $\mathcal{N}$ un noeud de ΓΓi le numéro de la variable associée à ce noeudΓet soit *path* un BDD symbolisant le chemin parcouru pour parvenir à ce noeud. On distingue les valeurs suivantes de $\mathcal{F}_\mathcal{N}$:

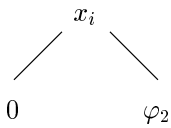
1. $\mathcal{F}_\mathcal{N} = 1$

   La formule est déjà vérifiée. Toutes les variables de sortie restantes sont indéterminéesΓi.e.Γ on poseΓpour tout $j \geq 0$

   $$o_j^* = o_j^* \vee (path \wedge d_j)$$

2. $\mathcal{F}_\mathcal{N} = 0$

   Cette formule ne peut pas être rendue vraieΓil n'existe donc pas de stratégie des sorties permettant de rester à coup sûr dans l'ensemble de configurations Γ. Ce cas est impossible si Γ est invariant.

3. $\mathcal{F}_\mathcal{N}$ est de la forme 

   Si $x_i = 0$Γla spécification est violée. Donc

   (a) Si x_i est une variable d'état ou une variable d'entréeΓon met à jour le chemin parcouru

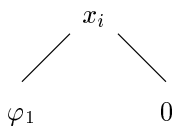
   $$path \ = \ path \wedge x_i$$

   et on continue récursivement le calcul pour φ_2.

   (b) Si x_i est une variable de sortieΓon met à jour $o^*_{i-(n+m)}$ en posant

   $$o^*_{i-(n+m)} = o^*_{i-(n+m)} \vee path$$

   et on continue récursivement le calcul pour φ_2 (sans modifier la variable *path*).

4. $\mathcal{F}_\mathcal{N}$ est de la forme 

Ce cas est symétrique au précédent i.e.

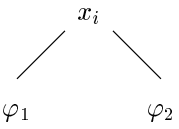
(a) Si x_i est une variable d'état ou une variable d'entrée on met à jour le chemin parcouru

$$path \;=\; path \wedge \overline{x_i}$$

et on continue récursivement le calcul pour φ_1.

(b) Si x_i est une variable de sortie on continue récursivement le calcul pour φ_1 (sans modifier la variable $path$). $o^*_{i-(n+m)}$ n'est pas modifiée puisque x_i ne peut pas être vraie dans la configuration $path$.

5. $\mathcal{F}_\mathcal{N}$ est de la forme



Une exécution peut rester dans Γ que x_i soit vraie ou fausse. Si x_i est une sortie elle reste indéterminée (on pose $o^*_{i-(n+m)} = o^*_{i-(n+m)} \vee (path \wedge d_{i-(n+m)})$). Puis on itère le calcul dans le cas où x_i est vraie et dans celui où x_i est fausse :

(a) En partant de φ_1 on pose

$$path \;=\; path \wedge \begin{cases} x_i & \text{si } x_i \text{ est une variable d'état ou d'entrée.} \\ d_{i-(n+m)} & \text{si } x_i \text{ est une variable de sortie.} \end{cases}$$

et on itère le calcul.

(b) En partant de φ_2 on pose

$$path \;=\; path \wedge \begin{cases} \overline{x_i} & \text{si } x_i \text{ est une variable d'état ou d'entrée.} \\ \overline{d_{i-(n+m)}} & \text{si } x_i \text{ est une variable de sortie.} \end{cases}$$

et on itère le calcul.

Extension à un ordre quelconque : Cet algorithme n'est vraiment efficace que si les variables de sortie sont classées dernières (dans l'ordre des variables des BDD). Néanmoins si le réordonnancement automatique (voir section 12.1.2.2 page 140) est utilisé (ce qui est obligatoire pour les programmes de taille importante) il est facile d'étendre cet algorithme à un "ordre proche". Tant que le BDD contient des variables d'état ou d'entrée on considère la décomposition de Shannon en fonction de la première variable d'état ou d'entrée apparaissant au lieu de considérer systématiquement la variable racine du BDD. Si l'ordre initial n'a pas été trop modifié on peut s'attendre à ce que cette variable soit souvent la variable racine. La perte d'efficacité de l'algorithme est alors négligeable.

4.5 Synthèse des événements de sortie

Les algorithmes précédents permettent de générer des événements de sortie forçant toute exécution à demeurer dans un ensemble de configurations donné. Ils sont appliqués pour les ensembles $\mathcal{I}_P$ (l'invariant à respecter) et F^l_P (l'ensemble de configurations conduisant vers un état final). On obtient ainsi les deux fonctions de sortie $\delta^O_\mathcal{I}$ et δ^O_F. Infiniment souvent les sorties du processus synthétisé doivent être générées par δ^O_F de façon à atteindre un état final. Le reste du temps elles sont générées par $\delta^O_\mathcal{I}$. Pour modéliser ce comportement introduisons un signal oracle supplémentaire noté d qui décide d'atteindre un état final. La fonction de sortie du processus P est alors définie de la manière suivante :

$$\forall (q,i) \in Q \times (2^{I \cup \{d\}}),\; \delta^O_P(q,i) \;=\; if \;\; d \in i \;\; then\; \delta^O_F(q,i) \;\; else \;\; \delta^O_\mathcal{I}(q,i)$$

Pour que le processus synthétisé ait un comportement correct il faut que le signal oracle soit présent infiniment souvent et qu'il le reste jusqu'à ce que le processus ait atteint un état marqué.

Chapitre 5

Exemples de synthèse de processus

Les techniques de synthèse de processus ont été implémentées (voir partie III). Dans ce chapitre, nous présentons deux exemples qui nous ont permis de tester l'efficacité des différentes méthodes. Un petit simulateur permet d'exécuter les processus synthétisés. Les calculs ont été effectués sur un **Ultra-Sparc 1** avec 256 Mo de mémoire.

5.1 Jeu de Nim

Le jeu de Nim est un jeu d'allumettes qui se joue à deux joueurs. On considère n rangées d'allumettes : la première contient 1 allumette, la seconde 3, la troisième 5, la $n^{ième}$ $2n-1$. Les joueurs enlèvent à tour de rôle un nombre quelconque d'allumettes (au moins égal à 1), dans une seule rangée. Celui qui prend la dernière allumette a gagné.

Nous avons écrit un programme C qui prend le paramètre n en entrée et qui génère le circuit booléen correspondant. Nous avons essayé de générer une stratégie gagnante pour différentes valeurs de n, que l'ordinateur commence ou non (dans ce jeu, l'utilisateur joue le rôle de l'environnement du processus). La propriété à démontrer peut être interprétée soit comme une propriété de sûreté, soit comme une propriété de vivacité :

Sûreté : L'environnement ne prend pas la dernière allumette.

Vivacité : Le processus prend la dernière allumette.

Les tableaux suivants indiquent s'il existe une stratégie gagnante pour ce problème, et dans l'affirmative, le temps de calcul nécessaire pour synthétiser un processus qui gagne à tous les coups. Les résultats pour les trois méthodes vues au chapitre précédent sont présentés :

Méthode ensembliste (sûreté) :

Nombre de ...			Premier coup		Synthèse
Lignes	Allumettes	Etats	Ordinateur	Utilisateur	Partielle
2	4	16	<1" 7 Ko	Echec	<1" 3 Ko
3	9	512	12" 83 Ko	Echec	5" 32 Ko
4	16	65 536	Echec	1h29'18" 1 232 Ko	17'47" 377 Ko
5	25	33 554 432	saturation		

La synthèse partielle consiste à ne déterminer qu'une solutionΓen choisissant une valeur arbitraire pour les oracles. Le programme ainsi obtenu est moins généralΓmais est déterministe et a une taille plus raisonnable.

Pour le cas à 5 lignesΓl'algorithme sature la mémoire en plusieurs heures.

Méthode par tests (sûreté) :

Nombre de ...			Premier coup		Synthèse
Lignes	Allumettes	Etats	Ordinateur	Utilisateur	Partielle
2	4	16	<1" 3.5 Ko	Echec	<1" 3 Ko
3	9	512	5" 74 Ko	Echec	5" 47 Ko
4	16	65 536	saturation		

Les résultats de cette méthode sont très décevantsΓpour la raison suivante : le fait de décorer les noeuds du BDD avec des informations interdit l'usage du réordonnancement automatique (voir section 12.1.2.2Γpage 140). Pour des petits automatesΓcette technique est très efficaceΓmais elle est inutilisable pour des gros. Le chargement (sans réordonnancement automatique) du circuit booléen correspondant au cas à 4 lignesΓfait exploser la mémoireΓavant même le calcul des états accessibles.

Méthode par conjonction (sûreté) :

Nombre de ...			Premier coup		Synthèse
Lignes	Allumettes	Etats	Ordinateur	Utilisateur	Partielle
2	4	16	<1" 5 Ko	Echec	<1" 4 Ko
3	9	512	6" 44 Ko	Echec	5" 30 Ko
4	16	65 536	Echec	22'38" 822 Ko	15'23" 374 Ko
5	25	33 554 432	saturation		

Cette technique estΓet de loinΓla plus efficace. On s'aperçoit néanmoins qu'il est très difficile de générer l'ensemble des solutions du problème de la synthèseΓle résultat occupant en général un très gros espace.

Méthode par conjonction (vivacité) : Nous essayons de synthétiser un processus qui prend la dernière allumette. Vu les résultats précédentsΓnous utilisons la technique par conjonction. Les résultats sont les suivants :

Nombre de ...			Premier coup		Synthèse
Lignes	Allumettes	Etats	Ordinateur	Utilisateur	Partielle
2	4	16	<1" 4 Ko	Echec	<1" 3 Ko
3	9	512	7" 36 Ko	Echec	6" 17 Ko
4	16	65 536	Echec	36'3" 753 Ko	23'22" 189 Ko
5	25	33 554 432	saturation		

La figure 5.1 présente le déroulement d'une partieΓoù il y a 3 rangées d'allumettesΓet l'ordinateur joue en premier. Le premier schéma correspond à l'état initial. Les carrés grisés correspondent

aux allumettes retirées au coup précédent. Dès le deuxième schémaΓil est facile de voir que l'ordinateur a trouvé une stratégie gagnante.

5.2 Chariot automatique

On considère une usine dans laquelle circule un chariot automatique. L'objectif de ce chariot est de transporter des pièces d'un atelier à un autreΓdans un ordre prédéfini. Dans cette usineΓ un piéton peut également se déplacer de façon aléatoire. Notre objectif est de synthétiser les commandes de ce chariot de façon à ce qu'il puisse transporter les pièces d'un atelier à un autreΓ sans heurter le piéton qui se déplace.

Soit n un entier. Nous modélisons l'usine par un carré de 2^n sur 2^n cases. Le chariot ainsi que l'homme sont modélisés par leurs coordonnées dans ce carré. Le processus qui contrôle le chariot reçoit en entrée les déplacements de l'hommeΓet émet les commandes du chariot. Ces commandes sont :

left	Déplacement vers la gauche
right	Déplacement vers la droite
up	Déplacement vers le haut
down	Déplacement vers le bas

Notre spécification admet donc les signaux d'entrée $\text{left}^{\text{piéton}}$Γ$\text{right}^{\text{piéton}}$Γ$\text{up}^{\text{piéton}}$ et $\text{down}^{\text{piéton}}$ provenant du piéton (considéré comme l'environnement) et émet les commandes $\text{left}^{\text{chariot}}$Γ$\text{right}^{\text{chariot}}$Γ $\text{up}^{\text{chariot}}$ et $\text{down}^{\text{chariot}}$ pour le chariot.

Le processus doit servir 4 ateliersΓsitués respectivement aux quatre sommets du carré représentant l'usine (voir figure 5.2).

Nous nous sommes limité à la technique par conjonctionΓavec réordonnancement automatique des variables. Le tableau suivant donne le résultat de la synthèse :

Taille		Complet		Partiel	
n	Nb états	Temps de calcul	Taille du résultat	Temps de calcul	Taille du résultat
3	3780	4'37"	227 Ko	1'45"	79 Ko
4	64260	41'37"	788 Ko	17'58"	358 Ko
5	1 043 460	8h1'57"	1.38 Mo	4h56'41"	841 Ko
6	16 756 740	112h	2 176 Ko	92h	1 223 Ko

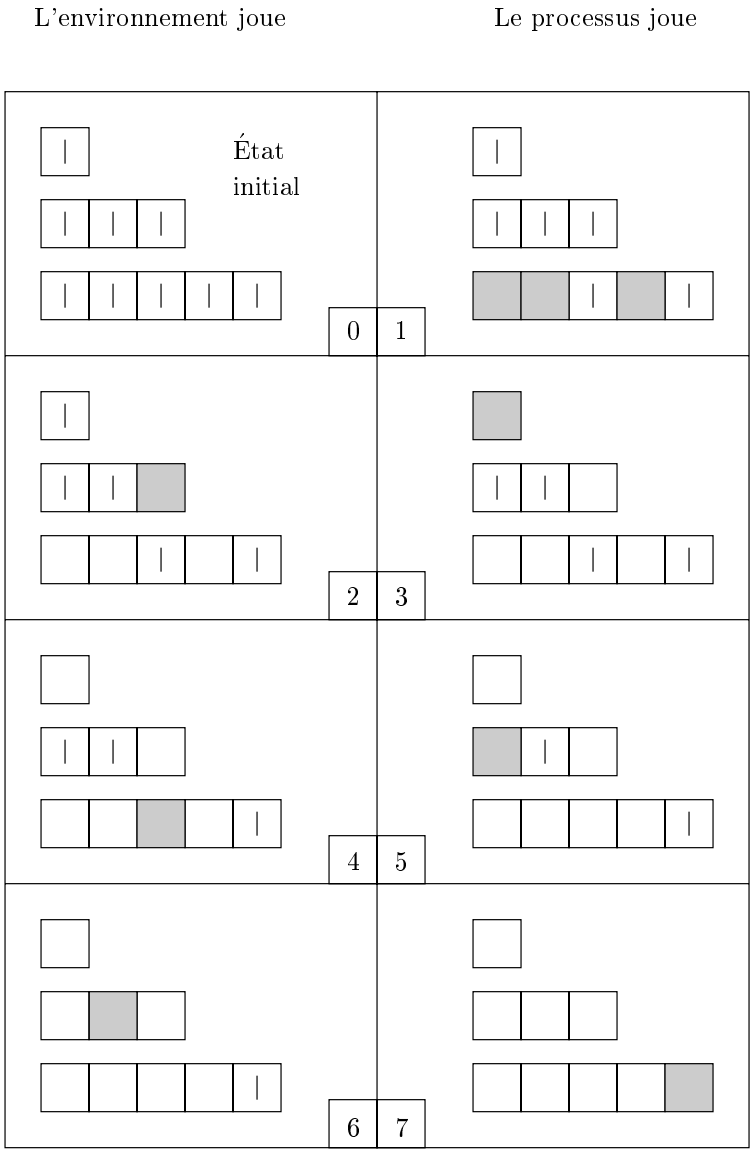


FIG. 5.1 – Une partie de jeu de nim

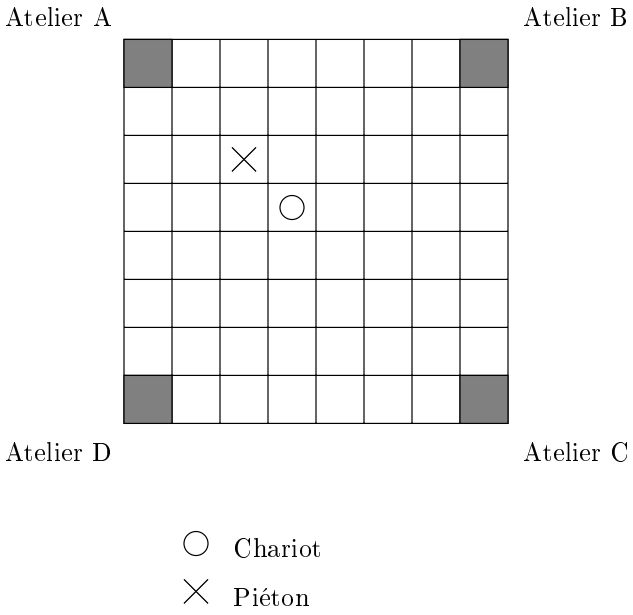


FIG. 5.2 – Une usine avec un chariot automatique

Deuxième partie

Synthèse d'invariants

Chapitre 6

Spécification et vérification des réseaux réguliers

Dans cette seconde partie nous allons étendre la preuve modulaire qui fait intervenir deux processus parallèles communicants au cas de n processus parallèles communicants. Dans ce chapitre nous allons présenter le modèle des grammaires de réseau. Nous présentons l'état de l'art de la spécification et de la vérification des réseaux de processus.

6.1 Description des réseaux de processus

6.1.1 Systèmes infinis

La vérification automatique de processus synchrones est souvent basée sur l'exploration de l'ensemble des états accessibles du système. La difficulté principale dans l'application de cette méthode est le nombre important et parfois même infini d'états à considérer pour couvrir l'ensemble des exécutions possibles d'un processus (voir section 10.1 page 109).

Principalement deux méthodes ont été envisagées pour pallier à ce problème d'explosion du nombre des états :

1. La définition de structures plus évoluées que les automates permettant de représenter des comportements infinis : on trouve dans cette catégorie les réseaux de Petri les automates à piles les systèmes hybrides etc ... Ces différents formalismes permettent de modéliser un grand nombre de systèmes mais leur complexité limite leur utilité pratique : certains problèmes élémentaires sont au moins PSPACE-complets ou même indécidables. Néanmoins cette direction de recherche est toujours en pleine expansion. En particulier des techniques d'extrapolation ([CC77 CC92]) sont étudiées pour calculer une approximation supérieure de l'ensemble des états accessibles. Dans la section 8.3 page 90 nous utiliserons des techniques similaires mais dans un cadre différent pour calculer des approximations inférieures de plus grands points fixes.

2. La construction récursive de systèmes infinis à partir de systèmes finis. Les algèbres de processus ([Mil80 NS94]) sont l'exemple le plus classique de cette approche; pourtant leur utilisation reste surtout adaptée au monde asynchrone et est ainsi la base de nombreux travaux sur les protocoles de communication.

   Une autre approche très technique a été développée récemment par Quemener [QJ95] : il décrit un graphe infini à l'aide d'une grammaire de graphe. Cette technique permet de représenter une classe importante de graphes infinis (correspondant en fait à la classe des langages hors contexte) mais peut paraître peu intuitive par rapport à une algèbre de processus. La complexité des algorithmes utilisés limite également l'intérêt de la méthode.

Des travaux récents s'intéressent à la vérification de réseaux particuliersΓréseaux en anneau [EN95] ou réseaux de processus asynchrones parallèles [EN96ΓSV94ΓID96]. Ces approches sont présentées dans la section 6.3Γpage 63.

Nous nous intéresserons à une autre approcheΓinspirée des précédentesΓqui peut s'adapter aussi bien au modèle synchrone qu'au modèle asynchrone : les grammaires de réseau.

6.1.2 Grammaires de réseau

6.1.2.1 Définition

Les grammaires de réseau ont été introduites par GrumbergΓShtadler et Marelly [SG89ΓMG91] : elles offrent un formalisme pour décrire des ensembles de réseaux de processus.

Une grammaire de réseau est un tuple $\mathcal{G} =< T, N, \mathcal{P}, \mathcal{S} >$ où

- T est un ensemble de processus terminaux.
- N est un ensemble de processus non terminaux. Chaque processus non terminal définit un sous-réseau.
- $\mathcal{P}$ est un ensemble de règles de production de la forme

  $$A \rightarrow B_1 \|_i B_2 \|_i \ldots \|_i B_n$$

  où $A \in N\Gamma B_1, \ldots, B_n \in T \cup N$ et $\|_i$ est une fonction de composition d'arité n.
- $\mathcal{S} \in N$ est le symbole de départ qui représente le réseau généré par la grammaire.

Exemple 6.1 :

Considérons une grammaire de réseau définissant un système assurant l'exclusion mutuelle à l'aide d'un algorithme à jeton circulant ("token ring"). Cette grammaire possède deux processus terminaux E et $P\Gamma$et un non terminal R.

- P est un processus élémentaire qui peut demander une ressource et l'utiliser.
- E est l'environnement dans lequel fonctionne le réseau. L'utilité de cet environnement sera discutée par la suite.
- R est un sous-réseau constitué de n processus P hors environnement.

La grammaire est définie de la manière suivante.

$$\begin{aligned} \mathcal{S} &\rightarrow E\|_1 R \\ R &\rightarrow P\|_2 R \\ R &\rightarrow P\|_2 P \end{aligned}$$

Cette grammaire permet ainsi de générer les réseaux suivants :

$$\begin{aligned} &E\|_1(P\|_2 P) \\ &E\|_1(P\|_2 P\|_2 P) \\ &E\|_1(P\|_2 P\|_2 P\|P_2) \\ &\text{etc } \ldots \end{aligned}$$

i.e.Γl'ensemble des réseaux à jeton circulant de taille $n\Gamma$pour tout n. □

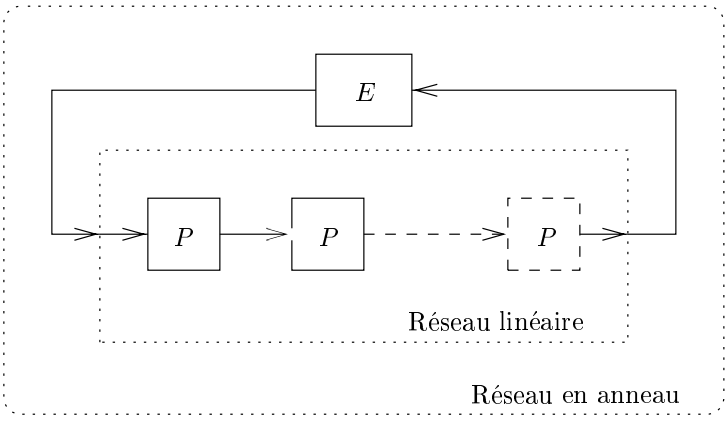


FIG. 6.1 – Construction par récurrence d'un réseau en anneau

Remarque sur l'utilité de l'environnement : Dans cet exemple, l'environnement peut servir à créer des réseaux en anneau ("ring"). En effet, un tel réseau ne peut pas être directement construit par récurrence. Pour construire le réseau de taille $n+1$ à partir de celui de taille n, il faudrait "couper" le réseau pour l'agrandir. Dans cet exemple, on crée un réseau linéaire de taille n, qui est ensuite rebouclé par un environnement (voir figure 6.1).

Cet exemple sera repris et développé dans le chapitre 10.

6.1.2.2 Des grammaires de réseau particulières

On s'intéressera dans les chapitres 8 et 9 à deux types particuliers de réseau.

Réseau linéaire : Un réseau est dit linéaire si toute règle de production est de la forme

$$R \;=\; (A \to B \|_i C)$$

où B et C ne sont pas tous les deux non terminaux. Ces réseaux permettent par exemple de construire des anneaux de processus (voir l'exemple précédent).

Réseau arborescent : Un réseau est dit arborescent binaire si toute règle de production est de la forme

$$R \;=\; (A \to B \|_i C)$$

où soit B et C sont tous les deux terminaux, soit $B = C$.

Exemple 6.2 :

La grammaire suivante décrit une structure arborescente binaire. Soient P un processus terminal, et N (pour "node") un processus non terminal.

$$\begin{aligned} \mathcal{S} &\to N \|_1 N \\ N &\to N \|_2 N \\ N &\to P \|_2 P \end{aligned}$$

□

(voir figure 6.2).

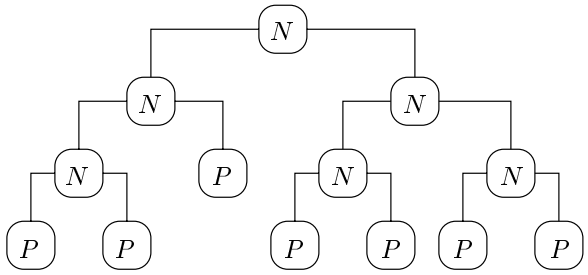


FIG. 6.2 – Réseau arborescent

6.2 Spécification de propriétés

On a vu dans le chapitre 1Γpage 13 deux façons différentes (mais équivalentes) d'exprimer des propriétés sur des processus. Exprimer des propriétés sur des grammaires de réseau est une tâche plus difficile: la spécification d'une propriété doit être indépendante de la taille du réseau.

Une première technique de spécification consiste à utiliser une formule logique contenant des quantificateurs existentiels ou universels sur les processus; l'inconvénient de cette technique est qu'elle ne peut pas prendre en compte la structure du réseau. Dans cette sectionΓnous présentons les langages de spécification introduits par ClarkeΓGrumberg et Jha dans [CGJ95]Γainsi que l'extension qui en a été faite par KestenΓMalerΓMarcusΓPnueli et Shahar dans [KMM+97].

6.2.1 Langage de spécification

Les langages de spécification ont été introduits par ClarkeΓGrumberg et Jha dans [CGJ95]. Soit $\mathcal{G} =< T, N, \mathcal{P}, \mathcal{S} >$ une grammaire de réseau et soit Q l'ensemble de tous les états de tous les processus terminaux de la grammaire $\mathcal{G}$ (comme l'ensemble des processus terminaux est finiΓet que les processus terminaux sont à états finisΓQ est fini). Soit un réseau généré par la grammaire $\mathcal{G}$. Si ce réseau est de taille nΓun état de ce système est un tuple $(q_1, \ldots, q_n) \in Q^n$. Un tel état peut être vu comme un mot de Q^*. Une propriété atomique (i.e.Γun ensemble d'états) peut alors être spécifié par un langage régulier sur Q.

Définition 6.1 :

> Un langage de spécification sur la grammaire de réseau $\mathcal{G}$ est un langage régulier sur l'alphabet Q. □

Soit D un automate déterministe reconnaissant un langage de spécification $T_D \subseteq Q^\infty$. Nous dirons donc qu'un état $\tau \in Q^*$ du système satisfait cette propriété si et seulement si $\tau \in T_D$. ClarkeΓ Grumberg et Jha expriment alors une propriété à l'aide des logiques $\forall CTL$ ou $\forall CTL^*$Γoù les propositions atomiques sont de tels langages réguliers.

Exemple 6.3 :

> Supposons un réseau engendré par un seul processus terminal P à deux états: $Q = \{cs, nc\}$. L'état cs correspond à un processus dans une section critique (par exemple l'utilisation d'une ressource)Γalors que nc correspond à un processus hors de la section critique.
>
> La propriété d'exclusion mutuelleΓqui spécifie qu'un et un seul processus est dans la section critique se traduit par
>
> $\forall\Box(nc^*.cs.nc^*)$

(où on utilise une expression régulière pour décrire le langage T_D). La propriété qui spécifie qu'au plus un processus est dans la section critique se traduit par

$$\forall\Box(nc^* \ + \ nc^*.cs.nc^*)$$

De mêmeΓla propriété qui spécifie la non famine du premier processus est exprimée par

$$\forall\Diamond(cs.nc^*)$$

□

6.2.2 Une extension des langages de spécification

Dans [KMM+97]ΓKestenΓMalerΓMarcusΓPnueli et Shahar proposent d'étendre le modèle des langages de spécification (qui spécifient des propriétés) à la spécification du système lui-même.

Soit Q l'ensemble des états de tous les processus terminaux du réseau. Comme précédemmentΓun état du système peut être vu comme un mot $\tau \in Q^*$. KestenΓMalerΓMarcusΓPnueli et Shahar proposent de décrire le comportement du système directement sur les mots de Q^*. AinsiΓla fonction de transition du système est une fonction de réécriture de mots sur Q^*.

Exemple 6.4 :

Reprenons l'exemple précédent de l'exclusion mutuelle. Supposons que cette exclusion mutuelle soit assurée à l'aide d'un système à jeton circulant. Les processus sont disposés en anneau. Un jeton peut circuler dans le sens des aiguilles d'une montre. Un processus qui possède le jeton est dans l'état t (pour "token")Γun processus qui ne le possède pas est dans l'état nt (pour "no token"). Un processus ne peut passer dans la section critique que s'il a le jeton. Le système est alors décrit par les règles de réécriture suivantes :

$t.nt$	$\rightarrow$	$nt.t$	Passage du jeton
t	$\rightarrow$	cs	Entrée en section critique
cs	$\rightarrow$	t	Sortie de la section critique

Il est alors possible de prouver par exploration du graphe des états que la propriété d'exclusion mutuelle est vérifiée. □

Ce modèle est étendu au cas arborescentΓgrâce à l'utilisation d'automates arborescents ("tree automata"). ActuellementΓseul le cas linéaire a été implémenté.

Commentaires : L'intérêt principal de ce modèle est sa simplicité. En contrepartieΓil semble extrêmement difficile de décrire des systèmes plus complexesΓsurtout dans le cas des réseaux arborescents. De plusΓles recherches actuelles n'ont pas encore abordé le problème de la convergence lors de la recherche des états accessibles. Il semble probable que dans la plupart des casΓil faille inventer des heuristiques pour extrapoler cet ensemble. ActuellementΓce modèleΓbien que prometteurΓn'en est encore qu'au stade des préliminaires.

Dans le chapitre suivantΓnous proposerons une nouvelle technique de spécification basée sur les réseaux d'observateursΓqui sera comparée à la méthode de ClarkeΓGrumberg et Jha.

6.3 Vérification de propriétés

Soit $\mathcal{G} =< T, N, \mathcal{P}, \mathcal{S} >$ une grammaire de réseauΓet soit φ une propriété. On dit que la grammaire $\mathcal{G}$ satisfait la propriété φ si et seulement si tout processus généré par la grammaire vérifie φ.

La vérification de propriétés sur des grammaires de réseau est une tâche difficile. Apt et Kozen ont montré dans [AK86] que ce problème est indécidable même s'il est décidable pour chaque processus généré par la grammaire. Pour traiter le problème de la vérification des grammaires de réseau on peut donc distinguer deux approches :

1. Définir une sous-classe décidable de problèmes. L'inconvénient est que ces sous-classes sont souvent très restrictives : soit les réseaux considérés ont une structure très particulière soit la propriété qu'il est possible de vérifier est très faible.

2. Utiliser une technique qui peut échouer ou qui ne finit éventuellement pas : n'importe quel système et n'importe quelle propriété peuvent alors être pris en compte mais l'algorithme peut avoir les comportements suivants :

   - Répondre oui en un temps fini.
   - Répondre non en un temps fini.
   - Répondre "je ne sais pas" en un temps fini.
   - Ne pas donner de réponse en un temps fini i.e. l'algorithme ne termine pas.
     En pratique même les procédures de décision sont souvent d'une telle complexité qu'elles saturent la mémoire.

La section suivante expose la technique développée par Clarke Grumberg et Jha pour extrapoler des invariants d'une grammaire de réseau. Nous résumerons ensuite rapidement les approches d'Emerson et Namjoshi de Szymanski et Vidal d'Ip et Dill et de Sistla et German qui proposent des sous-classes décidables.

6.3.1 Approche de Clarke, Grumberg et Jha

Dans [CGJ95] Clarke Grumberg et Jha se posent comme objectif de résoudre le problème général de la vérification des grammaires de réseau en utilisant des techniques d'extrapolation. Le problème étant indécidable leur algorithme peut éventuellement échouer.

Les propriétés d'une grammaire de réseau sont spécifiées à l'aide d'un langage de spécification T_D (voir section 6.2.1 page 62). A partir de l'automate D ils construisent un automate abstrait pour chaque réseau de processus généré par la grammaire. Pour cela ils considèrent des réseaux décrits par des langages (de façon similaire à la spécification D) et calculent une abstraction de cette description basée sur la construction du monoïde syntaxique [Eil74] : cette construction consiste à définir une relation d'équivalence entre les états basée sur l'idée qu'un mot τ du langage de spécification ($\tau \in T_D$) peut être vu comme une fonction sur les ensembles d'états d'un automate. Deux états sont alors équivalents s'ils induisent la même fonction sur l'automate D.

Plus formellement le monoïde syntaxique se construit de la manière suivante : soit $D = (\Theta, \theta_0, Q, \emptyset, \delta, F)$ un automate reconnaissant un langage de spécification. On rappelle que cet automate reconnaît des langages sur Q où Q est l'ensemble des états de tous les processus terminaux de la grammaire. La fonction induite par un mot fini $\tau = (\tau_0, \tau_1, \ldots, \tau_{n-1}) \in T_D$ sur Q est définie par

$$f_\tau(\theta) \;=\; \theta' \quad \Longleftrightarrow \quad \theta \xrightarrow{\tau} \theta' \;=\; \theta \xrightarrow{\tau_0} \theta_0 \xrightarrow{\tau_1} \ldots \xrightarrow{\tau_{n-1}} \theta'$$

Un état d'un réseau est un tuple $\tilde{q} = (q_1, \ldots, q_n)$ i.e. un mot de Q^*. Deux états $\tilde{q}$ et $\tilde{q}'$ de Q^* sont alors équivalents si et seulement si $f_{\tilde{q}} = f_{\tilde{q}'}$. La relation d'équivalence ainsi obtenue permet de construire un automate abstrait. De même ils définissent une fonction de composition abstraite et obtiennent le résultat suivant :

Proposition 6.1 :

Si le système abstrait vérifie les équations d'induction le système concret vérifie la propriété spécifiée par D. □

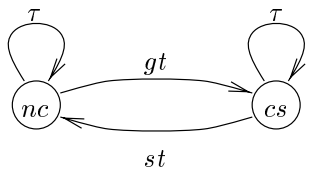


FIG. 6.3 – Processus P_0 si $q^0 = cs$; processus P si $q^0 = nc$.

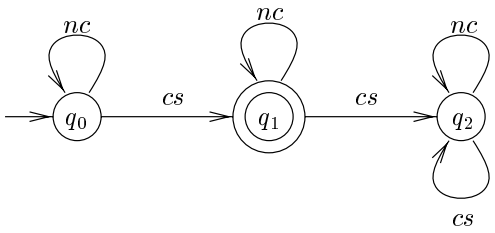


FIG. 6.4 – Propriété d'exclusion mutuelle.

Exemple 6.5 :

Cet exemple utilisé dans [CGJ95] sera également repris section 10.1 page 109. On considère un algorithme d'exclusion mutuelle à jeton circulant défini par la grammaire $\mathcal{G} =< \{P, P_0\}, R, \mathcal{P}, \mathcal{S} >$ où

$$\begin{array}{lcccl} \mathcal{P} & : & \mathcal{S} & \rightarrow & P_0 \| R \\ & & R & \rightarrow & P \| R \\ & & R & \rightarrow & P \end{array}$$

P et P_0 sont définis sur l'ensemble d'états $\{nc, cs\}$ et l'ensemble d'événements $\{\tau, gt, st\}$ (pour respectivement aucune action "*get_token*" et "*send_token*"). Les automates sont donnés figure 6.3 L'automate décrivant l'exclusion mutuelle est donné figure 6.4 : il reconnaît le langage $T_D = nc^*.cs.nc^*$. Cet automate induit 3 fonctions définies comme précédemment qui sont

$$\begin{array}{lcll} f_1 & = & \{ \ (q_0,q_0),(q_1,q_1),(q_2,q_2) \ \} & \text{représentant les mots de } nc^*. \\ f_2 & = & \{ \ (q_0,q_1),(q_1,q_2),(q_2,q_2) \ \} & \text{représentant les mots de } nc^*.cs.nc^*. \\ f_3 & = & \{ \ (q_0,q_2),(q_1,q_2),(q_2,q_2) \ \} & \text{représentant les mots de} \\ & & & nc^*.cs.nc^*.cs.(cs+nc)^*. \end{array}$$

Considérons un réseau de taille 3 i.e. $P_0 \| P \| P$. Son graphe d'accessibilité est présenté figure 6.5. Le seul état accessible de l'automate abstrait est $nc^*.cs.nc^*$. On vérifie que l'automate abstrait obtenu est (trivialement) un invariant du réseau qui vérifie l'exclusion mutuelle. □

Commentaires : Ces résultats sont très intéressants. Cette technique est la première réellement capable de résoudre des problèmes non triviaux comme la vérification de réseaux paramétrés générés par des grammaires.

- La méthode est en théorie très générale. Elle peut s'appliquer à tout type de grammaire.
- La technique d'abstraction est élégante et peut être utilisée soit pour réduire la taille d'un réseau soit pour trouver un invariant. Elle permet de vérifier des propriétés exprimées dans la logique ∀CTL.

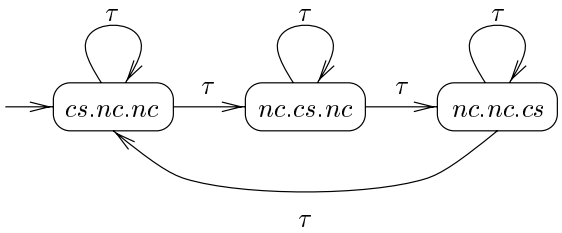


FIG. 6.5 – États accessibles de $P_0 \| P \| P$.

En pratique, cette technique a néanmoins un certain nombre de désavantages :

- Comme nous l'avons déjà dit à la section 6.2.1, page 62, la technique des langages de spécification ne semble applicable qu'à certains cas simples, surtout dans le cas linéaire. Il serait intéressant d'effectuer une étude de cas sur des exemples moins triviaux que l'exclusion mutuelle.
- L'abstraction proposée n'est pas un élargissement dans le sens où nous l'utiliserons section 8.3, page 90. Elle réduit bien la taille du réseau modulo la propriété à démontrer, mais elle ne prend pas en compte la structure de la grammaire. Son application pour déterminer un invariant reste donc assez aléatoire : elle fonctionne, ou elle ne fonctionne pas.

6.3.2 Approche d'Emerson et de Namjoshi pour les anneaux de processus

Emerson et Namjoshi s'intéressent dans [EN95] au cas particulier des anneaux à jeton circulant (ou "token rings"). Ils ont l'idée que la structure d'un tel réseau est suffisante pour généraliser le cas n au cas $n+1$ (voir également cette idée dans [CGB86]).

Ils considèrent donc un réseau de processus en anneau, dans lequel la seule communication entre processus s'effectue à l'aide d'un jeton circulant, et tel que :

- Initialement, le jeton est donné à un processus, choisi de façon aléatoire.
- A tout moment, un seul processus a le jeton.
- Un processus peut avoir des transitions autorisées indépendemment de la possession du jeton. Dans ce cas, le processus est libre d'effectuer une telle transition.
- Le processus peut être bloqué par des transitions nécessitant la possession du jeton. Dès qu'il entre en possession du jeton, il devient libre d'effectuer ces transitions.
- Le processus possédant le jeton ne peut le garder indéfiniment.

Sur de tels systèmes, ils montrent qu'une propriété est satisfaite par tous les réseaux, de toutes tailles si et seulement si elle est satisfaite sur un réseau d'une taille donnée, appelée raccourci (ou "cutoff").

Plus précisément, ils considèrent les propriétés exprimées par des formules de la logique temporelle arborescente CTL* sans l'opérateur *next time* X. Soit g_i une formule concernant le $i^{\text{ème}}$ processus, et $g_{i,j}$ une formule concernant les $i^{\text{ème}}$ et $j^{\text{ème}}$ processus. Alors

1. $\bigwedge_i g_i$ a un raccourci de 2.
2. $\bigwedge_i g_{i,i+1}$ a un raccourci de 3.
3. $\bigwedge_{i \neq j} \forall g'_{i,j}$, où $g'_{i,j}$ est une formule sans quantificateur de chemin ($\forall$ ou $\exists$), a un raccourci de 4.

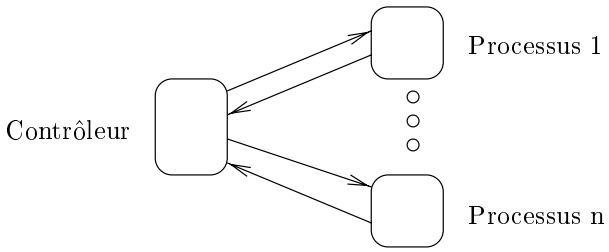


FIG. 6.6 – Un contrôleur et ses processus

4. $\bigwedge_{i \neq j} \exists g'_{i,j}$Γoù $g'_{i,j}$ est une formule sans quantificateur de chemin (∀ ou ∃)Γa un raccourci de 6.

Ce résultat est fort intéressant car il permet de vérifier sans intervention humaine (à l'aide d'outils comme ALDEBARAN [FGK+96]Γou le CONCURRENCY WORKBENCH [CPS89])Γun nombre important de réseaux. AinsiΓnotre premier exemple de calcul de point fixeΓun jeton circulant simple (voir section 10.1Γpage 109)Γaurait pu être vérifié par cette méthode en vérifiant la propriété d'exclusion mutuelle sur un réseau de taille 4. Ce résultat n'est bien entendu pas suffisant : les autres exemples traités dans ce document ne peuvent pas être simplement ramenés à ce cas.

Ce résultat pourrait néanmoins être utilisé avec les résultats décrits plus loin sur les calculs de points fixes. Le résultat d'Emerson et Namjoshi n'est valable que s'il y a exactement un jeton dans le réseau. Dans certains casΓnotre outil (voir partie III) est capable de parvenir à ce résultat.

6.3.3 Approche d'Emerson et de Namjoshi pour les systèmes parallèles

Dans [EN96]ΓEmerson et Namjoshi s'intéressent aux systèmes composés d'un nombre arbitraire de processus : ils considèrent un système constitué d'un contrôleur et de n processus identiques (dits processus utilisateurs)Γà un renommage des variables près.

Chaque processus ne peut communiquer qu'avec le contrôleur (les communications inter-processus ne sont pas possiblesΓvoir figure 6.6). Ce modèle se distingue donc de celui présenté dans ce document [CGJ95]Γoù les communications entre processus sont décrites par une grammaire de réseau. Ces deux approches sont en fait complémentaires :

- Les grammaires de réseau permettent de décrire des structures complexes de réseaux de processus (en anneauΓarborescentΓetc ...).
- Le modèle d'Emerson et Namjoshi permet de décrire un ensemble de processus mobiles pouvant communiquer avec un processus maître. Il peut par exemple s'appliquer dans le cas de la téléphonie mobile.

Le contrôleur C et chaque processus utilisateur P_i sont codés par des automates :

$$C = (Q_C, q_C^0, I_C, O_C, \delta_C, F_C) \quad \text{et} \quad P_i = (Q_i, q_i^0, I_i, O_i, \delta_i, F_i)$$

où les Q_i sont tous deux à deux disjoints et isomorphes à un ensemble d'états QΓet où tous les q_i^0ΓI_iΓO_iΓδ_i et F_i sont égaux à un renommage près. Ils considèrent alors l'automate abstrait A dont chaque état est un couple $(q, \tilde{q})$Γoù $q \in Q_C$ est un état du contrôleur et où $\tilde{q} \subseteq Q$ est un ensemble d'états des processus utilisateurs. On dit alors que A est dans l'état $(q, \tilde{q})$ si le contrôleur C est dans l'état q et si on a $q_i \in \tilde{q}$ si et seulement s'il existe au moins un processus utilisateur P_i dans un état isomorphe à q_i. En d'autres termesΓl'automate abstrait se "souvient" qu'il existe des processus dans un certain étatΓmais il ne se "souvient" pas combien. L'automate abstrait A peut alors effectuer trois types de transition :

- Si $q \to q'$ est une transition interne au contrôleurΓA peut effectuer la transition $(q, \tilde{q}) \to (q', \tilde{q})$.

– Si $q_i \to q_i'$ est une transition interne à un processus P_iΓA peut effectuer les transitions

$$\begin{array}{lcll} (q, \tilde{q} \cup \{q_i\}) & \to & (q, \tilde{q} \cup \{q_i, q_i'\}) & \text{et} \\ (q, \tilde{q} \cup \{q_i\}) & \to & (q, \tilde{q} \cup \{q_i'\}) & \end{array}$$

La première transition correspond à un système dans lequel plusieurs processus utilisateurs sont dans l'état q_i. Un de ces processus passe dans l'état q_i'Γles autres restent dans l'état q_i. La deuxième transition correspond au contraire à un système dans lequel un seul processus est dans l'état q_i. Après la transitionΓil n'y a plus de processus dans cet état.

– Si $q \to q'$ et $q_i \to q_i'$ sont respectivement des transitions du contrôleur et d'un processus utilisateur dans lesquelles les mêmes événements interviennentΓl'automate abstrait A peut effectuer les transitions

$$\begin{array}{lcll} (q, \tilde{q} \cup \{q_i\}) & \to & (q', \tilde{q} \cup \{q_i, q_i'\}) & \text{et} \\ (q, \tilde{q} \cup \{q_i\}) & \to & (q', \tilde{q} \cup \{q_i'\}) & \end{array}$$

Emerson et Namjoshi montrent dans ce modèle les résultats suivants :

– Toute formule de la logique temporelle linéaire satisfaite par l'automate abstrait A sur tous ses chemins est également satisfaite sur tous les chemins de toutes les instances du systèmeΓ quelle que soit sa taille.

Il est à noter que cette proposition n'est pas une équivalence : montrer qu'il existe un n tel que le système de taille n ne vérifie pas une propriété est plus difficile. Il existe en effet des exécutions de l'automate abstrait A qui ne correspondent pas à une exécution d'un système concret. Les exécutions de A qui correspondent à une exécution d'un système concret sont appelées des chemins corrects (“good path”). Les résultats suivants portent sur une logique arborescente.

– Tout chemin fini de A est correct.

– Une formule $\forall\varphi$ (“pour tout chemin φ”) est fausse si et seulement s'il existe un bon chemin de l'automate abstrait A violant φ.

L'existence de bon chemin dans A est équivalente à l'existence de boucles acceptrices (qui contiennent au moins un état accepteur).

– La symétrie du problème permetΓcomme dans [ID93ΓID96] (voir section 6.3.5Γpage 69) de prouver des propriétés sur les processus utilisateurs. AinsiΓ

  – $\bigwedge_i \forall\varphi(i)$ est vraie si et seulement si $\forall\varphi(0)$ est vraie.
  – $\bigwedge_{i \neq j} \forall\varphi(i,j)$ est vraie si et seulement si $\forall\varphi(0,1)$ est vraie.

6.3.4 Approche de Szymanski et de Vidal

Dans [SV94]ΓSzymanski et Vidal utilisent la symétrie de programmes parallèles pour prouver automatiquement certaines propriétés (l'idée se rapproche ainsi de celle de [ID93ΓID96]). Pour celaΓils restreignent l'usage des variables de communication (“shared variables”) de la manière suivante : l'identificateur d'un processus apparaissant dans une condition doit être une variable logique quantifiée universellement ou existentiellement. Comme dans [EN96]Γils construisent un diagramme d'états de leur système. Si chaque processus est représenté par un automate $P = (Q, q^0, I, O, \delta, F)$Γun état du diagramme d'états (qu'Emerson et Namjoshi appellent automate abstrait) est un sous-ensemble $\tilde{q} \subseteq Q$: on a alors $q \in \tilde{q}$ si et seulement s'il existe au moins un processus dans l'état q (voir section précédente).

Plus précisémentΓles processus sont définis par des programmes qui possèdent les propriétés suivantes :

– Les variables de communication n'appartiennent qu'à un seul processus. Tous les processus peuvent les lireΓmais un seul peut les modifier.

- Les conditions apparaissant dans un test **if** ou une itération **while** ne peuvent contenir que les expressions booléennes suivantes :
  - toute expression portant sur les propres variables du processus.
  - toute expression de la forme $(\forall i \in \pi, \varphi(i))$ ou $(\exists i \in \pi, \varphi(i))$ où $\varphi(i)$ est un prédicat défini sur les variables de communication appartenant au processus i et où π est soit l'ensemble de tous les processus soit un ensemble de processus que précède le processus courant dans un certain ordre total sur les processus.

Les algorithmes classiques de vérification par exploration du graphe d'état peuvent alors être employés pour prouver des propriétés sur tous les systèmes indépendemment de leur taille.

6.3.5 Approche d'Ip et de Dill

Dans [ID93 ID96] Ip et Dill tentent de limiter l'explosion du nombre des états en utilisant les symétries d'un réseau de n processus parallèles. Considérant une liste d'actions ou de variables concernant des processus ils remarquent que l'ordre de cette liste n'intervient pas dans la satisfaction d'une propriété. Ils définissent un nouveau type de donnée : les ensembles d'entiers ("scalarset"). La non symétrie des processus due à l'ordre des processus peut ainsi disparaître et il devient possible de définir des automorphismes sur les états d'un graphe puis un graphe quotient sur lequel la propriété d'accessibilité est préservée.

Sur certains exemples ils ont constaté expérimentalement que le graphe quotient de tous réseaux de taille $n \geq 2$ était isomorphe au graphe quotient de taille $n = 2$. Pour ces exemples ces techniques permettent donc de résoudre le problème de la vérification de propriétés sur les réseaux paramétrés. Plus précisément ils montrent que ces techniques sont applicables à des programmes pour lesquels les ensembles d'entiers de données ("data scalarset") ne sont pas utilisés dans des indices de tableau ou des boucles **for**.

Il serait intéressant de déterminer les rapports entre cette approche et la précédente. Celle de Ip et Dill a forcément le désavantage de ne pas être spécialisée dans la preuve des réseaux paramétrés : il semble pourtant que les idées utilisées sont très proches de celles de Szymanski et Vidal.

6.3.6 Approche de Sistla et German (comptage des processus)

Dans [SG87] Sistla et German considèrent un modèle ou deux processus peuvent se synchroniser s'ils acceptent des signaux complémentaires (x et $\overline{x}$).

n processus identiques : Ils considèrent dans un premier temps n processus identiques. Pour cela ils calculent l'ensemble des états d'un processus (dits accessibles) tels qu'il existe un nombre de processus pour lequel cet état est atteint. Un état peut ainsi être accessible dans leur terminologie s'il est accessible dans un système constitué de 4 processus même s'il ne l'est pas pour un système constitué de 2 processus.

Dans ce modèle il est possible de prouver des propriétés locales (ils calculent l'ensembles des états accessibles d'un processus) mais pas de propriétés globales (ils montrent en particulier qu'un nombre arbitrairement grand de processus peut se trouver dans tout état accessible).

Un processus contrôleur et n processus identiques : Dans un second temps ils étendent ce modèle en considérant un seul processus de contrôle et un nombre arbitraire de processus utilisateurs identiques. Ils pensent ainsi pouvoir modéliser un certain nombre de protocoles ainsi que certains algorithmes d'exclusion mutuelle.

Leur système est modélisé par un vecteur d'entiers décrivant le nombre de processus utilisateurs se trouvant dans un état donné. Une transition est autorisée s'il existe deux processus dans des états permettant de se synchroniser donc si les deux variables comptant le nombre de processus dans ces états sont non nulles. Ils obtiennent ainsi un système à commandes gardées.

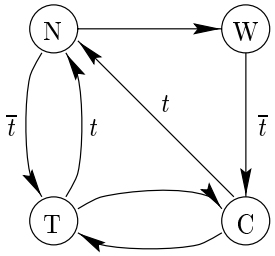


FIG. 6.7 – Un processus du jeton circulant de Sistla et German

Application aux réseaux de processus : Les algorithmes précédents peuvent être appliqués dans certains cas pour modéliser des réseaux de processus où la structure du réseau est abstraite. Sistla et German prennent l'exemple des réseaux linéairesΓoù chaque processus ne peut communiquer qu'avec son voisin. En abstrayant la structure du réseau on obtient un système où chaque processus peut communiquer avec n'importe quel autre processus dans le réseau.

La figure 6.7 montre la description d'un processus. Le signal t signifie que le processus émet le jetonΓle signal $\overline{t}$ qu'il le reçoit. Ce processus a quatre états : T ("token")Γil a le jeton; W ("waiting")Γil attend; C ("critical")Γil est dans sa section critique; et N ("neutral") dans les autres cas.

Ce système est donc décrit par 4 variables entièresΓn_TΓn_WΓn_C et n_N comptant le nombre de processus dans chacun de ces 4 états. Ces variables sont définies par un système à commandes gardées de la manière suivante :

– Un processus effectue une action interne :

$$\begin{array}{lcll} n_N \geq 1 & \longrightarrow & n_N := n_N - 1;\ n_W := n_W + 1 & N \ \rightarrow \ W \\ n_T \geq 1 & \longrightarrow & n_T := n_T - 1;\ n_C := n_C + 1 & T \ \rightarrow \ C \\ n_C \geq 1 & \longrightarrow & n_C := n_C - 1;\ n_T := n_T + 1 & C \ \rightarrow \ T \end{array}$$

– Deux processus communiquent :

$$\begin{array}{lcll} n_N \geq 1 \wedge n_T \geq 1 & \longrightarrow & SKIP & N \rightarrow T \ \text{et} \ T \rightarrow N \\ n_N \geq 1 \wedge n_C \geq 1 & \longrightarrow & n_C := n_C - 1;\ n_T := n_T + 1 & N \rightarrow T \ \text{et} \ C \rightarrow N \\ n_W \geq 1 \wedge n_T \geq 1 & \longrightarrow & n_W := n_W - 1;\ n_T := n_T - 1; & W \rightarrow C \ \text{et} \ T \rightarrow N \\ & & n_C := n_C + 1;\ n_N := n_N + 1 & \\ n_W \geq 1 \wedge n_C \geq 1 & \longrightarrow & n_W := n_W - 1;\ n_N := n_N + 1 & W \rightarrow C \ \text{et} \ C \rightarrow N \end{array}$$

En partant de la condition initiale $(n_N = N, n_T = 0, n_W = 0, n_C = 0)$Γon montre aisément à la mainΓou en utilisant les techniques de [Rac78ΓHMP95]Γque $n_C \leq 1$Γi.e.Γque l'exclusion mutuelle est vérifiée.

Chapitre 7

Réseaux de processus et réseaux d'observateurs

7.1 Spécification et preuves à l'aide d'observateurs

7.1.1 Observateurs d'un réseau de processus

Dans cette sectionΓnous allons montrer que la notion d'observateur (voir section 1.2Γpage 19 et [HLR93]) peut facilement s'étendre aux réseaux de processus.

A chaque processus dans le réseauΓon peut associer un observateur lisant les entrées/sorties du processus observé avec les observations provenant des autres observateurs du réseau.

Par exempleΓconsidérons un réseau linéaire $P\|P\|\ldots\|P$ de processus identiques. Chaque processus P émet un signal u (pour "use") quand il utilise une ressource. Supposons que nous voulions exprimer la propriété d'exclusion mutuelle (au plus un processus utilise la ressource à un instant donné).

A chaque processusΓon associe un observateur recevant le signal u du processus P observé et les signaux émis par son successeur de droite dans le réseau (voir figure 7.1.a). Chaque observateur émet deux signaux : α est émis lorsqu'une violation de l'exclusion mutuelle est détectéeΓet ν est émis lorsque la ressource est utilisée soit par le processusΓsoit par un des ses successeurs de droite dans le réseau. Un tel observateur peut être décrit par le système suivant d'équations booléennes :

$$\alpha = \alpha' \vee (\nu' \wedge u) \quad \text{et} \quad \nu = \nu' \vee u$$

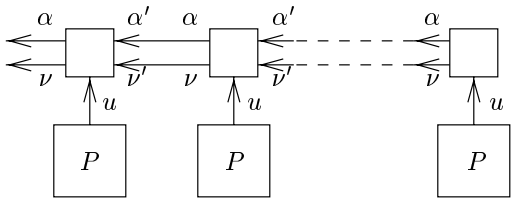
AinsiΓun réseau satisfait la propriété d'exclusion mutuelle si et seulement si l'observateur le plus à gauche n'émet jamais le signal d'alarme α.

Il est à noter que cette technique peut être naturellement étendue à des structures plus complexes : par exempleΓsi le réseau a une structure arborescenteΓon peut construire un observateur recevant les signaux émis par ses "fils" (voir figure 7.1.b) :

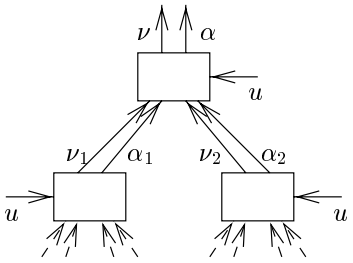
$$\alpha = \alpha_1 \vee \alpha_2 \vee (\nu_1 \wedge \nu_2) \vee ((\nu_1 \vee \nu_2) \wedge u) \quad \text{et} \quad \nu = \nu_1 \vee \nu_2 \vee u$$

Implication : Les observateurs de réseaux nous permettent d'exprimer une propriété sur un nombre infini de processus à l'aide d'un nombre borné de signaux (dans l'exemple précédent α et ν). AinsiΓdans une grammaire de réseauΓtous les non terminaux pourront être décrits avec un nombre fixe de signaux.

La section suivante compare le pouvoir d'expression des langages de spécification et des observateurs de réseaux.



(a) Structure linéaire



(b) Structure arborescente

FIG. 7.1 – Observateurs de réseaux

7.1.2 Observateurs de réseaux et langages de spécification

Théorème 7.1 :

Le pouvoir d'expression des observateurs de réseaux est strictement plus fort que celui des langages de spécification. □

Preuve :

(1) Tout langage de spécification peut être décrit par des observateurs :

Soit T_D un langage de spécification. Comme ce langage est régulierΠil peut être exprimé à l'aide d'une expression régulière ρ_D. On rappelle qu'une expression régulière suit la syntaxe suivante

$$\rho \quad ::= \quad x \mid \rho + \rho \mid \rho^* \mid \rho.\rho$$

correspondant à la sémantique

$S(x)$	$::=$	$\{x\}$	
$S(\rho_1 + \rho_2)$	$::=$	$\rho_1 \cup \rho_2$	Union des langages
$S(\rho^*)$	$::=$	$\rho.\rho \ldots \rho$	ρ répété 0 ou plusieurs fois
$S(\rho_1.\rho_2)$	$::=$	$\rho_1.\rho_2$	Concaténation des langages

On attribue à ρ_D et à chacune de ses sous-expressions un signal d'observation. On note ainsi α^ρ le signal associé à l'expression régulière ρ (α^ρ est émis par un réseau si et seulement s'il satisfait ρ).

Les expressions régulières x et $\rho_1 + \rho_2$ sont aisément exprimées par récurrence à l'aide de signaux :

- $\rho = x$

Seul un réseau constitué d'un unique processus peut être dans l'état ρ. Le signal d'observation est émis si le processus est dans l'état x.

$$\alpha^x = x$$

- $\rho = \rho_1 + \rho_2$
  L'union des langages se traduit par la disjonction des signaux.

$$\alpha^{\rho_1+\rho_2} = \alpha^{\rho_1} \vee \alpha^{\rho_2}$$

Le calcul des signaux correspondant aux expressions régulières ρ^* et $\rho_1.\rho_2$ dépend des règles de production de la grammaire. Pour simplifier, nous nous limitons au cas binaire (toutes les fonctions de composition de la grammaire sont d'arité 2) : le cas général s'en déduit trivialement. Ainsi, toute règle de production peut s'écrire

$$\begin{array}{lcl} A & \rightarrow & B \quad \text{ou} \\ A & \rightarrow & B \| C \end{array}$$

On construit alors notre observateur par récurrence. Dans le cas de la première règle, les signaux de A prennent les même valeurs que ceux de B. Dans le cas de la deuxième, notons α_A, α_B et α_C les signaux émis respectivement par A, B et C.

- $\rho = \rho_1^*$
  Les deux réseaux B et C sont soit dans l'état ρ_1, soit dans l'état ρ_1^*. Donc

$$\alpha_A^{\rho_1^*} = (\alpha_B^{\rho_1^*} \vee \alpha_B^{\rho_1}) \wedge (\alpha_C^{\rho_1^*} \vee \alpha_C^{\rho_1})$$

- $\rho = \rho_1.\rho_2 \ldots \rho_n$
  où les expressions régulières ρ_i ne sont pas de la forme $\rho'.\rho''$. Pour que le réseau satisfasse ρ, il doit exister un i, $1 \leq i < n$, tel que le sous-réseau B satisfasse $\rho_1.\rho_2 \ldots \rho_i$, et le sous-réseau C $\rho_{i+1}.\rho_{i+2} \ldots \rho_n$. Donc

$$\alpha_A^{\rho_1.\rho_2 \cdots \rho_n} = \bigvee_{1 \leq i < n} \alpha_B^{\rho_1.\rho_2 \cdots \rho_i} \wedge \alpha_C^{\rho_{i+1}.\rho_{i+2} \cdots \rho_n}$$

(2) Exemple exprimable par des observateurs et pas par un langage de spécification.

Un langage de spécification étant régulier, il ne peut pas reconnaître une propriété du type

$$\{\ a^n.b^n \mid \text{pour tout } n\ \}$$

Cette propriété peut s'exprimer à l'aide d'observateurs en construisant un réseau de la manière suivante :

$$\begin{array}{lcl} \mathcal{S} & \rightarrow & R \\ R & \rightarrow & P \| R \| P \\ R & \rightarrow & P \| P \end{array}$$

L'observateur correspondant à la deuxième équation vérifie que le processus de gauche est dans l'état a et le processus de droite dans l'état b. □

Exemple 7.1 :

Reprenons notre problème d'exclusion mutuelle. Les processus élémentaires ont deux états: cs (section critique) et nc (section non critique). La propriété d'exclusion mutuelle est spécifiée par la formule

$$\forall\Box\ (\ \{nc\}^*cs\{nc\}^*\ )$$

On introduit un signal d'observation pour chaque état et chaque sous formule de la formule précédente.

$$\begin{aligned} a &::= cs \\ b &::= nc \\ c &::= \{nc\}^* \\ d &::= \{nc\}^*cs \\ e &::= cs\{nc\}^* \\ f &::= \{nc\}^*cs\{nc\}^* \end{aligned}$$

Pour une grammaire binaire, notre observateur est alors défini par récurrence par les formules suivantes

$$\begin{aligned} c_A &= (b_B \vee c_B) \wedge (b_C \vee c_C) \\ d_A &= c_B \wedge a_C \\ e_A &= a_B \wedge c_C \\ f_A &= (c_B \wedge e_C) \vee (d_B \wedge c_C) \end{aligned}$$

□

Remarques :

1. En pratique, les langages de spécification sont bien adaptés aux réseaux linéaires pour des propriétés simples. Ainsi, l'exclusion mutuelle d'un réseau linéaire est exprimée plus naturellement avec un langage de spécification qu'avec un observateur. Pour des réseaux plus complexes (en particulier pour les réseaux arborescents vus au chapitre 9), les observateurs sont beaucoup plus faciles à utiliser.

2. Ces deux modes de spécifications fonctionnent sur des principes différents. L'expression d'une propriété à l'aide d'observateurs doit se faire en parallèle avec la définition d'un réseau par une grammaire. L'exemple $a^n.b^n$ montre que cette propriété ne peut être spécifiée que sur un certain type de réseau. Au contraire, les langages de spécification peuvent s'appliquer à n'importe quel réseau, quelle que soit sa définition: cette technique semble donc plus générale. Pourtant, outre qu'elle soit moins expressive que la technique des observateurs, nous pensons qu'elle est vraiment contre intuitive pour de nombreuses propriétés et qu'en pratique, les observateurs sont plus facilement utilisables.

7.1.3 Preuve par calcul d'invariant

7.1.3.1 Réseaux linéaires

Kurshan et McMillan d'une part [KM89], et Wolper et Lovinfosse d'autre part [WL89], ont proposé une méthode inductive pour résoudre le problème de la vérification des réseaux dans le cas

des réseaux linéaires. Cette technique nécessite de définir une relation d'ordre $\preceq$ sur les processus compatible avec la propriété φ i.e. vérifiant pour tout processus P et P'

$$\left(\; P \preceq P' \; \wedge \; P' \models \varphi \; \right) \implies \left(\; P \models \varphi \; \right)$$

Il est alors suffisant de trouver un processus I appelé invariant de réseau satisfaisant φ et plus grand que tous les processus générés par la grammaire.

$$\left(\; \forall P \in \mathcal{G}, \; P \preceq I \; \right) \; \wedge \; \left(\; I \models \varphi \; \right)$$

Pour un réseau constitué de n processus P identiques

$$\underbrace{P \| \ldots \| P}_{n \text{ fois}}$$

il est suffisant que I satisfasse les propriétés suivantes

[INIT]	P	$\preceq$	I	Initiation
[INDUC]	$P \times I$	$\preceq$	I	Induction
[SAT]	I	$\models$	φ	Satisfaction

Propriété 7.1 :

(**Induction Theorem for Processes** [KM89]). S'il est existe un processus I satisfaisant les propriétés précédentes le réseau vérifie la propriété φ quelle que soit sa taille.
□

7.1.3.2 Réseaux quelconques

Le cas linéaire s'étend trivialement au cas des grammaires de réseau. La méthode consiste à trouver pour chaque symbole non terminal $P \in N$ un processus I_P appelé invariant du symbole P permettant de vérifier la propriété φ. Par exemple si on se limite à des opérateurs de composition binaire ces invariants doivent vérifier

$$\begin{array}{rrcll} & I_S & \models & \varphi & \\ \forall R = (A \to B \|_i C) \in \mathcal{P}, & I_B \|_i I_C & \preceq & I_A & \text{si } B, C \in N \\ & B \|_i I_C & \preceq & I_A & \text{si } B \in T, C \in N \\ & I_B \|_i C & \preceq & I_A & \text{si } B \in N, C \in T \\ & B \|_i C & \preceq & I_A & \text{si } B, C \in T \end{array}$$

Pour simplifier l'écriture et pour tout processus P notons I_P

- L'invariant associé à P si P est un processus non terminal.
- Le processus P lui-même si P est un processus terminal.

Les inéquations précédentes peuvent alors se récrire dans le cas général

$$\begin{array}{ll} I_S \models \varphi & \\ \forall R = (P \to P_1 \|_i \ldots \|_i P_n) \in \mathcal{P}, & I_{P_1} \|_i \ldots \|_i I_{P_n} \preceq I_P \end{array}$$

Dans les sections suivantes nous proposerons un formalisme pour récrire de façon pratique ces équations puis nous proposerons une technique de calcul d'invariants par point fixe.

7.2 Sémantique de traces

Dans cette section nous allons récrire les inéquations d'induction de la section précédente en utilisant une sémantique de traces.

Ainsi tout processus est décrit par le langage de ses traces. De même (voir section 1.2.1 page 19) une propriété est un ensemble de traces. Nous nous limitons dorénavant aux propriétés de sûreté. On utilise l'inclusion des traces comme opérateur de comparaison $\preceq$.

Les deux sections suivantes introduisent les opérateurs de base nécessaires pour l'utilisation de la sémantique de traces.

7.2.1 Opérations sur les traces

Introduisons l'opérateur $\odot$ sur les ensembles de tracesΓafin de traduire la composition de processus.

Définition 7.1 :

Soient X et X' deux ensembles disjoints de signauxΓet soient $\tau \in X^\infty$ et $\tau' \in X'^\infty$ deux mots de même longueur. AlorsΓ$\tau \odot \tau'$ est une trace sur $X \cup X'$Γdéfinies par

$$\tau \odot \tau' = (\tau_0 \cup \tau'_0, \ldots, \tau_n \cup \tau'_n \ldots).$$

□

Cette opération est étendue aux langages: soient $T \subseteq X^\infty$ et $T' \subseteq X'^\infty$Γdeux langages respectivement sur X et X'. AlorsΓ

$$T \odot T' \ = \ \{ \ \tau \odot \tau' \mid \tau \in T, \tau' \in T', |\tau| = |\tau'| \ \}$$

Nous écrirons souvent $T \odot X'^\infty$ pour considérer T comme un sous-ensemble de $\left(\ X \cup X' \ \right)^\infty$Γoù les signaux de X' ne sont pas contraints.

Définition 7.2 :

Soient X et X' deux ensembles de signaux de même cardinalitéΓreliés l'un à l'autre par une bijection ϕ. AlorsΓpour tout mot $\tau = (\tau_0, \ldots, \tau_n, \ldots)$ sur XΓon définit $\tau[X/X']$ comme étant le mot $(\tau'_0, \ldots, \tau'_n, \ldots)$ avec $\forall i, \tau'_i = \{\phi(x) \mid x \in \tau_i\}$. □

On étend de même cette opération aux langages.

$$T[X/X'] \ = \ \{ \ \tau[X/X'] \mid \tau \in T \ \}$$

7.2.2 Composition synchrone

Si nos langages représentent des traces de processusΓil est nécessaire d'introduire des opérateurs sur les langages permettant de traduire les opérations de composition synchrone sur les processus.

Définition 7.3 :

Soient X et X' deux ensembles de signauxΓet $T \subseteq X^\infty$Γ$T' \subseteq X'^\infty$ deux langages. On définit alors

$$\begin{aligned} T \otimes T' &= (T \odot \left(\ X' \setminus X \ \right)^\infty) \cap (T' \odot \left(\ X \setminus X' \ \right)^\infty) \\ T \oplus T' &= (T \odot \left(\ X' \setminus X \ \right)^\infty) \cup (T' \odot \left(\ X \setminus X' \ \right)^\infty) \end{aligned}$$

□

$T \otimes T'$ est l'ensemble des mots qui s'accordent sur les signaux de $T \cap T'$. En d'autres termesΓsi P et P' sont deux processusΓ$T_P \otimes T'_P$ représente l'ensemble des traces du produit synchrone de P et P'.

$$T_{P \times P'} \ = \ T_P \otimes T_{P'}$$

De mêmeΓ$T \oplus T'$ est l'union de T et T' en tant que sous-ensemble de $\left(\ X \cup X' \ \right)^\infty$. $T_P \oplus T'_P$ représente donc l'ensemble des traces du produit non synchronisé de P et P'.

7.2.3 Propriétés des opérateurs d'abstraction

Définissons les opérateurs d'abstraction de signaux.

Définition 7.4 :

Soit $T \subseteq X^\infty$ un langage sur $X\Gamma$et soit $X' \subseteq X$ un sous-ensemble de X. Les abstractions existentielle et universelle de T par rapport aux variables de $X'\Gamma$notées respectivement $\exists X', T$ et $\forall X', T\Gamma$sont définies par

$$\begin{aligned} \exists X', T &= \{\ \tau \in (X \setminus X')^\infty \mid \exists \tau' \in X'^{|\tau|},\ \tau \odot \tau' \in T\ \} \\ \forall X', T &= \{\ \tau \in (X \setminus X')^\infty \mid \forall \tau' \in X'^{|\tau|},\ \tau \odot \tau' \in T\ \} \end{aligned}$$

□

$\exists X', T_P$ est l'ensemble des traces d'un processus pour lequel tous les signaux de X' sont considérés internes (i.e.Γcachés). $\forall X', T$ est considéréΓpar dualitéΓcomme

$$\forall X', T = \overline{\exists X', \overline{T}}$$

Soit X un ensemble fini de signauxΓet soit $X' \subseteq X$ un sous-ensemble de X. Soit $T \subseteq X^\infty$ un langage sur X.

Propriété 7.2 :

$$(\ \forall X', T\ ) \odot (\ X \setminus X'\ )^\infty \ \subseteq\ T\ \subseteq\ (\ \exists X', T\ ) \odot (\ X \setminus X'\ )^\infty$$

□

Preuve :

Les preuves des deux inclusions précédentes sont duales.

$$\begin{aligned} (\ \forall X', T\ ) \odot (\ X \setminus X'\ )^\infty &= \{\ \tau \in (\ X \setminus X'\ )^\infty \mid \forall \tau' \in X'^\infty,\ \tau \odot \tau' \in T\ \} \\ &\quad \odot (\ X \setminus X'\ )^\infty \\ &= \{\ \tau \odot \tau'' \mid \forall \tau' \in X'^\infty,\ \tau \odot \tau' \in T \text{ et } \tau'' \in X'^\infty\ \} \\ &\subseteq T \end{aligned}$$

□

Propriété 7.3 :

$$\begin{aligned} \exists X', T_1 \cup T_2 &= (\ \exists X', T_1\ ) \cup (\ \exists X', T_2\ ) \\ \forall X', T_1 \cap T_2 &= (\ \forall X', T_1\ ) \cap (\ \forall X', T_2\ ) \end{aligned}$$

□

Preuve :

Les preuves de ces deux propriétés sont duales en remplaçant $\exists$ par $\forall$ et $\cup$ par $\cap$.

$$\begin{aligned}
\exists X',\ (T_1 \cup T_2) &= \{\ \tau \in \left(\ X \setminus X'\ \right)^{\infty} \mid \exists \tau' \in X'^{|\tau|},\ \tau \odot \tau' \in T_1 \cup T_2\ \} \\
&= \{\ \tau \in \left(\ X \setminus X'\ \right)^{\infty} \mid \exists \tau' \in X'^{|\tau|},\ \tau \odot \tau' \in T_1 \text{ ou } \tau \odot \tau' \in T_2\ \} \\
&= \left\{ \begin{array}{rl} \tau \in \left(\ X \setminus X'\ \right)^{\infty} \mid & \exists \tau' \in X'^{|\tau|},\ \tau \odot \tau' \in T_1 \text{ ou} \\ & \exists \tau' \in X'^{|\tau|},\ \tau \odot \tau' \in T_2 \end{array} \right\} \\
&= (\exists X',\ T_1) \cup (\exists X',\ T_2)
\end{aligned}$$

□

Remarque : En généralΓ

$$\begin{aligned}
\exists X',\ T_1 \cap T_2 &\neq (\exists X',\ T_1) \cap (\exists X',\ T_2) \\
\exists X',\ \overline{T} &\neq \overline{\exists X',\ T} \\
\forall X',\ T_1 \cup T_2 &\neq (\forall X',\ T_1) \cup (\forall X',\ T_2) \\
\forall X',\ \overline{T} &\neq \overline{\forall X',\ T}
\end{aligned}$$

Ces résultats s'appliquent de même aux opérateurs $\otimes$ et $\oplus$.

Corollaire 7.1 :

$$\begin{aligned}
\exists X', T_1 \oplus T_2 &= \left(\ \exists X', T_1\ \right) \oplus \left(\ \exists X', T_2\ \right) \\
\forall X', T_1 \otimes T_2 &= \left(\ \forall X', T_1\ \right) \otimes \left(\ \forall X', T_2\ \right) \\
\exists X',\ T_1 \otimes T_2 &\neq (\exists X',\ T_1) \otimes (\exists X',\ T_2) \\
\forall X',\ T_1 \oplus T_2 &\neq (\forall X',\ T_1) \oplus (\forall X',\ T_2)
\end{aligned}$$

□

7.2.4 Opérateur de composition quelconque

Un opérateur de composition d'arité n est une fonction qui prend en entrée n processusΓet renvoie un processus. Dans notre sémantique de tracesΓc'est une fonction de $\left(\ X^{\infty}\ \right)^n$ dans X^{∞}. De la même manière qu'on utilise une sémantique de traces pour décrire des processusΓnous allons utiliser une sémantique de traces pour décrire les opérateurs de composition.

Définition 7.5 :

Soient $X\Gamma X_1\Gamma \ldots \Gamma X_n\Gamma n+1$ ensembles de signaux disjoints de même cardinalité. Soit $\|$ un opérateur de composition d'arité n sur X. On appelle trace de $\|$ l'ensemble de traces

$$\begin{aligned}
T_{\|} = \{\ & T_{P_1\|\ldots\|P_n} \otimes \left(\ T_{P_1}[X/X_1] \odot \ldots \odot T_{P_n}[X/X_n]\ \right) \mid \\
& P_1, \ldots, P_n \text{ sont des processus sur } X\ \}
\end{aligned}$$

□

L'ensemble des traces de $\|$ est donc un sous-ensemble de $\left(\ X \cup X_1 \cup \ldots X_n \ \right)^\infty$ tel que pour tout processus $P_1 \Gamma \ldots \Gamma P_n \Gamma$on ait

$$T_{P_1 \| \ldots \| P_n} \ = \ \exists X_1, \ldots, \exists X_n, \ \left(\ T_\| \otimes T_{P_1}[X/X_1] \otimes \ldots \otimes T_{P_n}[X/X_n] \ \right)$$

IntuitivementΓcette formule peut être lue de la façon suivante :

> Une trace τ est une exécution possible de la composition $P_1 \| \ldots \| P_n$ (i.e.$\Gamma \tau \in T_{P_1 \| \ldots \| P_n}$) si et seulement s'il existe n traces $\tau_1 \Gamma \ldots \ \Gamma \tau_n$ (i.e.$\Gamma \exists X_1, \ldots, \exists X_n$) respectivement de $P_1 \Gamma \ldots \ \Gamma P_n$ (i.e.$\Gamma T_{P_1}[X/X_1] \otimes \ldots \otimes T_{P_n}[X/X_n]$) telles que ces n traces soient reliées par l'opérateur de composition $\|$ (i.e.$\Gamma T_\|$).

7.3 Invariants et plus petits points fixes

Soit $\mathcal{G} =< T, N, \mathcal{P}, \mathcal{S} >$ une grammaire de réseauΓet soit φ une propriété. Récrivons les inéquations que doivent vérifier les invariants du réseau à l'aide de notre sémantique de traces.

$$I_\mathcal{S} \ \subseteq \ \varphi$$

$$\forall R = (P \to P_1 \|_i \ldots \|_i P_n) \in \mathcal{P}, \quad \exists X_1, \ldots, X_n, \ T_{\|_i} \otimes \left(\bigotimes_{1 \leq j \leq n} I_{P_j}[X/X_j] \right) \ \subseteq \ I_P \qquad (7.1)$$

On cherche alors à déterminer un vecteur V de langages de traces de taille $n\Gamma$où n est le nombre de processus non terminaux (la taille de T).

$$V \ \in \ \left\{ \ X^\infty \ \right\}^n \qquad V \ = \ \left(\ T_{P_1}, \ldots, T_{P_n} \ \right)$$

Nous utiliserons l'ordre suivant sur les vecteurs de langages :

Définition 7.6 :

$$V = \left(\ T_{P_1}, \ldots, T_{P_n} \ \right) \preceq V' = \left(\ T_{P'_1}, \ldots, T_{P'_n} \ \right) \quad \Longleftrightarrow \quad \forall i, 1 \leq i \leq n, \ T_{P_i} \subseteq T_{P'_i}$$

> De mêmeΓon étend les opérateurs d'union $\cup$ et d'intersection $\cap$ aux vecteurs de langages :
>
> $$\begin{aligned} \left(\ T_{P_1}, \ldots, T_{P_n} \ \right) \cup \left(\ T_{P'_1}, \ldots, T_{P'_n} \ \right) &= \left(\ T_{P_1} \cup T_{P'_1}, \ldots, T_{P_n} \cup T_{P'_n} \ \right) \\ \left(\ T_{P_1}, \ldots, T_{P_n} \ \right) \cap \left(\ T_{P'_1}, \ldots, T_{P'_n} \ \right) &= \left(\ T_{P_1} \cap T_{P'_1}, \ldots, T_{P_n} \cap T_{P'_n} \ \right) \end{aligned}$$
>
> □

Nous dirons que le vecteur $V = \left(\ T_{P_1}, \ldots, T_{P_n} \ \right)$ est un invariant du réseau si les langages $T_{P_1} \Gamma \ldots \Gamma T_{P_n}$ sont des invariants du réseau.

Théorème 7.2 :

> Il existe un invariant du réseau V^m minimal. V^m est le plus petit point fixe de la fonction f définie par
>
> $$f(I_{P_1}, \ldots, I_{P_n}) \ = \ \bigcup_{R \in \mathcal{P}} \exists X_1, \ldots, X_n, \ T_{\|_i} \otimes \left(\bigotimes_{1 \leq j \leq n} I_{P_j}[X/X_j] \right)$$
>
> □

Preuve :

L'inéquation 7.1 peut se récrire

$$\forall R = (P \to P_1 \|_i \ldots \|_i P_n) \in \mathcal{P}, \qquad f_R(I_{P_1}, \ldots, I_{P_n}) \subseteq P$$

où

$$f_R(I_{P_1}, \ldots, I_{P_n}) = \exists X_1, \ldots, X_n,\ T_{\|_i} \otimes \left(\bigotimes_{1 \leq j \leq n} I_{P_j}[X/X_j] \right)$$

La conjonction de l'ensemble des inéquations de ce type produit l'inéquation

$$f(I_{P_1}, \ldots, I_{P_n}) \subseteq (I_{P_1}, \ldots, I_{P_n})$$

où

$$f(I_{P_1}, \ldots, I_{P_n}) = \bigcup_{R \in \mathcal{P}} f_R(I_{P_1}, \ldots, I_{P_n})$$

Si on pose $V = (I_{P_1}, \ldots, I_{P_n})$Γon obtient

$$f(V) \subseteq V$$

où

$$f(V) = \bigcup_{R \in \mathcal{P}} f_R(V)$$

La fonction f est croissanteΓdonc cette inéquation admet un plus petit point fixeΓV^m.
□

Notons I_P^m l'invariant minimal associé à P: $V^m = (\ I_{P_1}^m, \ldots, I_{P_n}^m\ )$. On note en particulier $I_{\mathcal{S}}^m$ l'invariant minimal associé au symbole générant la grammaire.

Corollaire 7.2 :

La grammaire $\mathcal{G}$ vérifie la propriété φ si et seulement si $I_{\mathcal{S}}^m$ vérifie la propriété φ. □

En pratiqueΓV^m est calculé par itération

$$\begin{array}{rclcl} V^0 & = & (I_{P_1}^0, \ldots I_{P_n}^0) & = & (\emptyset, \ldots, \emptyset) \\ V^{i+1} & = & (I_{P_1}^{i+1}, \ldots I_{P_n}^{i+1}) & = & V^i \cup \left(\bigcup_{R \in \mathcal{P}} f_R(V^i) \right) \\ V^m & = & (I_{P_1}^m, \ldots I_{P_n}^m) & = & \lim_{i \to \infty} V^i \end{array}$$

L'indécidabilité de notre problème provient du fait que ce calcul de plus petit point fixe ne converge en général pas en un temps fini. En effetΓles I_P^m représentent l'ensemble des traces de tous les réseaux pouvant être générés par le symbole non terminal P. Les I_P^m sont donc en général des processus à nombre infini d'étatsΓqui ne peuvent pas être représentés par des automates à nombre fini d'états.

On a néanmoins le résultat suivant :

Corollaire 7.3 :

La grammaire $\mathcal{G}$ ne vérifie pas la propriété φ si et seulement si il existe un entier nΓtel $I_{\mathcal{S}}^n$ ne vérifie pas φ. □

Dans les chapitres suivantsΓnous allons utiliser une démarche symétrique. Au lieu de calculer des plus petits points fixesΓdont la taille dépend de celle du réseauΓnous allons essayer de calculer des plus grands points fixesΓdont la taille dépend de celle de la propriété à démontrer.

7.4 Exemple : un arbre de parité

Cette section présente un exemple de calcul en avant dont la limite peut être facilement approchée.

On considère un réseau d'arbres binaires dans lesquels chaque feuille possède un bit de valeur. Cet exemple décrit un algorithme qui calcule la parité des valeurs des feuilles. Il est tiré de [Ull84, CGJ95]. Un arbre binaire est défini par la grammaire de réseau suivante :

$$\mathcal{G} = < \{root, inter, leaf\} , \{\mathcal{S}, SUB\} , \mathcal{P} , \mathcal{S} >$$

où on rappelle que $root$, $inter$ et $leaf$ sont des processus terminaux, $\mathcal{S}$ et SUB sont des non terminaux, et $\mathcal{P}$ est l'ensemble de règles de production suivant :

$$\begin{array}{rcl} \mathcal{S} & \longrightarrow & root \| SUB \| SUB \\ SUB & \longrightarrow & inter \| SUB \| SUB \\ SUB & \longrightarrow & inter \| leaf \| leaf \end{array}$$

L'algorithme est le suivant. La racine $root$ débute une onde en envoyant le signal $readydown$ à ses enfants. Lorsque le signal $readydown$ atteint un processus feuille, la feuille envoie un signal $readyup$ et sa valeur $value$ à son père. Un noeud $inter$ de l'arbre qui reçoit les signaux $readyup$ et $value$ de ces deux fils, envoie à son père le signal $readyup$ et la parité des valeurs de ses enfants (le ou exclusif des $values$). Lorsque le signal $readyup$ atteint la racine, l'onde est terminée. On suppose, pour simplifier l'exemple, que la racine cesse alors toute activité (i.e. n'envoie pas de nouvelles ondes).

Si q est une variable d'état du programme, notons **next** q la valeur de cette variable à l'instant suivant. On note respectivement rd, ru et v pour $readydown$, $readyup$ et $value$, afin de simplifier les équations. Un processus est décrit de la manière suivante.

$$\begin{array}{lcl} \textbf{next}\ iv & = & iv; \\ \textbf{next}\ ru & = & rd \vee ru; \\ \textbf{next}\ v & = & ru \wedge iv; \end{array}$$

(voir figure 7.2). La variable iv (pour "internal value") contient la valeur associée au processus. Elle est initialisée aléatoirement à 0 ou à 1. L'observateur d'un processus possède de plus un signal ov (pour "observed value"), qui transmet de façon instantanée la valeur du processus à l'observateur de la propriété. Bien entendu, ce signal n'est défini que pour l'observation et ne modifie pas le comportement du système.

$$ov = iv;$$

Pour décrire un noeud de l'arbre, on note respectivement lrd, rrd, lru et rru pour $left\ readydown$, $right\ readydown$, $left\ readyup$ et $right\ readyup$ (voir figure 7.3). Le comportement du noeud est décrit par les équations suivantes :

$$\begin{array}{lcl} \textbf{next}\ lrd & = & rd; \\ \textbf{next}\ rrd & = & rd; \\ ru & = & lru \wedge rru; \\ v & = & ru \wedge (lv \oplus rv); \quad \text{où } \oplus \text{ dénote le } \textbf{ou exclusif} \end{array}$$

L'observateur d'un noeud reçoit les deux signaux d'observation olv et orv, pour respectivement "observed left value" et "observed right value". Il émet ov défini de la façon suivante :

$$ov = olv \oplus orv;$$

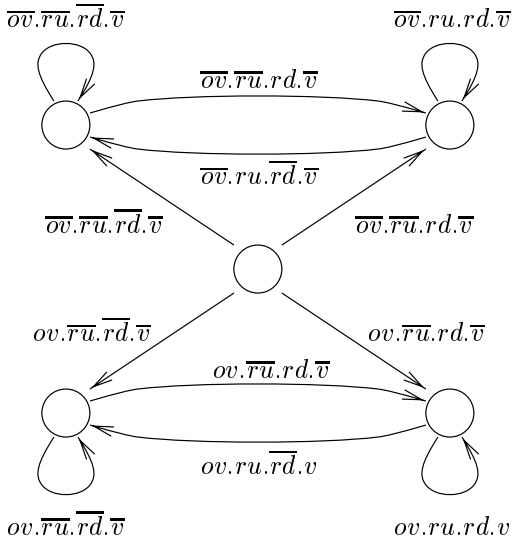


FIG. 7.2 – Un processus de l'arbre de parité

Le calcul en avant du plus petit point fixe ne converge pas. On a représenté respectivement figure 7.4 et 7.5 les pas 3 et 4 du calcul. La limite de cette suite est évidente à extrapoler. Il se crée deux branches infinies correspondant au temps maximum que met l'onde pour parvenir à une feuille et remonter à la racine: ces deux branches sont donc de taille 3 au pas 3Γet de taille 4 au pas 4. On peut supposer qu'il en est de même par la suite. On tente de construire à la main l'approximation du plus petit point fixe (voir figure 7.6). Notre outil (voir partie III) peut alors vérifier automatiquement que cet automate est bien un invariant du circuitΓi.e.Γque tout réseau de processus généré par la grammaire est plus petit (dans le sens de l'inclusion des traces) que cet automate. Comme cet automate satisfait la propriété demandée (i.e.Γque les variables v et ov sont égales lorsque ru est vrai)Γle réseau vérifie également cette propriété.

Nous verronsΓdans les chapitres suivantsΓque cette technique d'extrapolation de limites de suite fonctionnent rarement dans le cas des plus petits points fixes. En effetΓV^m représente l'ensemble des comportements possibles de tous les réseaux pouvant être générés par la grammaire: intuitivementΓet sauf cas particulierΓV^m ne peut pas être approché par un processus à nombre fini d'états. Dans la suiteΓnous allons étendre cette technique d'extrapolation au calcul du plus grand point fixeΓdont la taille dépend de celle de la propriété à démontrer (qui est en général assez petite).

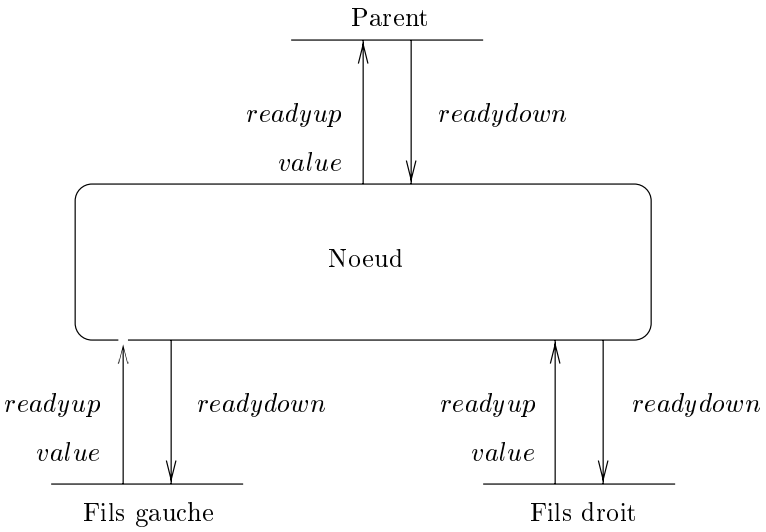


FIG. 7.3 – Un noeud de l'arbre de parité

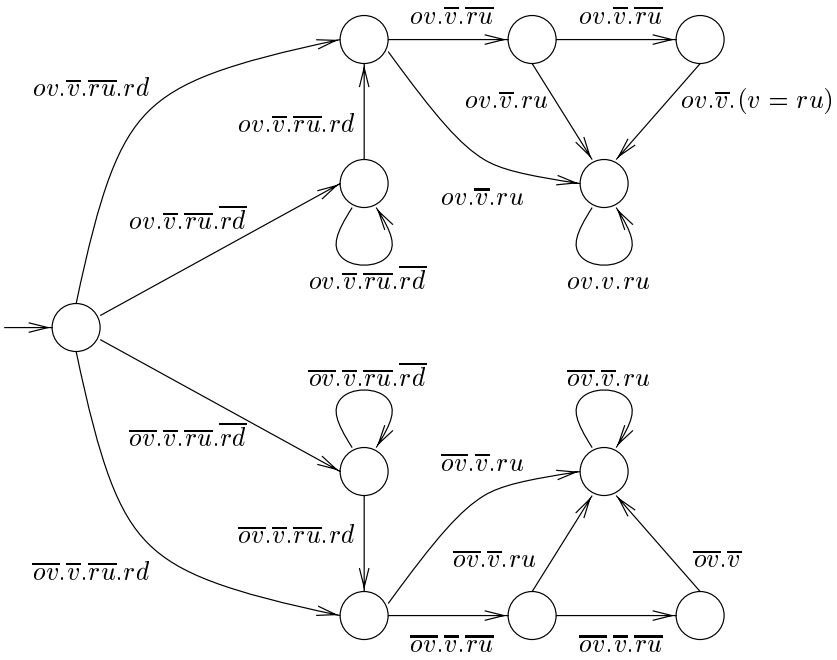


FIG. 7.4 – Arbre de parité: pas 3 du calcul en avant

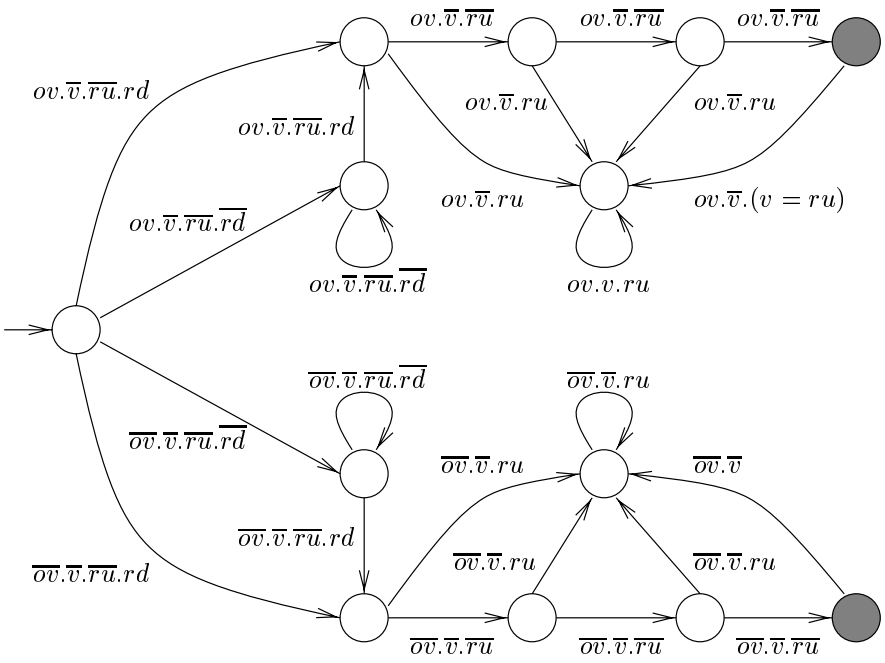


FIG. 7.5 – Arbre de parité: pas 4 du calcul en avant. Les états grisés correspondent à des états qui ont été ajoutés par rapport au pas 3.

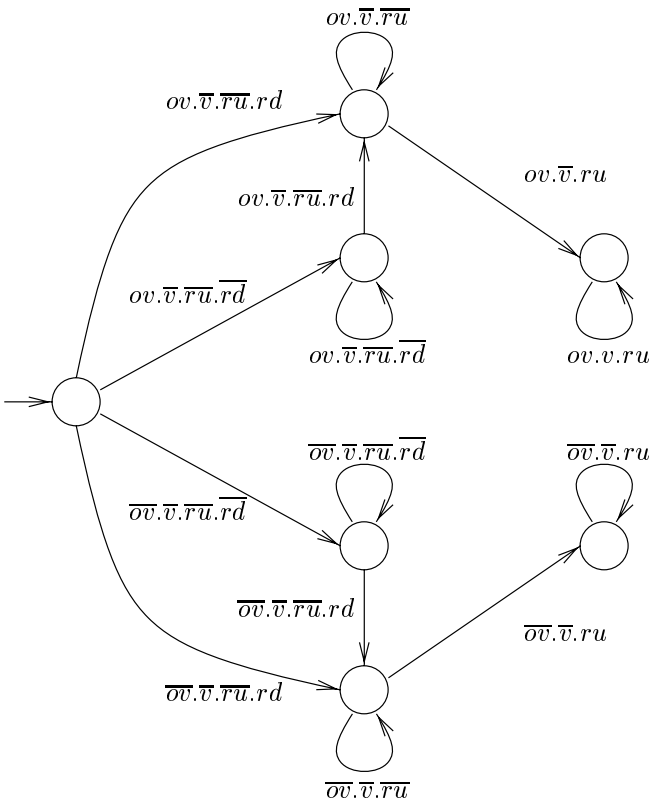


FIG. 7.6 – Arbre de parité : extrapolation de l'invariant par le calcul en avant

Chapitre 8

Invariants des réseaux linéaires

Dans ce chapitre nous nous intéressons au cas particulier des réseaux linéaires (voir section 6.1.2.2 page 61). Ce cas a été étudié dans [LHR96, LHR97].

8.1 Équations d'induction linéaires

Une grammaire linéaire peut être caractérisée par un ensemble fini de processus $\{P_1, \ldots, P_k\}$ et un ensemble fini d'opérateurs de composition binaire sur les processus $\{\|_1, \ldots, \|_k\}$. L'ensemble des processus générés par la grammaire est alors défini par

$$\forall i = 1 \ldots k, \ \left(\ P_i \in \mathcal{G} \ \right) \text{ et } \left(\ P \in \mathcal{G} \Rightarrow P \|_i P_i \in \mathcal{G} \ \right)$$

Montrer par induction que cette grammaire vérifie une propriété φ revient à déterminer un invariant I tel que

[SAT]	$I \models \varphi$	Satisfaction
[INIT]	$\forall i = 1 \ldots k, \ P_i \preceq I$	Initiation
[INDUC]	$\forall i = 1 \ldots k, \ I \|_i P_i \preceq I$	Induction

8.2 Plus petit et plus grand points fixes

Dans notre sémantique de traces, l'équation d'induction

$$I \|_i P_i \preceq I$$

peut être exprimée par

$$\left(\ \exists X', X'', T_I[X/X'] \otimes T_{\|} \otimes T_P[X/X''] \ \right) \subseteq T_I$$

Le problème du calcul de l'invariant peut donc se récrire

[SAT]	$T_I \subseteq T_\varphi$
[INIT]	$\forall i = 1 \ldots k, T_{P_i} \subseteq T_I$
[INDUC]	$\forall i = 1 \ldots k, T_{I \|_i P_i} \subseteq T_I$ i.e.
	$\left(\ \exists X', X'', T_I[X/X'] \otimes T_{\|} \otimes T_P[X/X''] \ \right) \subseteq T_I$

Lemme 8.1 :

[**INDUC**] peut être récrit de la manière suivante :

$$\forall i = 1 \ldots k,\ T_I \quad \subseteq \quad \Big(\quad \forall X, \forall X'',\ \big(\ \big(\ X \cup X' \cup X''\ \big)^{\infty} \setminus T_{\|_i}\ \big)\ \oplus \\ \big(\ X^{\infty} \setminus T_{P_i}\ \big)\,[X/X'']\ \oplus\ T_I \Big)[X'/X]$$

□

Preuve :

La preuve est fastidieuse mais très facile : il suffit de remplacer les opérateurs d'abstraction par leur définition.

$$\big(\ \exists X', X'', T_I[X/X'] \otimes T_{\|} \otimes T_P[X/X'']\ \big) \subseteq T_I$$
$$\Updownarrow$$
$$\forall x \in X^{\infty},\ x \in \big(\ \exists X', X'', T_I[X/X'] \otimes T_{\|} \otimes T_P[X/X'']\ \big) \Rightarrow x \in T_I$$
$$\Updownarrow$$
$$\forall x \in X^{\infty},\ x \notin \Big\{ x \in X^{\infty} \mid \exists x', x'' \in X^{\infty}, x' \in T_I \wedge (x \odot x' \odot x'') \in T_{\|} \wedge \\ x'' \in T_P \Big\} \vee x \in T_I$$
$$\Updownarrow$$
$$\forall x, x', x'' \in X^{\infty}, x' \notin T_I \vee (x \odot x' \odot x'') \notin T_{\|} \vee x'' \notin T_P \vee x \in T_I$$
$$\Updownarrow$$
$$\forall x' \in X^{\infty}, x' \notin T_I \vee \big(\ \forall x, x'' \in X^{\infty}, (x \odot x' \odot x'') \notin T_{\|} \vee x'' \notin T_P \vee x \in T_I\ \big)$$
$$\Updownarrow$$
$$\forall x' \in X^{\infty}, x' \notin T_I \vee \\ x' \in \big\{\ x' \in X^{\infty} \mid \forall x, x'' \in X^{\infty}, (x \odot x' \odot x'') \notin T_{\|} \vee x'' \notin T_P \vee x \in T_I\ \big\}$$
$$\Updownarrow$$
$$\forall x' \in X^{\infty}, x' \in T_I \Rightarrow x' \in \{x' \in X^{\infty} \mid \forall X, X'', \big(\ \big(\ X \cup X' \cup X''\ \big)^{\infty} \setminus T_{\|}\ \big) \oplus \\ \big(\ X^{\infty} \setminus T_P\ \big)\,[X/X''] \oplus T_I\}$$
$$\Updownarrow$$
$$T_I[X/X'] \subseteq \forall X, X'', \big(\ \big(\ X \cup X' \cup X''\ \big)^{\infty} \setminus T_{\|}\ \big) \oplus \big(\ X^{\infty} \setminus T_P\ \big)\,[X/X''] \oplus \\ T_I$$
$$\Updownarrow$$
$$T_I \subseteq \Big(\forall X, X'', \big(\ \big(\ X \cup X' \cup X''\ \big)^{\infty} \setminus T_{\|}\ \big) \oplus \\ \big(\ X^{\infty} \setminus T_P\ \big)\,[X/X''] \oplus T_I \Big)[X'/X]$$

□

Théorème 8.1 :

Il existe un plus petit langage de traces T_I^m satisfaisant [**INIT**] et [**INDUC**]Γet un plus grand T_I^M satisfaisant [**SAT**] et [**INDUC**]. T_I^m et T_I^M sont respectivement le plus petit et le plus grand point fixe des fonctions f et F définies par

$$f \quad = \quad \lambda T. \bigcup_{i=1}^{k} T_{P_i} \cup \big(\ \exists X', \exists X'',\ T_{\|_i} \otimes T[X/X'] \otimes T_{P_i}[X/X'']\ \big)$$

$$F \quad = \quad \lambda T. \varphi \cap \bigcap_{i=1}^{k} \Big(\forall X, \forall X'',\ \big(\ \big(\ X \cup X' \cup X''\ \big)^{\infty} \setminus T_{\|_i}\ \big) \oplus \\ \big(\ X^{\infty} \setminus T_{P_i}\ \big)\,[X/X''] \oplus T \Big)[X'/X]$$

□

Preuve :

L'existence de T_I^m découle du théorème 7.2Γpage 79. On rappelle la preuve.

[**INDUC**] se récrit

$$\forall i = 1 \ldots k,\ f_i(T_I) \ \subseteq \ T_I$$

La conjonction avec [**INIT**] donne

$$f(T_I) \ \stackrel{\Delta}{=} \ \bigcup_{i=1}^{k} (f_i(T_I) \cup T_{P_i}) \ \subseteq \ T_I$$

T_I est un post point fixe de la fonction croissante f. Il existe donc une plus petite solutionΓT_I^m.

De la même manièreΓ[**INDUC**] peut se récrire en

$$\forall i = 1 \ldots k,\ T_I \ \subseteq \ F_i(T_I) \quad \text{(d'après le lemme 8.1)}$$

Il existe donc un plus grand ensemble de traces T_I^M satisfaisant [**SAT**] et [**INDUC**]Γ qui est le plus grand point fixe de la fonction croissante

$$F \ = \ \lambda T.\varphi \cap \bigcap_{i=1}^{k} F_i(T)$$

En résuméΓ

$$T_I^m \ = \ \bigcup_{n \geq 0} f^{(n)}(\emptyset) \qquad \text{et} \qquad T_I^M \ = \ \bigcap_{n \geq 0} F^{(n)}(X^\infty)$$

avec

$$f \ = \ \lambda T. \bigcup_{i=1}^{k} T_{P_i} \cup \left(\ \exists X', \exists X'',\ T_{\|_i} \otimes T[X/X'] \otimes T_{P_i}[X/X''] \ \right)$$

$$F \ = \ \lambda T.\varphi \cap \bigcap_{i=1}^{k} \Big(\forall X, \forall X'',\ \left(\ \left(\ X \cup X' \cup X'' \ \right)^\infty \setminus T_{\|_i} \ \right) \oplus \left(\ X^\infty \setminus T_{P_i} \ \right)[X/X''] \oplus T \Big)[X'/X]$$

□

AinsiΓ notre problème de vérification est équivalent à montrer que soit $T_I^m \subseteq \varphi$Γ soit $\forall i = 1 \ldots k, T_{P_i} \subseteq T_I^M$.

Comme précédemmentΓles deux suites définissant T_I^m et T_I^M ne convergent en général pas en un temps fini : les limites sont des processus à nombre infini d'étatsΓet ne peuvent pas être aisément représentées par des automates à nombre fini d'états. Dans la suiteΓnous allons nous inspirer de la technique des élargissementsΓintroduite par Cousot et Cousot [CC77ΓCC92]Γafin d'approcher ces limites par des processus à nombre fini d'états.

T_I^m est l'ensemble de toutes les traces possibles de tous les réseaux générés par la grammaire : l'expérience montre qu'en généralΓle nombre d'états des automates reconnaissant les $f^{(n)}(\emptyset)$ augmente de façon exponentielle. Il semble à priori difficile d'extrapoler la limite d'une telle suite. Au contraireΓcomme le montre l'exemple de la section 10.1Γ page 109Γil arrive que le calcul du plus grand point fixe converge. GénéralementΓmême s'il ne converge pasΓle nombre d'états des automates reconnaissant $F^{(n)}(X^\infty)$ augmente de façon "modérée". C'est pourquoi nous nous intéresserons particulièrement aux approximations du plus grand point fixe.

8.3 Élargissement

8.3.1 Principe de l'élargissement

Afin d'approcher le plus grand point fixe T_I^M dans un treillis complet LΓla méthode d'extrapolation proposée par [CC77ΓCC92] consiste à définir un opérateur binaire ∇Γappelé élargissementΓ satisfaisant les deux propositions suivantes :

[**INCL**] : $\forall x, y \in L,\ x \nabla y \subseteq x \cap y$

[**CHAIN**] : pour toute suite décroissante $x_0 \supseteq x_1 \supseteq \ldots \supseteq x_n \supseteq \ldots$ dans LΓla suite de L $(y_0, y_1, \ldots, y_n, \ldots)$ définie par $y_0 = x_0$ et $y_{n+1} = y_n \nabla x_{n+1}$Γest strictement décroissante (i.e.ΓΓdevient constante à partir d'un nombre fini de termes).

AlorsΓpour toute fonction monotone $F : L \to L$Γla suite $y_0 = \top$ (la borne supérieure du treillis)Γ $y_{n+1} = y_n \nabla F(y_n)$ convergeΓen un nombre fini de pasΓvers une limite yΓqui est plus petite que le plus grand point fixe de F.

Dans cette approcheΓnous avons à définir un opérateur d'extrapolation sur les langages de traces. La conception d'un tel opérateur est une tâche expérimentaleΓnécessitant un compromis entre l'efficacité du calcul et la précision du résultat : suivant l'opérateur utiliséΓon peut obtenir soit une très longue suite convergeant vers une solution très proche du point fixeΓou inversementΓ une convergence rapide vers une solution grossière.

Nous avons décidé de définir cet opérateur d'élargissement sur des automates reconnaissant des langages de traces.

8.3.2 Quelques idées d'élargissements

8.3.2.1 Un premier opérateur

Soient $A_{T_1} = (Q_1, q_1^0, X, \{\alpha_1\}, \delta_1^Q, \delta_1^O)$ et $A_{T_2} = (Q_2, q_2^0, X, \{\alpha_2\}, \delta_2^Q, \delta_2^O)$ les automates minimaux déterministes reconnaissant les langages T_1 et T_2 sur XΓtels que $T_2 \subseteq T_1$. Nous avons à l'esprit que T_1 et T_2 sont deux pas consécutifs du calcul du plus grand point fixe T_I^M. L'automate $A_{T_1} \nabla A_{T_2}$Γreconnaissant l'élargissement $T_1 \nabla T_2$Γva être calculé en comparant les structures des deux automates A_{T_1} et A_{T_2}.

Soit $A_\times = (Q_1 \times Q_2, (q_1^0, q_2^0), X, \{\alpha_1, \alpha_2\}, \delta_\times^Q, \delta_\times^O)$ le produit synchrone des deux automates A_{T_1} et A_{T_2}. $A_{T_1} \nabla A_{T_2}$ va être construit à partir de $A_\times$. Soit

$$A_{T_1} \nabla A_{T_2} = (Q_1 \times Q_2, (q_1^0, q_2^0), X, \{\alpha_1, \alpha_2\}, \delta_\nabla^Q, \delta_\nabla^O)$$

On pose dans un premier temps $\delta_\nabla^Q = \delta_\times^Q$. Il reste à définir δ_∇^O.

Soit $q = (q_1, q_2)$ un état de $A_\times$ et soit $x \in 2^X$ un événement sur X.

- Si $\delta_1^O(q_1, x) = \{\alpha_1\}$ et $\delta_2^O(q_2, x) = \{\alpha_2\}$Γi.e.Γl'événement x est refusé par les deux automatesΓ A_∇ le refuse également : $\delta_\nabla^O(q, x) = \{\alpha\}$.

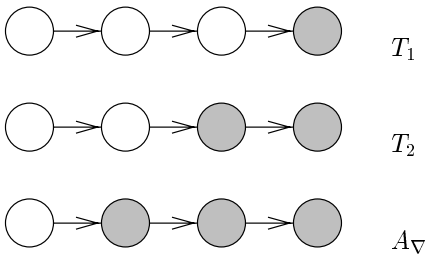


FIG. 8.1 – Principe de l'élargissement

- Si $\delta_1^O(q_1, x) = \emptyset$ et $\delta_2^O(q_2, x) = \emptyset$Γi.e.Γl'événement x est accepté par les deux automatesΓA_∇ l'accepte également : $\delta_\nabla^O(q, x) = \emptyset$.

- Si $\delta_1^O(q_1, x) = \emptyset$ et $\delta_2^O(q_2, x) = \{\alpha_2\}$Γi.e.Γl'événement x a été interdit par le dernier pas de calculΓl'état q est alors dit semi-accepteur. L'ensemble des états semi-accepteurs reconnaît le langage $T_1 \setminus T_2$.

Définition 8.1 :

On appelle ensemble des états semi-accepteursΓnoté DΓl'ensemble des états de $A_\times$ reconnaissant le langage $T_1 \setminus T_2$.

$$D \;=\; \{\; (q_1, q_2) \mid \exists x \in 2^X,\; \delta_1^O(q_1, x) = \emptyset \text{ et } \delta_2^O(q_2, x) = \{\alpha_2\} \;\}$$

□

L'opérateur proposé consiste à extrapoler un pas de calcul en interdisant les états pouvant conduire à un état semi-accepteur. On va donc définir δ_∇^O par

$$\delta_\nabla^O\left(\; (q_1, q_2), x \;\right) \;=\; \begin{cases} \{\alpha\} & \text{si } \delta_1^O(q_1, x) = \{\alpha_1\} \text{ ou } \delta_2^O(q_2, x) = \{\alpha_2\} \\ & \text{ou } \exists x' \in 2^X,\; \delta_2^O\left(\; \delta_2^Q(q_2, x), x' \;\right) = \{\alpha_2\} \\ \emptyset & \text{sinon} \end{cases}$$

(voir figure 8.1).

8.3.2.2 Projections

Afin d'obtenir des opérateurs plus puissantsΓon définit les projections de $A_\times$ sur les ensembles d'états Q_1 et Q_2.

Définition 8.2 :

On définit ainsi les opérateurs d'élargissement universel et existentiel respectivement notés $T_1 \nabla^\forall T_2 = (Q_1, q_1^0, X, \{\alpha_1\}, \delta_1^Q, \delta_\nabla^O)$ et $T_1 \nabla^\exists T_2 = (Q_2, q_2^0, X, \{\alpha_2\}, \delta_2^Q, \delta_\exists^O)$Γoù

$$\delta_\nabla^O(q, x) \;=\; \begin{cases} \emptyset & \text{si } \left(\; \delta_1^O(q, x) = \emptyset \;\right) \text{ et } \left(\; \forall q',\; \delta^O\left(\; (q, q'), x \;\right) = \emptyset \;\right) \\ \{\alpha\} & \text{sinon} \end{cases}$$

$$\delta_\exists^O(q', x) \;=\; \begin{cases} \emptyset & \text{si } \left(\; \delta_2^O(q', x) = \emptyset \;\right) \text{ et } \left(\; \exists q,\; \delta^O\left(\; (q, q'), x \;\right) = \emptyset \;\right) \\ \{\alpha\} & \text{sinon} \end{cases}$$

□

Justification des quantificateurs universel et existentiel : L'utilisation du quantificateur existentiel dans la projection sur Q_1 n'assurerait pas la propriété $T_1 \nabla T_2 \subseteq T_2$. Celle du quantificateur universel dans la projection sur Q_2 conduirait trop souvent au langage nul.

Propriété 8.1 :

1. $T_1 \nabla^\forall T_2 \subseteq T_1 \nabla^\exists T_2 \subseteq T_2 \subseteq T_1$.
2. La suite définie par

$$\begin{cases} T^0 &= X^\infty \\ T^1 &= F(T^0) \\ T_{i+1} &= T^i \nabla^\forall F(T^i) \end{cases}$$

converge en un temps fini. En effet l'automate T^{i+1} a au plus autant d'états que T^i.

3. La suite définie par

$$\begin{cases} T^0 &= X^\infty \\ T^1 &= F(T^0) \\ T^{i+1} &= T^i \nabla^\exists F(T^i) \end{cases}$$

converge en un temps éventuellement infini.

Cet opérateur n'est donc pas un élargissement au sens que lui ont donné Cousot et Cousot [CC77 CC92]. Il s'agit plutôt d'un "accélérateur" qui accélère la convergence sans la forcer à coup sûr.

□

Ces deux opérateurs ont été implémentés dans un outil (voir partie III). Les résultats expérimentaux n'ont pas été concluants : dans la plupart des exemples du chapitre 10 page 109 ces opérateurs forcent la convergence de la suite d'invariants vers le langage nul. Dans la section suivante nous proposons de raffiner ces opérateurs de façon à ce qu'ils prennent en compte la structure des automates reconnaissant les langages à extrapoler.

8.3.3 Un premier raffinement de l'élargissement

En fait certains comportements passant par des états semi-accepteurs (i.e. des états de D) doivent être interdits mais pas tous ces comportements. L'expérience a montré que la convergence en un temps infini pouvait être due au fait que des motifs "réguliers" étaient répétés de plus en plus de fois afin de produire des boucles infinies dans le langage limite. Par exemple la suite (T_i) définie pour $i \geq 1$ par

$$T_i \;=\; \{\; b^n + b^i.(a+b)^* \mid 0 \leq n < i \;\}$$

est infinie mais converge vers b^*.

Nous avons donc eu l'idée de créer des boucles en rebranchant de façon non déterministe des transitions $(q_0, q_0') \xrightarrow{x} (q_1, q_1')$ atteignant D vers d'autres états $(q_2, q_2') \notin D$.

Afin d'assurer l'inclusion des traces **[INCL]** le langage reconnu par (q_2, q_2') doit être inclus dans celui reconnu par (q_1, q_1'). Pour créer des boucles de façon "logique" les nouveaux états cible (q_2, q_2') sont cherchés parmi (q_0, q_0') et ses prédécesseurs qui satisfont cette inclusion. Ils sont recherchés jusqu'à une profondeur d_1 qui est un paramètre de l'opérateur.

Malheureusement un tel opérateur ne satisfait plus la propriété **[CHAIN]** : le nombre d'états du nouvel automate décroît mais comme cet automate n'est pas déterministe le nombre d'états de sa version déterministe peut augmenter. Nous obtenons donc de nouveau un accélérateur qui n'assure pas la terminaison de notre algorithme. Par contre si l'algorithme termine nous sommes

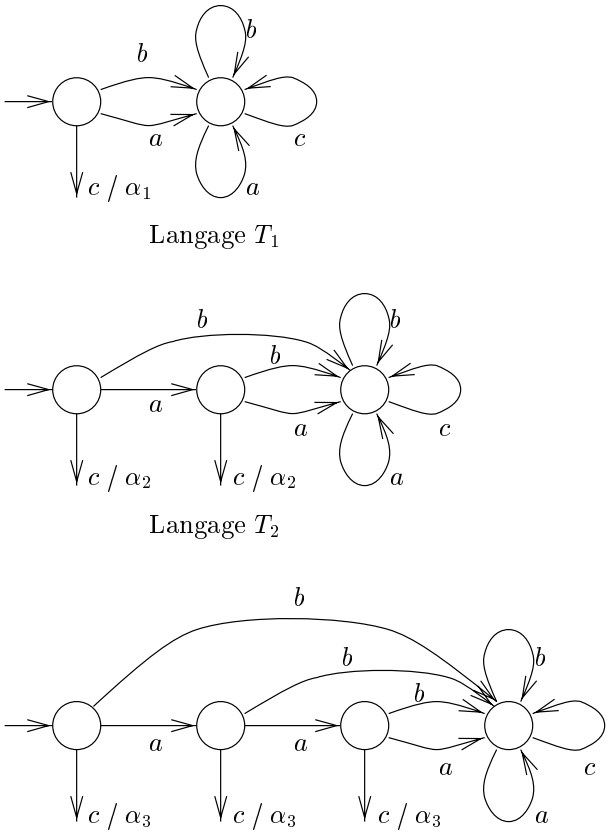


FIG. 8.2 – Les premiers termes d'une séquence infinie de langages

sûr d'obtenir une approximation inférieure du point fixe. Le problème de la non terminaison est discuté section 8.3.5.

Exemple 8.1 :

Soient T_1 et T_2 deux langages définis par

$$\begin{array}{lcll} T_1 & = & (b+a).(a+b+c)^* & \text{et} \\ T_2 & = & (b+a.(b+a)).(a+b+c)^* & \end{array}$$

IntuitivementΓon peut s'attendre à ce que le prochain pas de calcul produise le langage

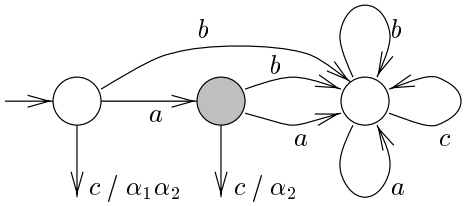
$$T_3 \;=\; (b+a.(b+a.(b+a))).(a+b+c)^*$$

Les automates reconnaissant $T_1 \Gamma T_2$ et T_3 sont présentés figure 8.2. La figure 8.3 représente le produit $T_1 \times T_2$ (où l'état grisé est le seul état semi-accepteurΓi.e.Γde D) ainsi que l'élargissement obtenu par rebranchement d'une transition avec le paramètre $d_1 = 1$ (i.e.Γle nouvel état cible ne peut être que le source). On obtient ainsi l'extrapolation

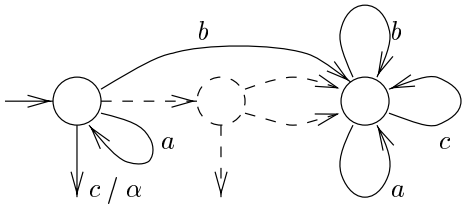
$$T_1 \nabla T_2 \;=\; a^*.b.(a+b+c)^*$$

Il est à remarquer que si le rebranchement n'est pas effectué (i.e.Γavec $d = 0$)Γnous obtenons

$$T_1 \nabla T_2 \;=\; b.(a+b+c)^*$$



(a) $T_1 \times T_2$



(b) $T_1 \nabla T_2$

FIG. 8.3 – Élargissement sur les automates

Cette extrapolation est probablement trop grossière car elle ne traduit pas le fait qu'un nombre arbitraire d'événements a peuvent survenir avant un événement b. □

Cet opérateur a été testé sur les exemples du chapitre 10Γpage 109Γet a permis d'extrapoler un certain nombre d'invariants. Dans la section suivanteΓnous proposons une amélioration de cet extrapolation.

8.3.4 Un deuxième raffinement de l'élargissement

Considérons une suite d'invariants dont deux pas consécutifs ont la forme suivante :

$$T_1 = b + c.c + a.(b + c.c + a.(b + (a + c).(a + b + c)^*))$$
$$T_2 = b + c.c + a.(b + c.c + a.(b + c.c + a.(b + (a + c).(a + b + c)^*)))$$

(voir figure 8.4). IntuitivementΓle pas de calcul suivant doit produire le langage

$$T_3 = b + c.c + a.(b + c.c + a.(b + c.c + a.(b + c.c + a.(b + (a + c).(a + b + c)^*))))$$

La figure 8.5 présente l'application de l'opérateur d'extrapolation vu section précédente. L'action de cet opérateur est visiblement incorrecte. Au lieu de supprimer l'état reconnaissant le chaos $(a + b + c)^*$Γil tente de supprimer l'état reconnaissant le suffixe $c.c$. Puisqu'aucun état parvenant à l'état grisé ne le simuleΓaucun rebranchement n'est effectué. Pour résoudre ce problèmeΓnous proposons non pas d'interdire les états de DΓmais les états parvenant à D sur une certaine profondeur (qui peut être un paramètre d_2). La figure 8.6 présente l'application de cette idée avec $d_2 = 1$.

On obtient ainsi un opérateur paramétré par deux entiers d_1 et d_2. L'expérience a montré qu'un bon choix pour ces valeurs pouvait être $d_1 = 3$ et $d_2 = 1$. Bien entenduΓen cas d'échecΓ l'utilisateur peut jouer sur ces deux paramètres pour améliorer l'élargissement.

8.3.5 Un algorithme de semi-décision

En pratique, la non convergence de notre suite d'élargissement n'est pas gênante. En effet, nous ne calculons pas simplement la limite de la suite T_i définie par

$$\begin{aligned} T_0 &= X^\infty \\ T_{i+1} &= T_i \nabla F(T_i) \end{aligned}$$

car elle est en général trop grossière. En fait, nous pouvons arbitrairement améliorer la solution en retardant l'application de l'extrapolation. Pour tout $k \in \mathbb{N}$, définissons $T^{(k)}$ comme étant la limite de la suite

$$\begin{aligned} T_0 &= X^\infty \\ T_{i+1} &= \begin{cases} F(T_i) & \text{si } i < k \\ T_i \nabla F(T_i) & \text{si } i \geq k \end{cases} \end{aligned}$$

Chaque $T^{(k)}$ est une approximation inférieure du point fixe (l'approximation la plus grossière est $T^{(0)}$, et plus k est grand, plus l'approximation $T^{(k)}$ est précise). La méthode consiste à calculer $T^{(k)}$ en laissant croître le paramètre k tant que l'invariant $T^{(k)}$ est trop fort (i.e. ne satisfait pas [**INIT**]). Ces itérations sur k peuvent ne pas terminer, et pour chaque k le calcul de $T^{(k)}$ peut ne pas terminer (puisque notre opérateur ne satisfait pas la propriété [**CHAIN**]). En théorie, il pourrait arriver que le calcul de $T^{(k)}$ soit infini, alors que celui de $T^{(k+1)}$ converge vers un invariant correct, i.e. que la convergence dépende de l'ordre dans lequel on incrémente i et k.

D'un point de vue théorique, si nous souhaitons une procédure de semi-décision (dans le sens que s'il existe k tel que l'approximation $T^{(k)}$ soit calculable en un temps fini et soit correcte, il sera forcément atteint par notre algorithme), l'algorithme doit effectuer une exploration en largeur du graphe des approximations, i.e. laisser augmenter i et k en même temps.

D'un point de vue pratique, la taille des automates calculés augmente rapidement, et tous les calculs, soit convergent rapidement, soit saturent la mémoire.

8.4 Combiner le calcul en avant et en arrière

Dans cette section, nous proposons de combiner le calcul en avant (vu dans le chapitre 7) avec le calcul en arrière (vu précédemment dans ce chapitre). Considérons l'inéquation d'induction [**INDUC**]

$$\begin{aligned} T_I \subseteq \Big(& \forall X, \forall X'', \ \big(\ \big(\ X \cup X' \cup X'' \ \big)^\infty \setminus \|_i \ \big) \ \oplus \\ & \big(\ X^\infty \setminus T_{P_i} \ \big) [X/X''] \ \oplus \ T_I \Big) [X'/X] \end{aligned}$$

Celle-ci dépend du processus terminal P_i, et peut donc être récrite en explicitant ce paramètre.

$$T_I \subseteq F(T_{P_i}, T_I)$$

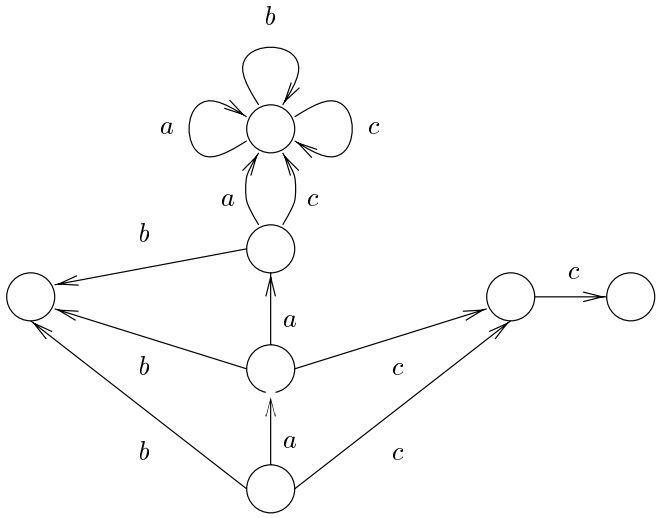
On remarque que la fonction F est croissante par rapport à T_I et décroissante par rapport à T_{P_i}. Pour tout processus P_i' tel que $T_{P_i} \subseteq T_{P_i'}$, $F(T_{P_i'}, T_I) \subseteq F(T_{P_i}, T_I)$.

En cas de non convergence de notre calcul de point fixe, il est donc possible de tenter le même calcul avec des processus P_i' plus grands que les P_i. Dans le chapitre 11, page 125, nous présenterons des exemples utilisant cette technique. Nous avons envisagé d'effectuer l'extrapolation du processus P_i de 3 manières.

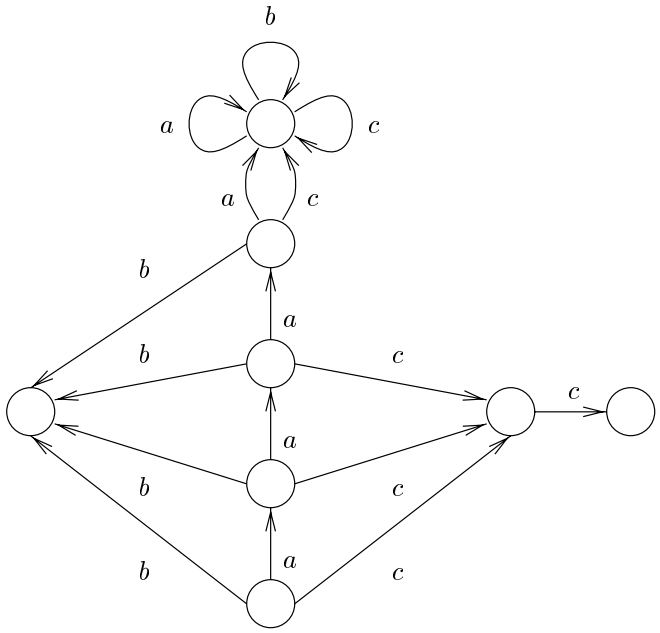
1. En effaçant des signaux sans rapport avec la propriété à démontrer. Le concepteur d'un algorithme a introduit des communications pour des raisons précises. Par exemple, pour vérifier une propriété d'exclusion mutuelle, il n'est pas nécessaire de considérer les signaux introduits pour gérer la priorité entre les processus.

2. En comprenant l'algorithmeΓet en modifiant certains paramètres. Il est possible de modifier un algorithme qui vérifie toujours une propriété donnéeΓet tel que les processus considérés soient plus grands que les processus du réseau initial. Cette technique dépend bien entendu uniquement de "l'intelligence" de l'utilisateur.

3. En utilisant des outils d'extrapolation en avant. Il est possible d'extrapoler des processus contenant des données infinies et d'effectuer le calcul de point fixe sur cette abstraction (voir par exemple [Saï96ΓGS96]).



Langage T_1



Langage T_2

FIG. 8.4 – Un exemple de suite de langages

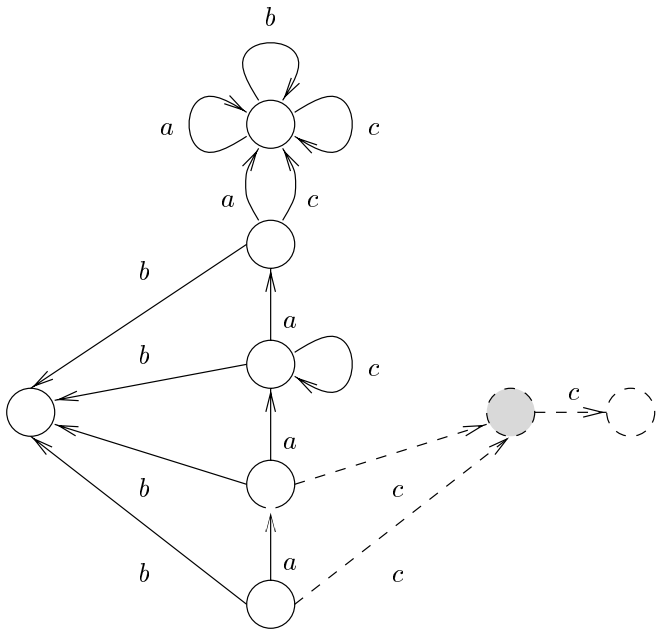


FIG. 8.5 – Application de la première version de l'élargissement

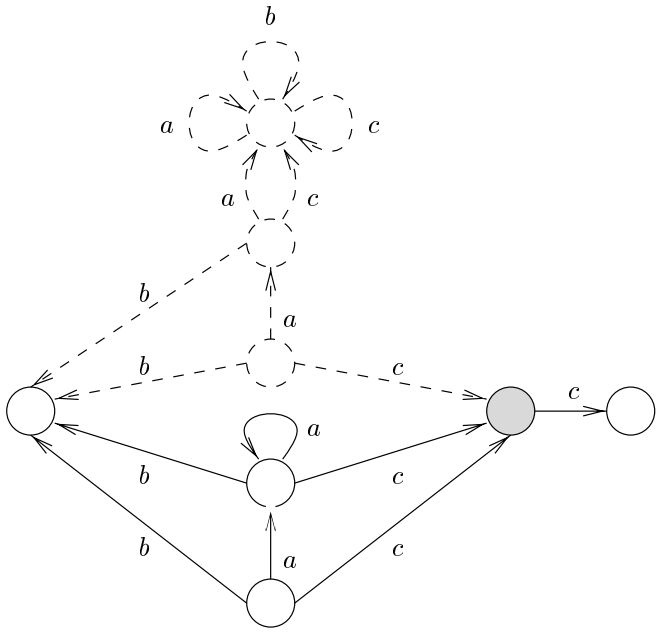


FIG. 8.6 – Application de la deuxième version de l'élargissement

Chapitre 9

Invariants des réseaux arborescents

Dans ce chapitre une deuxième forme particulière de réseau est étudiée celle définie par les grammaires binaires arborescentes. Les techniques développées dans ce chapitre peuvent être aisément étendues à des grammaires de réseau quelconques. Ce cas a été étudié dans [Les97b].

9.1 Introduction

On considère des grammaires arborescentes binaires.

Définition 9.1 :

Une grammaire $\mathcal{G} =< T, N, \mathcal{P}, \mathcal{S} >$ est dite arborescente binaire si elle est de la forme

- $T = \{P\}$ où P est un processus feuille ("leaf").
- $N = \{N\}$ où N est un noeud de l'arbre ("node").
- $\mathcal{P} = \{ \mathcal{S} \to N,$
  $N \to N \| N,$
  $N \to P \ \}$

□

Selon le même principe que la section 7.1.3.2 page 75 nous pouvons montrer qu'une telle grammaire de réseau satisfait une propriété φ en trouvant un invariant I tel que

[SAT]	I	$\models$	φ	Satisfaction
[INIT]	P	$\preceq$	I	Initiation
[INDUC]	$I \| I$	$\preceq$	I	Induction

ce qui se traduit dans notre sémantique de traces par

[SAT]	$T_I \subseteq T_\varphi$
[INIT]	$T_P \subseteq T_I$
[INDUC]	$T_{I\|I} \subseteq T_I$ ou de façon équivalente
	$\left(\exists X', X'', T_I[X/X'] \cap T_\| \cap T_I[X/X''] \right) \subseteq T_I$

Notons **[INDUCTION]**=**[SAT]**∧**[INIT]**∧**[INDUC]** l'ensemble de ces inéquations.

Corollaire 9.1 :

(du théorème 7.2 page 79). Il existe un plus petit langage de traces T_I^m satisfaisant **[INIT]** et **[INDUC]**. □

Preuve :

On rappelle que T_I^m est le plus petit point fixe de la fonction croissante f définie par

$$f(T) \;=\; T \cup P \cup \left(\; \exists X', X'', T_I[X/X'] \cap T_{\|} \cap T_I[X/X''] \; \right)$$

T_I^m est défini par

$$T_I^m \;=\; \bigcup_{n \geq 0} f^{(n)}(\emptyset)$$

□

Le calcul du plus petit point fixe par itération est fortement divergent (voir les exemples du chapitre 10). Comme dans le chapitre précédent, nous préférons donc calculer un plus grand point fixe. La technique utilisée pour les réseaux linéaires ne peut hélas pas s'appliquer simplement aux réseaux arborescents.

Problème 9.1 :

Existe-t-il un plus grand langage de traces satisfaisant [**SAT**] et [**INDUC**] ? □

La question reste ouverte. En effet, l'existence d'un plus grand point fixe ne peut pas être démontré à partir de l'inéquation [**INDUC**]. Pour prouver la non existence, il faudrait exhiber un contre-exemple, i.e. trouver deux invariants T_I^1 et T_I^2 d'un réseau satisfaisant la propriété, et tels que leur union $T_I^1 \cup T_I^2$ ne satisfasse pas la propriété.

9.2 Expression du plus grand point fixe

9.2.1 Difficulté du problème

Essayons de comprendre pourquoi la méthode de la section 8.2, page 87 ne marche pas ici. Nous avons l'inéquation de récurrence

$$I \| I \preceq I \qquad \text{qui se traduit par} \qquad \left(\; \exists X', X'', T_I[X/X'] \otimes T_{\|} \otimes T_I[X/X''] \; \right) \subseteq T_I$$

Cette inéquation signifie que si le fils gauche et le fils droit satisfont la propriété de récurrence I, alors le nouveau noeud formé par la composition du fils gauche et du fils droit satisfait la propriété I. En utilisant une technique similaire à celle de la section 8.2, on montre aisément que

$$T_I \subseteq \left(\; \forall X, \forall X'', \left(\; \left(\; X \cup X' \cup X'' \; \right)^{\infty} \setminus T_{\|} \; \right) \oplus \left(\; X^{\infty} \setminus T_I \; \right) [X/X''] \oplus T_I \; \right) [X'/X]$$

et

$$T_I \subseteq \left(\; \forall X, \forall X', \left(\; \left(\; X \cup X' \cup X'' \; \right)^{\infty} \setminus T_{\|} \; \right) \oplus \left(\; X^{\infty} \setminus T_I \; \right) [X/X'] \oplus T_I \; \right) [X''/X]$$

i.e.,

$$T_I \subseteq F(T_I)$$

où la fonction F n'est plus monotone. Les théorèmes de points fixes ne peuvent alors pas s'appliquer : en particulier, l'itération de la fonction F ne converge plus forcément vers un plus grand point fixe.

Intuition : La non convergence vers un plus grand point fixe peut se comprendre intuitivement de la manière suivante (voir figure 9.1) : la récurrence consiste à déterminer une condition sur les fils d'un noeud pour que ce noeud satisfasse une propriété donnée. Pour ce faire, il est possible de contraindre fortement le comportement du fils gauche, et moins celui du fils droit, ou au contraire contraindre fortement le comportement du fils droit et moins celui du fils gauche. L'idéal serait de déduire une seule propriété pour les deux fils. En pratique, cela semble impossible, car le problème n'est pas parfaitement symétrique. Ainsi, dans l'exemple de la section 10.4, page 117, le père passe le jeton à son fils gauche, avant de le passer à son fils droit. Le reste de ce chapitre va exploiter l'idée qu'il est nécessaire de distinguer les propriétés du fils gauche et du fils droit.

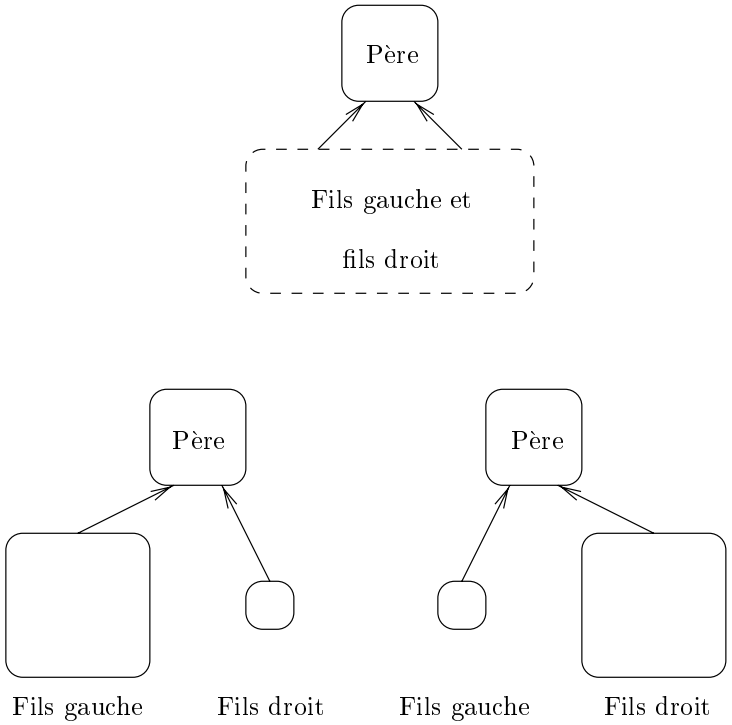


FIG. 9.1 – Intuition relative au calcul d'un invariant d'un réseau arborescent. Pour que le père satisfasse une propriété donnéeΓil est possible de contraindre de façon asymétrique les fils gauche et droit

9.2.2 Induction à l'aide de 2 invariants

En faitΓil est suffisant de trouver 2 invariants G (pour fils gauche) et D (pour fils droit) tels que si le fils gauche satisfait G et le fils droit DΓalors le nouveau noeud satisfait I. Il est bien entendu nécessaire que $I \preceq G$ et $I \preceq D$.

On peut donc récrire l'inéquation d'induction [**INDUC**] sous la forme

$$G \| D \ \preceq \ I \ \text{ avec } \ I \ \preceq \ G \ \text{ et } \ I \ \preceq \ D$$

ou plus simplement

$$G \| D \ \preceq \ G \ \text{ et } \ G \| D \ \preceq \ D$$

Il est donc suffisant de trouver deux invariants G et DΓtels que

[**SAT**]	G	$\models$	φ	et
	D	$\models$	φ	
[**INIT**]	P	$\preceq$	G	et
	P	$\preceq$	D	
[**INDUC**]	$G \| D$	$\preceq$	G	et
	$G \| D$	$\preceq$	D	

Dans notre sémantique de tracesΓces inéquations se récrivent de la façon suivante :

$$\begin{array}{lrcll}
\textbf{[SAT]} & & T_G & \subseteq & T_\varphi \quad \text{et} \\
 & & T_D & \subseteq & T_\varphi \\
\textbf{[INIT]} & & T_P & \subseteq & T_G \quad \text{et} \\
 & & T_P & \subseteq & T_D \\
\textbf{[INDUC]} & \exists X', \exists X'',\ T_G[X/X'] \otimes T_{\|} \otimes T_D[X/X''] & \subseteq & T_G & \text{et} \\
 & \exists X', \exists X'',\ T_G[X/X'] \otimes T_{\|} \otimes T_D[X/X''] & \subseteq & T_D &
\end{array}$$

9.2.3 Vecteur d'invariants

L'utilisation de deux invariants au lieu d'un découle intrinsèquement du problème arborescent : il est nécessaire de distinguer le fils gauche du fils droitΓmême s'ils ont la même définition. PourtantΓ un calcul de plus grand point fixe s'effectue beaucoup plus simplement à l'aide d'un seul invariant.

Afin de ne considérer qu'un invariantΓnous proposons d'utiliser un vecteur d'invariants (voir section 7.3Γpage 79)Γque nous codons de la manière suivante : supposons que le problème soit résoluΓi.e.Γque nous connaissions les deux invariants G et D satisfaisant les inéquations précédentesΓ et définissons $V \subseteq \left(X' \cup X'' \right)^\infty$ comme étant l'union non synchronisée de G et D :

$$V = T_G[X/X'] \odot T_D[X/X'']$$

Il est à noterΓque G et D peuvent alors aisément être calculés à partir de V par simple projection

$$\begin{array}{rcl}
T_G & = & (\exists X'', V)[X'/X] \\
T_D & = & (\exists X', V)[X''/X]
\end{array}$$

L'idée va maintenant consister à récrire nos inéquations en fonction de V.

[SAT] : $V \subseteq T_\varphi[X/X'] \odot T_\varphi[X/X'']$

[INIT] : Comme $T_G[X/X'] \odot T_D[X/X''] \subseteq \left(X' \cup X'' \right)^\infty$ qui est orthogonal à X^∞Γl'inéquation $T_P \subseteq T_G$ est équivalente à

$$T_P \odot T_G[X/X'] \odot T_D[X/X''] \subseteq T_G \odot T_G[X/X'] \odot T_D[X/X'']$$

ce qui se récrit en fonction de V en $T_P \odot V \subseteq V[X'/X] \otimes V$.

Utilisons la propriété suivante. Si AΓB et C sont des ensemblesΓ

$$A \cap B \subseteq C \iff A \cap B \subseteq C \cap B$$

AlorsΓ$T_P \subseteq T_G$ est équivalent à $T_P \odot V \subseteq V[X'/X]$

De mêmeΓ$T_P \subseteq T_D$ est équivalent à $T_P \odot V \subseteq V[X''/X]$

[INDUC] : Utilisons de nouveau la propriété $A \cap B \subseteq C \iff A \cap B \subseteq C \cap B$.

[INDUC] se récrit

$$\begin{array}{rcll}
T_G[X/X'] \otimes T_{\|} \otimes T_D[X/X''] & \subseteq & T_G \odot T_D[X/X''] & \text{et} \\
T_G[X/X'] \otimes T_{\|} \otimes T_D[X/X''] & \subseteq & T_G[X/X'] \odot T_D &
\end{array}$$

soit

$$V \otimes T_{\|} \subseteq V[X'/X] \otimes V[X''/X]$$

En résuméΓsi V s'écrit $V = T_G[X/X'] \odot T_D[X/X'']$Γles inéquations **[SAT]**Γ**[INIT]** et **[INDUC]** sont respectivement équivalentes aux inéquations **[SAT']**Γ**[INIT']** et **[INDUC']** définies par

$$\begin{array}{lrcl}
\textbf{[SAT']} & V & \subseteq & T_\varphi[X/X'] \odot T_\varphi[X/X''] \\
\textbf{[INIT']} & T_P \odot V & \subseteq & V[X'/X] \otimes V[X''/X] \\
\textbf{[INDUC']} & V \otimes T_{\|} & \subseteq & V[X'/X] \otimes V[X''/X]
\end{array}$$

Notons **[INDUCTION']**=**[SAT']**∧**[INIT']**∧**[INDUC']** l'ensemble de ces inéquations.

Théorème 9.1 :

Il existe un langage maximal V^M satisfaisant **[INDUCTION']**. V^M est le plus grand point fixe de la fonction F définie par

$$F = \lambda V.\ \forall X,\ \Big(\big(T_\varphi[X/X'] \odot T_\varphi[X/X''] \big) \otimes \big(\big(X \cup X' \cup X'' \big)^\infty \setminus \big(T_\| \oplus T_P \big) \oplus \big(V[X'/X] \otimes V[X''/X] \big) \big) \Big)$$

□

Preuve :

La preuve est similaire à celle du théorème 8.1. Les inéquations **[INDUCTION']** peuvent être reformulées de la manière suivante:

$$\begin{array}{lll}
\textbf{[SAT']} & V \subseteq & T_\varphi[X/X'] \odot T_\varphi[X/X''] \\
\textbf{[INIT']} & V \subseteq & \forall X, \Big(\big(X^\infty \setminus T_P \big) \oplus \big(V[X'/X] \otimes V[X''/X] \big) \Big) \\
\textbf{[INDUC']} & V \subseteq & \forall X, \Big(\big(\big(X' \cup X'' \big)^\infty \setminus T_\| \big) \oplus \big(V[X'/X] \otimes V[X''/X] \big) \Big)
\end{array}$$

La conjonction de ces trois inéquations donne

$$V \subseteq \forall X,\ \Big(\big(T_\varphi[X/X'] \odot T_\varphi[X/X''] \big) \otimes \big(\big(X \cup X' \cup X'' \big)^\infty \setminus \big(T_\| \oplus T_P \big) \oplus \big(V[X'/X] \otimes V[X''/X] \big) \big) \Big)$$

i.e.Γ$V \subseteq F(V)$Γoù

$$F = \lambda V.\ \forall X,\ \Big(\big(T_\varphi[X/X'] \odot T_\varphi[X/X''] \big) \otimes \big(\big(X \cup X' \cup X'' \big)^\infty \setminus \big(T_\| \oplus T_P \big) \oplus \big(V[X'/X] \otimes V[X''/X] \big) \big) \Big)$$

La fonction F est croissante dans le treillis des langages sur $\big(X' \cup X'' \big)^\infty$. Elle admet donc un plus grand point fixeΓqui est V^M. □

Comme dans le chapitre 8Γle calcul de V^M peut ne pas finirΓet V^M peut être un processus à nombre infini d'états. Les techniques d'élargissement de la section 8.3Γpage 90 peuvent alors être utilisées pour approcher V^M par un post point fixe de la fonction F.

La propriété suivante est triviale.

Propriété 9.1 :

Si V^M est videΓle réseau arborescent ne satisfait pas la propriété φ. □

Preuve :

D'après le théorème 7.2Γpage 79Γil existe un plus petit langage de traces T_I^m satisfaisant [**INIT**] et [**INDUC**]. En raisonnant par l'absurdeΓsupposons que le réseau arborescent satisfait la propriété φ. AlorsΓcomme T_I^m représente l'ensemble des traces du réseauΓ T_I^m satisfait la propriété φ et donc l'inéquation [**SAT**]. Posons $V = T_I^m[X/X'] \odot T_I^m[X/X'']$. Alors V est non vide et d'après ce qui précèdeΓsatisfait les inéquations [**INDUCTION'**]Γce qui contredit l'hypothèse V^M vide. Donc le réseau arborescent ne satisfait pas la propriété φ. □

InversementΓsi V^M est non videΓla propriété φ est-elle forcément satisfaite par le réseau arborescentΓi.e.Γa-t-on l'équivalence

$$V^M \neq \emptyset \iff \mathcal{G} \models \varphi \ ?$$

Il nous est possible de répondre à cette question siΓétant donné V^MΓnous pouvons calculer un invariant I du réseau. Ceci n'est à priori pas toujours possibleΓcar les inéquations [**INDUCTION**] ne sont équivalentes à [**INDUCTION'**] que si V^M peut s'écrire $V^M = T_G[X/X'] \odot T_D[X/X'']$ (ce qui n'estΓen généralΓpas le cas).

En pratiqueΓnous ne sommes capables de calculer qu'une approximation inférieure V de V^M. Si V est videΓnous ne pouvons évidemment rien conclure. Dans le cas contraireΓil est nécessaire d'exhiber les deux invariants G et D pour prouver que le réseau satisfait la propriété.

9.3 Calcul des invariants G et D

Dans cette sectionΓnous allons supposer que nous avons réussi à calculer V^M de façon exacte (i.e.Γsans extrapolation). Dans le cas contraireΓles algorithmes suivants peuvent s'appliquer à une approximation inférieure de V^MΓmais les heuristiques proposées risquent d'être moins efficaces.

9.3.1 Cas idéal

Soit V^M le plus grand point fixe des inéquations [**INDUCTION'**]. Si T_G et T_D sont deux langages satisfaisant

$$T_G[X/X'] \odot T_D[X/X''] \ = \ V^M$$

ces deux langages satisfont les inéquations [**INDUCTION**]. Dans ce casΓT_G et T_D peuvent être calculés par

$$T_G \ = \ \left(\ \exists X'', V^M \ \right)[X'/X] \qquad \text{et} \qquad T_D \ = \ \left(\ \exists X', V^M \ \right)[X''/X]$$

Cette condition n'est en général pas satisfaiteΓi.e.Γon a

$$\left(\ \exists X'', V^M \ \right) \odot \left(\ \exists X', V^M \ \right) \ \supset \ V^M$$

9.3.2 Encadrement des invariants

9.3.2.1 Borne supérieure

Supposons que $\left(\ \exists X'', V^M \ \right) \odot \left(\ \exists X', V^M \ \right) \ \supset \ V^M$. Soient les deux langages T_G^M et T_D^M définis par

$$T_G^M \ = \ \left(\ \exists X'', V^M \ \right)[X'/X] \qquad \text{et} \qquad T_D^M \ = \ \left(\ \exists X', V^M \ \right)[X''/X]$$

Alors toute solution (T_G, T_D) des inéquations [**INDUCTION**] est telle que $T_G \subseteq T_G^M$ et $T_D \subseteq T_G^M$. (T_G^M, T_D^M) peut être considéré comme une borne supérieure.

9.3.2.2 Borne inférieure

Si on pose $T_G = T_G^M$ afin d'avoir l'inégalité précédente il faut poser

$$T_D = \Big(\forall X' \big(\ \big(\ X^\infty \setminus T_G^M \ \big) [X/X'] \otimes V^M \ \big) \Big) [X''/X]$$

On pose donc

$$\begin{array}{rcl} T_G^m & = & \Big(\forall X', \big(\ X^\infty \setminus T_D^M \ \big) [X/X''] \otimes V^M \Big) [X'/X] \\ & = & \Big(\forall X', \big(\ X'^\infty \setminus \big(\ \exists X'', V^M \ \big) \ \big) \otimes V^M \Big) [X'/X] \quad \text{et} \\ T_D^m & = & \Big(\forall X'', \big(\ X^\infty \setminus T_G^M \ \big) [X/X'] \otimes V^M \Big) [X''/X] \\ & = & \Big(\forall X'', \big(\ X''^\infty \setminus \big(\ \exists X', V^M \ \big) \ \big) \otimes V^M \Big) [X''/X] \end{array}$$

Propriété 9.2 :

Toute solution (T_G, T_D) des inéquations [**INDUCTION**] vérifie

$$\begin{array}{rcccl} T_G^m & \subseteq & T_G & \subseteq & T_G^M \quad \text{et} \\ T_D^m & \subseteq & T_D & \subseteq & T_D^M. \end{array}$$

□

(T_G^m, T_D^m) peut être considéré comme une borne inférieure. En général T_G^m T_G^M T_D^m et T_D^M ne satisfont pas [**INDUCTION**].

En pratique nous ne connaissons qu'une approximation inférieure de V^M. Dans ce cas T_G^M et T_D^M ne sont plus forcément des bornes supérieures.

La section suivante va proposer une méthode basée sur des heuristiques pour déterminer des invariants corrects à partir des langages T_G^M et T_D^M.

9.3.3 Un algorithme de décomposition

Soit V^M le plus grand point fixe de [**INDUCTION'**]. Notre but est de déterminer deux langages T_G et T_D vérifiant

$$T_G[X/X'] \odot T_D[X/X''] \quad \subseteq \quad V^M \tag{9.1}$$

Ce problème n'a pas de solution unique. Ainsi $(T_G, T_D) = (\emptyset, \emptyset)$ est une solution triviale de 9.1 qui n'a aucun intérêt. Intuitivement pour que T_G et T_D satisfassent [**INDUCTION**] le produit $T_G[X/X'] \odot T_D[X/X'']$ doit être le plus proche possible de V^M. Nous proposons donc dans cette section des heuristiques permettant de calculer une solution de 9.1 qui satisfait également [**INDUCTION**].

9.3.3.1 Principe

Soient A_G et A_D deux automates sur X tels que $T_{A_G}[X/X'] \odot T_{A_D}[X/X''] \not\subseteq V^M$.

Le principe de notre algorithme est de partir d'un tel couple d'automates (A_G, A_D) bien choisi et d'interdire certaines transitions de A_G et de A_D de manière à obtenir deux nouveaux automates A'_G et A'_D tels que $T_{A'_G}[X/X'] \odot T_{A'_D}[X/X''] \subseteq V^M$.

Ainsi si $T_{A_G}[X/X'] \odot T_{A_D}[X/X''] \not\subseteq V^M$ il doit exister deux mots $\tau_G, \tau_D \in X^\infty$ acceptés respectivement par A_G et A_D et tels que $\tau_G[X/X'] \odot \tau_D[X/X'']$ ne soit pas élément de V^M. Nous pouvons alors soit interdire une transition de A_G de façon à ce que τ_G soit refusé soit interdire

une transition de A_D de façon à ce que τ_D soit refusé. Nous pouvons à priori interdire n'importe quelle transition du moment que les inclusions $T_P \subseteq T_{A_G}$ et $T_P \subseteq T_{A_D}$ sont préservées. Plus généralement il est nécessaire de préserver les inclusions

$$T_G^m \subseteq T_{A_G} \quad \text{et} \quad T_D^m \subseteq T_{A_D} \tag{9.2}$$

(voir section précédente).

9.3.3.2 Choix de A_G et A_D

En théorie il est possible de partir de n'importe quels automates A_G et A_D satisfaisant $T_{A_G}[X/X'] \odot T_{A_D}[X/X''] \not\subseteq V^M$. En pratique il est nécessaire que la structure de ces automates soit dérivée de celle de V^M. En effet si le langage X^∞ satisfait la non inclusion précédente il est clair que l'automate le reconnaissant (qui n'a qu'un état) n'est d'aucune utilité. Les automates A_G^M et A_D^M reconnaissant respectivement T_G^M et T_D^M satisfont ces propriétés.

Afin de préserver les inclusions 9.2 nous nous proposons de plus de marquer les transitions de A_G^M et de A_D^M qui ne peuvent pas être détruites. On choisit donc A_G comme étant l'automate reconnaissant T_G^M tel que tout mot de T_G^m soit reconnu par des transitions marquées et tout mot τ de $T_M^G \setminus T_G^m$ soit reconnu par des transitions dont au moins une ne soit pas marquée. A_D est choisi de même. Plus précisément pour tout mot $\tau = (\tau_1, \ldots, \tau_n)$ $n \leq \infty$ il existe une suite $(q_0, \ldots, q_n)$ d'états telle que $q_0 = q^0$ et pour tout i $q_i \xrightarrow[\emptyset]{\tau_{i+1}} q_{i+1}$. Alors

- $\tau \in T_G^m$ si et seulement si pour tout i la transition $q_i \xrightarrow[\emptyset]{\tau_{i+1}} q_{i+1}$ est marquée.
- $\tau \in T_G^M \setminus T_G^m$ si et seulement s'il existe un indice k $0 \leq k < n$ tel que pour tout $i \leq k$ la transition $q_i \xrightarrow[\emptyset]{\tau_{i+1}} q_{i+1}$ est marquée et pour tout $i > k$ la transition $q_i \xrightarrow[\emptyset]{\tau_{i+1}} q_{i+1}$ n'est pas marquée.

D'un point de vue algorithmique introduisons un nouveau signal $m \notin X$. Nous dirons qu'une transition $q \xrightarrow[\emptyset]{x} q'$ est marquée si et seulement si $m \in x$. Supposons que A_{G^m} et A_{D^m} possèdent chacun un état puits tel que le signal d'alarme α ne soit émis que par les transitions atteignant cet état et marquons toutes les transitions de A_{G^m} et A_{D^m}. On pose alors

$$A_G = A_{G^m} \times A_{G^M} \qquad \text{et} \qquad A_D = A_{D^m} \times A_{D^M}$$

9.3.3.3 Heuristiques

Nous proposons ici des heuristiques pour interdire des transitions de A_G ou de A_D afin de vérifier l'inclusion 9.1. Soit F_Δ la fonction qui prend en argument A_G et A_D et renvoie l'ensemble des mots sur $(X' \cup X'')$ éléments de $T_{A_G}[X/X'] \odot T_{A_D}[X/X'']$ et qui n'appartiennent pas à V^M.

$$F_\Delta\left(\ A_G, A_D\ \right) = \left\{\ \tau \mid \tau \in T_{A_G}[X/X'] \odot T_{A_D}[X/X''] \text{ et } \tau \notin V^M\ \right\}$$

Soit $\Delta = F_\Delta(A_G, A_D)$ et soient $\tau_G = (\tau_G^0, \ldots, \tau_G^{n-1})$ et $\tau_D = (\tau_D^0, \ldots, \tau_D^{n-1})$ deux mot sur $\left(\ X \cup \{m\}\ \right)^\infty$ tels que $\tau[X/X'] \odot \tau'[X/X''] \in \Delta$. Il existe alors deux suites $(q_0, \ldots, q_n)$ et $(q'_0, \ldots, q'_n)$ d'états telles que $q_0 = q^0$ $\forall i, q_i \xrightarrow[\emptyset]{\tau_G^i} q_{i+1}$ et $q'_0 = q^{0\prime}$ $\forall i, q'_i \xrightarrow[\emptyset]{\tau_D^i} q'_{i+1}$. Afin de satisfaire l'inclusion 9.1 nous avons à choisir deux indices k_G et k_D tels que soit la transition $q_{k_G} \xrightarrow[\emptyset]{\tau_G^{k_G}} q_{k_G+1}$ soit la transition $q_{k_D} \xrightarrow[\emptyset]{\tau_D^{k_D}} q_{k_D+1}$ est interdite. Nous pouvons choisir

- k_G ou k_D maximal : ça revient intuitivement à supprimer tous les mots qui ont un certain suffixe.
- k_G ou k_D minimal : ça revient intuitivement à supprimer tous les mots qui ont un certain préfixe.

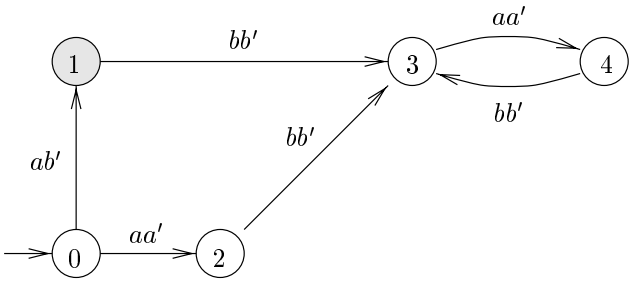


FIG. 9.2 – Un exemple d'automate reconnaissant le langage V. L'état grisé va être supprimé pour rendre cet automate symétrique

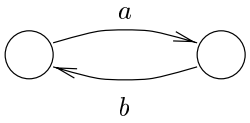


FIG. 9.3 – Résultat de la décomposition

L'expérience a montré que généralement seul le deuxième choix est efficace (voir chapitre 10).

Afin de formaliser l'algorithmeΓintroduisons la fonction f_m prenant en argument un mot τ et qui renvoie l'indice minimal k tel que $m \notin \tau^k$.

$$f_m(\tau) \quad = \quad \min\{ \ k \mid m \notin \tau^k \ \}$$

Notre algorithme de décomposition est alors le suivant :

Algorithme 9.1 :

$A_G = A_{G^m} \times A_{G^M}$; $A_D = A_{D^m} \times A_{D^M}$;

Tant que $T_{A_G}[X/X'] \odot T_{A_D}[X/X''] \not\subseteq V^M$

Soit $\tau_G \odot \tau_D = F_\Delta(A_G, A_D)$

Soit $k_G = f_m(\tau_G)$ et $k_D = f_m(\tau_D)$

On pose:
$$\begin{array}{rrcll} & \delta^O\ (\ q, (\exists m, \tau_G^{k_G})\ ) & = & \{\alpha\} & \text{si } k_G \leq k_D \\ & & = & \emptyset & \text{sinon} \\ & \delta^O\ (\ q', (\exists m, \tau_D^{k_D})\ ) & = & \{\alpha\} & \text{si } k_D < k_G \\ & & = & \emptyset & \text{sinon} \end{array}$$

Fin tant que

□

Exemple 9.1 :

On considère l'ensemble de signaux $\{a, b\}$. L'automate de la figure 9.2 reconnaît le langage V défini par

$$V \ = \ ab'.bb'.(aa'.bb')^* + (aa'.bb')^*$$

Le mot $(aa'.bb')^*$ est symétriqueΓdans le sens qu'en échangeant les variables primes et non primesΓon obtient un mot du langage V. Par contreΓsi $ab'.bb'.(aa'.bb')^*$ est

élément de VΓle mot symétrique $ba'.bb'.(aa'.bb')^*$ ne l'est pas. Ce mot est "effacé" en supprimant la transition $0 \xrightarrow{ab'} 1$. On obtient ainsi comme invariant le langage $(a.b)^*$Γ reconnu par l'automate présenté figure 9.3. □

L'algorithme précédent n'est pas parfaitement symétrique par rapport à G et D. On effectue les tests $k_G \leq k_D$ et $k_D < k_G$. Il est possible de définir un algorithme dual où on effectuerait les tests $k_G < k_D$ et $k_D \leq k_G$. On peut ainsi obtenir deux solutions (une pour chaque algorithme). Il n'est pas possible à priori de dire si l'une est meilleure que l'autre.

9.4 Généralisation aux grammaires de réseau

Le cas des arbres binaires peut s'étendre facilement au cas général des grammaires de réseau. Soit $\mathcal{G} =< T, N, \mathcal{P}, \mathcal{S} >$ une grammaire de réseau. On suppose que les règles de production sont de la forme suivante :

$$\forall R = (P \to P_1 \|_i \ldots \|_i P_k) \in \mathcal{P},\ \forall i, j = 1 \ldots k,\ \left(\ P_i = P_j \text{ et } P_i \in N\ \right) \Rightarrow\ i = j$$

i.e.Γtous les processus non terminaux apparaissant à droite d'une règle de production sont distincts. Si $\mathcal{G}$ n'est pas de ce typeΓil est facile de construire une grammaire $\mathcal{G}'$ ayant cette propriété et générant les même processus que $\mathcal{G}$. Pour celaΓil suffit de dédoubler les processus qui interviennent plus d'une fois à droite d'une même règle de productionΓcomme cela a été fait pour les grammaires binaires au début de ce chapitre.

Comme précédemmentΓsupposons notre problème résoluΓi.e.Γon connaît pour chaque symbole non terminal P un invariant I_P. Considérons alors le vecteur V de ces invariants. En utilisant les techniques des sections précédentesΓon peut exprimer le vecteur V comme la solution d'une équation de plus grand point fixe. Une solution non nulle V^M peut alors être décomposée en utilisant les heuristiques de la section précédente. L'utilisation des ces heuristiques nécessite de définir un ordre parmi ces processus non terminaux. Si N est de taille nΓil y a à priori $n!$ décompositions possibles.

Cette technique n'a été implémentée que dans le cas binaire. A cause de sa complexitéΓil semble assez peu réaliste de vérifier des grammaires plus complexes.

Chapitre 10

Exemples

Les techniques de synthèse d'invariant ont été implémentées (voir partie III). Les exemples présentés dans ce chapitre ont été programmés en LUSTRE [HCRP91].

Dans ce chapitre et le suivant on utilisera la syntaxe de description suivante (qui est équivalente à LUSTRE). Si q est une variable d'état **next** q représente sa valeur dans l'état suivant. Toutes les variables sont supposées être booléennes et initialisées à faux. Les calculs ont été effectués sur un **Ultra-Sparc 1** avec 256 Mo de mémoire.

10.1 Un algorithme simplifié du jeton circulant

Ce premier exemple est très simple. Il a été choisi car c'est le seul qui produise des invariants suffisamment petits pour être observés. Il fournit de plus un argument en faveur de notre choix de calculer le plus grand point fixe T_I^M au lieu du plus petit T_I^m.

10.1.1 Description de l'algorithme

Soient n processus $P_0 P_1 \ldots P_{n-1}$ se partageant une ressource en exclusion mutuelle. Ces processus sont connectés en anneau dans lequel un jeton peut circuler. La grammaire définissant le réseau est la suivante

$$\begin{array}{lcl} \mathcal{S} & \rightarrow & E \|_1 A \\ A & \rightarrow & P \|_2 A \\ A & \rightarrow & P \end{array}$$

E et P sont des symboles terminaux représentant respectivement un environnement et un processus.
Chaque processus P possède les propriétés suivantes :

- Il ne peut utiliser la ressource que s'il possède le jeton.
- Dès qu'un processus a fini d'utiliser la ressource il passe le jeton à son voisin.
- Lorsqu'un processus reçoit le jeton soit il le garde et il utilise la ressource soit il le transmet à son voisin.

Un processus P possède 3 signaux de communication :

- tk_{in} pour "token in" : le processus reçoit un jeton.
- tk_{out} pour "token out" : le processus émet un jeton.

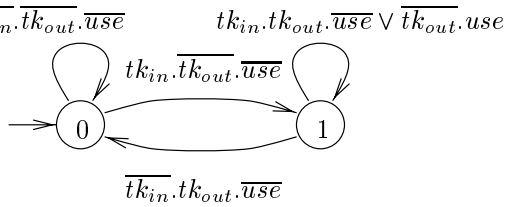


FIG. 10.1 – Un processus du jeton circulant

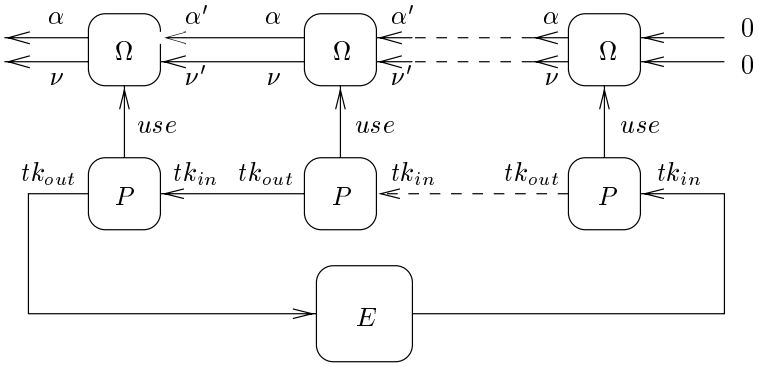


FIG. 10.2 – Un jeton circulant

– *use* pour "use resource" : le processus utilise la ressource.

L'observateur du processus est décrit par les équations booléennes suivantes :

$$\begin{array}{lcl} use & = & has\_tk.req \\ tk_{out} & = & has\_tk.\overline{req} \\ \textbf{next}\ has\_tk & = & tk_{in} \vee (has\_tk.\overline{tk_{out}}) \end{array}$$

has_tk est une variable interne qui vaut vrai si et seulement si le processus possède le jeton. *req* est un signal d'entrée qui vaut vrai lorsque le processus demande la ressource. L'automate de la figure 10.1 représente le comportement d'un processus (où le signal *req* a été caché). Dans l'état 0Γ le processus n'a pas le jeton et ne peut pas utiliser la ressource. Dans l'état 1Γle processus possède le jeton et peut utiliser la ressource.

A chaque processus est associé un observateur : celui-ci possède deux signaux d'observation :

– ν : la ressource est utilisée par le processus ou par l'un de ses voisins de droite.

– α : c'est un signal d'alarme signifiant que la propriété d'exclusion mutuelle est violée.

(voir figure 10.2). La tâche de l'environnement E est de fermer l'anneauΓi.e.Γde reboucler le jeton sortant du dernier processus vers le premier.

L'opérateur de composition $\|_2$ met en relation la sortie tk_{out} du sous-réseau de gauche avec l'entrée tk_{in} du sous-réseau de droite. A cet opérateur est associé un opérateur de composition sur les observateurs qui calcule les nouveaux signaux d'observation ν et α à l'aide de la formule

$$\alpha_{out} = \alpha_{in} \vee (\nu_{in}.use) \quad \text{et} \quad \nu_{out} = \nu_{in} \vee use$$

(voir figure 10.2).

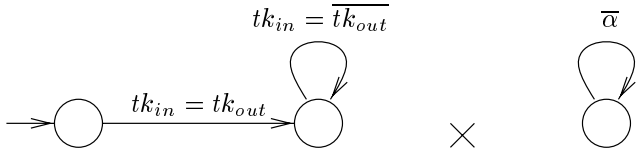


FIG. 10.3 – Calcul de l'automate $E \Rightarrow \varphi$

10.1.2 Calcul d'un invariant

10.1.2.1 Calcul en avant

Cet exemple montre l'inefficacité du calcul en avant. Le tableau suivant donne les tailles des automates obtenus à chaque pas de calcul (le calcul a été arrêté au bout de 4h) :

	Calcul en avant		Calcul en arrière	
Pas de calcul	Nb d'états	Nb de transitions	Nb d'états	Nb de transitions
0	2	6	3	21
1	8	32	5	58
2	97	808	8	80
3	>4h		Convergence	

10.1.2.2 Calcul en arrière

On cherche à vérifier la propriété φ d'exclusion mutuelle. Comme il n'est possible de construire par récurrence qu'un réseau "ouvert" (i.e.Γqui n'est pas rebouclé en anneau)Γon va chercher à démontrer que

Le réseau ouvert satisfait la propriété φ dans l'environnement E :

$$E \implies \varphi$$

L'automate reconnaissant $E \Rightarrow \varphi$ est présenté figure 10.4 : le signal α correspond au signal d'alarme de la propriété φ. Il est construit de la manière suivante : on construit tout d'abord le produit synchrone $E \times \Omega_\varphi$ (voir figure 10.3). Tant que l'assertion sur l'environnement est satisfaite (états 0 et 1 de la figure 10.4)Γle signal α ne doit pas être émis. Lorsque l'assertion sur l'environnement est violéeΓla propriété est considérée comme étant satisfaite pour toujours; le signal α peut alors être émis. On ajoute donc un état puits (l'état 2Γdans la figure 10.4)Γqui est atteint lorsque l'assertion sur l'environnement est violée.

Les invariants calculés (en moins d'une seconde par notre outil) lors des 3 pas de calcul sont montrés respectivement figure 10.4Γ10.5 et 10.6.

Le pas 0 (figure 10.4) montre la propriété à vérifier (i.e.Γ$E \Rightarrow \varphi$). L'automate a 3 états :

État 0 : État initial. L'environnement doit envoyer un jeton. L'automate passe alors dans l'état 1. Dans le cas contraireΓl'environnement est incorrect et on ne s'intéresse pas au comportement du système (voir ci-dessus).

État 1 : C'est le comportement normal du réseau sous un environnement correct. L'exclusion mutuelle est vérifiée ($\overline{\alpha}$) et le jeton est rebouclé ($tk_{in} = tk_{out}$).

État 2 : État chaos. L'environnement a été violéΓtous les comportements sont acceptés.

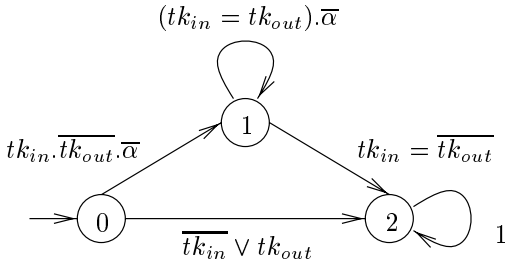


FIG. 10.4 – Pas 0 du calcul de l'invariant du jeton circulant simple. Cet automate représente la propriété à démontrer $E \implies \varphi$

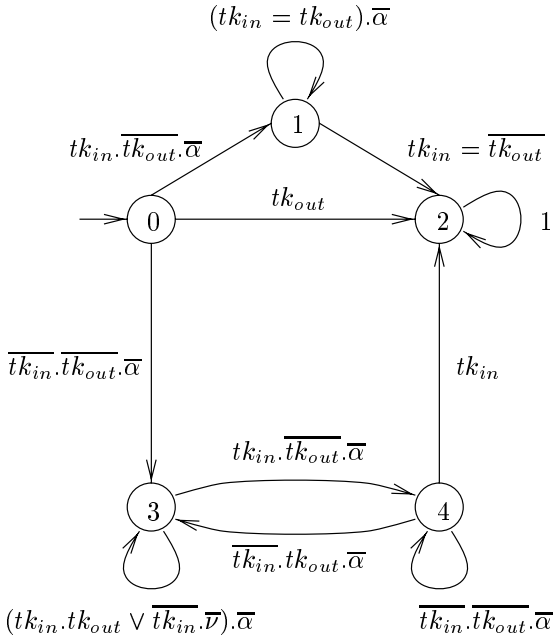


FIG. 10.5 – Pas 1 du calcul de l'invariant du jeton circulant simple

La figure 10.5 montre le premier pas de calcul. Le langage reconnu par cet automate est plus petit que celui reconnu par l'automate du pas 0. Il est intéressant de comparer ces deux automates pour comprendre le fonctionnement de l'algorithme. Les états à priori "utiles" de ces automates sont les états 0 et 1. Ils correspondent aux états vérifiant l'exclusion mutuelle. L'état 2 correspond à la violation de l'environnement. IntuitivementΓon pourrait penser que l'algorithme va essayer de renforcer la partie "utile"Γen raffinant le comportement du réseau dans un bon environnement. PourtantΓon se rend compte que l'état 1 n'est pas modifié (c'est également le cas dans le pas 2). En faitΓles deux nouveaux états 3 et 4 correspondent à des comportements lorsque l'environnement est incorrect. AinsiΓla transitions $0 \xrightarrow{\overline{tk_{in}}.\overline{tk_{out}}} 3$ correspond à un environnement qui n'émet pas de jeton à l'instant 0. La transition $4 \xrightarrow{tk_{in}} 2$ correspond à un réseau qui a reçu deux jetons : dans ce cas la propriété d'exclusion mutuelle peut être violée.

Le même phénomène se produit au pas 2 (mais en plus compliquéΓvoir figure 10.6). Les états 3 et 4 sont "éclatés" respectivement en des états $3'\Gamma 3''$ et $4'\Gamma 4''\Gamma$suivant que le réseau a ou n'a pas de jeton.

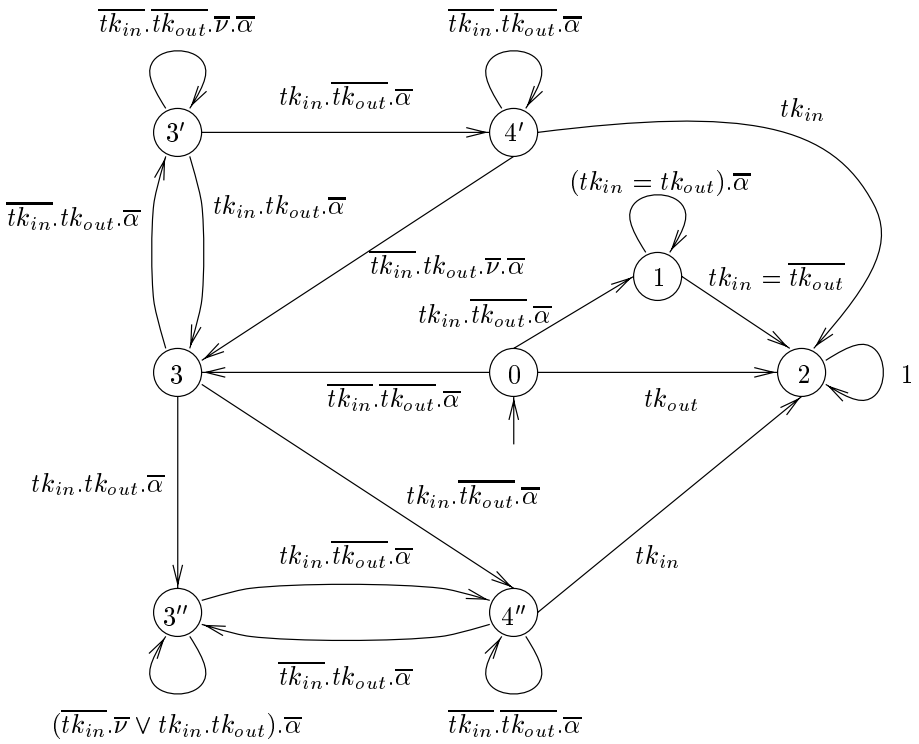


FIG. 10.6 – Pas 2 de calcul de l'invariant du jeton circulant simple. Cet automate représente le plus grand point fixe T_I^M (voir section 8.1Γpage 88)

Ces 3 invariants ont été calculés sans extrapolation. Cela signifie que l'automate de la figure 10.6 est exactement égal à T_I^MΓi.e.Γqu'il est le plus grand invariant du réseau permettant de prouver la propriété d'exclusion mutuelle. La convergence de cet exemple est un argument positif en faveur du calcul en arrière d'invariant.

Il est à noter que l'exclusion mutuelle sur ce jeton circulant aurait pu être démontrée en utilisant les résultats de [EN95] (voir section 6.3.2Γpage 66).

10.2 L'algorithme du jeton circulant de Dijkstra

Cet algorithme est adapté de celui utilisé dans [CGJ95][1]. Comme précédemmentΓon considère n processus P_0ΓP_1Γ... ΓP_{n-1} qui se partagent une ressource en exclusion mutuelle. Les processus sont connectés en anneau dans lequel un jeton peut circuler dans le sens des aiguilles d'une montre. Pour éviter de faire passer inutilement le jetonΓun signal de requête peut voyager dans le sens contraire des aiguilles d'une montre (voir figure 10.7).

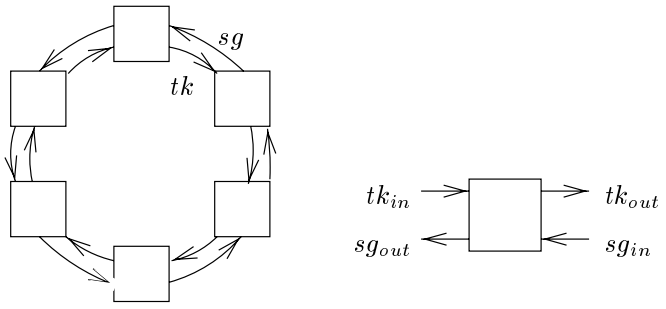
Un processus ne peut utiliser la ressource que s'il a le jeton. Dès qu'un processus a besoin du jetonΓil émet un signal vers sa gauche. Quand le processus qui a le jeton reçoit le signal de requêteΓ il transmet le jeton vers sa droite.

Chaque processus a 2 signaux d'entrée et 2 signaux de sortie :

– tk_{in} pour "token in" : le processus reçoit un jeton.

– sg_{in} pour "signal in" : le processus reçoit un signal.

1. Cet algorithme a en fait été inventé et présenté pour la première fois par Martin dans [Mar85], sous le nom de "reflecting privilege algorithm"



Le réseau

Un processus

FIG. 10.7 – Le jeton circulant de Dijkstra

– tk_{out} pour "token out" : le processus émet un jeton.

– sg_{out} pour "signal out" : le processus émet un signal.

ainsi que 2 signaux internes *req* et *rel* correspondant à la requête ("request") et la libération ("release") du jeton. Un processus est décrit par les équations suivantes :

$$\begin{array}{lcl}
\textbf{next}\ wait & = & (req \vee wait).\overline{tk_{in}}.\overline{has_tk} \\
\textbf{next}\ right_req & = & (right_req \vee sg_{in}).\overline{tk_{out}} \\
\textbf{next}\ has_tk & = & (has_tk \vee tk_{in}).\overline{tk_{out}} \\
tk_{out} & = & (right_req \vee sg_{in}).(tk_{in} \vee has_tk).\overline{wait}.\overline{req \vee rel} \\
sg_{out} & = & \overline{right_req}.\overline{has_tk \vee tk_{in}}.\overline{wait}.(sg_{in} \vee req)
\end{array}$$

L'observateur d'exclusion mutuelle a deux signaux d'observation en plus des signaux de communication utilisés par les processus : ν ("use resource") et α ("alarm signal" violation de l'exclusion mutuelle).

Cette exemple montre que prouver une propriété forte est parfois plus facile que montrer une propriété faible. Par exemple pour montrer qu'il y a un et un seul jeton dans le réseau un invariant est calculé en 3 pas et 1 extrapolation en 7 secondes. L'automate de l'invariant calculé a 33 états et 1483 transitions.

Par contre pour montrer qu'il y a toujours au plus un jeton dans le réseau (une propriété plus faible que précédemment) le calcul de l'invariant prend 19 secondes en 3 pas et 1 extrapolation. Son automate a 42 états et 1954 transitions.

L'exemple suivant met davantage en évidence ce phénomène.

Le tableau suivant montre la taille des automates générés pour montrer la propriété un et un seul jeton (le calcul a été arrêté au bout de 4h) :

	Calcul en avant		Calcul en arrière	
Pas de calcul	Nb d'états	Nb de transitions	Nb d'états	Nb de transitions
0	6	34	3	176
1	36	280	11	532
2	66	530	35	1588
3	152	1233	46	2071
4	430	3488	33	1483
5	1393	11272	Convergence	
5	>4h		Convergence	

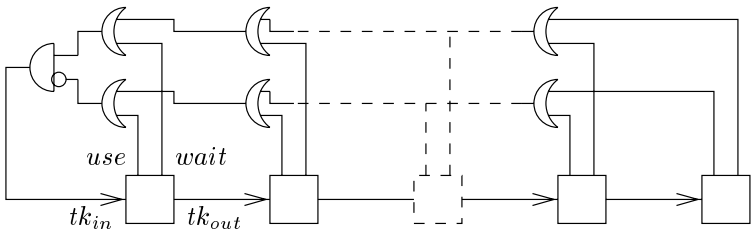


FIG. 10.8 – Un système d'arbitrage

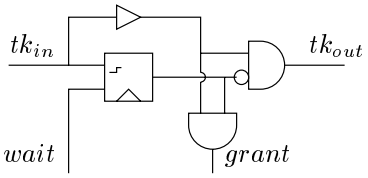


FIG. 10.9 – Un processus de l'arbitre

10.3 Un arbitre matériel

Le troisième exemple vient de [HLR92a] : comme précédemment on a n processus partageant une ressource en exclusion mutuelle. Les processus sont servis selon une règle de priorité fixée : dès que la ressource est libre et qu'un processus la demande un jeton est émis qui voyage de processus en processus à travers le réseau jusqu'à ce qu'il soit pris par un des processus demandant la ressource (voir figure 10.8).

La figure 10.9 montre le comportement d'un processus comme un circuit. La ressource est attribuée au processus lorsque celui ci reçoit un front montant du signal correspondant au jeton (tk_{in}) : ce signal est reconnu à l'aide d'une bascule D. Selon la sortie de la bascule le jeton provoque l'attribution de la ressource au processus ou est passé au processus suivant. Un processus est décrit par le système d'équations suivant :

$$
\begin{array}{lcl}
\textbf{next}\ flop & = & (edge.wait) \vee (\overline{edge}.flop) \\
edge & = & tk_{in}.was\_low \\
\textbf{next}\ was\_low & = & \overline{tk_{in}} \\
\textbf{next}\ tk_{out} & = & tk_{in}.\overline{\textbf{next}\ flop} \\
\textbf{next}\ grant & = & tk_{in}.\textbf{next}\ flop \\
\textbf{next}\ use & = & \textbf{next}\ grant \vee (use.\overline{\textbf{next}\ release}) \\
\textbf{next}\ wait & = & \overline{\textbf{next}\ use}.(\textbf{next}\ req \vee wait)
\end{array}
$$

Le jeton le plus à gauche est émis dès que la ressource est demandée et non utilisée. Pour décrire ce comportement chaque processus est associé à un processus d'arbitrage implémentant une porte "or". Il reçoit 4 signaux :

- *right_requested* : un processus demande la ressource sur la droite.
- *right_used* : un processus utilise la ressource sur la droite.
- *use* et *wait* qui viennent du processus.

Il émet les deux signaux *requested* et *used* définis par

$$
\begin{array}{lcl}
requested & = & right\_requested \vee wait \\
used & = & right\_used \vee use
\end{array}
$$

Le jeton est donc émis par le système d'arbitrage lorsqu'il reçoit $requested.\overline{used}$.

$$tk \;=\; requested.\overline{used}$$

Les propriétés suivantes ont été vérifiées :

Exclusion mutuelle : en ajoutant à chaque processus un observateur comme dans les exemples précédents.

Pas de perte de jeton : le processus le plus à droite n'émet jamais le jeton. Cette propriété est décrite par une légère modification de l'observateur de la propriété.

Priorité : ceci est un exemple de propriété temporelle non triviale. Idéalement l'arbitre devrait satisfaire une règle de priorité du type suivant

$$grant_i \implies \overline{wait_j}$$

pour toute paire (i, j) telle que $j < i$. Cette règle n'est pas satisfaite par ce système d'arbitrage car elle nécessiterait une connaissance instantanée de toutes les demandes de la ressource. En fait l'arbitre satisfait la règle de priorité plus faible suivante : si la ressource est attribuée au processus P_i à l'instant t aucun processus n'attendait la ressource au dernier arbitrage précédent t où on appelle arbitrage un front montant du signal correspondant au jeton. Cette propriété peut être exprimée en associant à chaque processus un observateur avec une connaissance instantanée des demandes. Bien sûr cette connaissance instantanée est nécessaire pour la spécification de la propriété et ne change pas le circuit lui-même. Chaque observateur reçoit un signal $prio_{in}$ si un processus plus prioritaire attendait la ressource lors du dernier arbitrage et émet un signal $prio_{out}$ défini par

$$\begin{array}{lcl} \alpha_{out} & = & \alpha_{in} \vee (prio_{in}.grant) \\ prio_{out} & = & prio_{in} \vee prio \\ \textbf{next}\ prio & = & (arb\_req.wait) \vee \\ & & (\overline{arb\_req}.prio) \end{array}$$

Nous avons essayé de vérifier chaque combinaison de ces 3 propriétés. Le tableau suivant montre les résultats : pour chaque combinaison de propriétés la table donne le nombre d'extrapolations le nombre de pas de calcul le nombre d'états et de transitions de l'invariant final et le temps de calcul total. Toutes les combinaisons ont pu être vérifiées sauf une : si on considère la propriété "pas de perte de jeton" le calcul ne semble pas converger (la mémoire est saturée au bout de plusieurs heures).

Exclusion mutuelle	x	x	x	x			
Pas de jeton perdu		x	x		x	x	
Règle de priorité			x	x		x	x
Nb d'élargissements	2	1	1	1	NC	1	3
Nb de pas de calcul	4	3	3	3	NC	3	5
Nb d'états	43	9	14	14	NC	14	17
Nb de transitions	745	166	459	453	NC	273	326
Temps de calcul	35"	11"	9"	12"	NC	15"	49"

Le tableau suivant montre la taille des automates générés pour montrer l'ensemble de ces 3 propriétés (le calcul a été arrêté au bout de 4h) :

	Calcul en avant		Calcul en arrière	
Pas de calcul	Nb d'états	Nb de transitions	Nb d'états	Nb de transitions
0	7	26	2	100
1	75	399	12	417
2	1105	9052	27	871
3	>4h		46	1439
4	>4h		14	459
5	>4h		Convergence	

Commentaires : Pour diminuer les temps de calculΓil faut

1. Minimiser le nombre de signaux d'observation utilisés. En effetΓpour chaque signal supplémentaireΓle nombre d'événements à traiter est multiplié par 2. Dans le calcul d'un composition (où interviennent les signaux $X\Gamma X'\Gamma X''$) le nombre d'événements est multiplié par 8.

2. Renforcer la propriété à démontrer. L'exemple précédent montre que le calcul de l'invariant pour une propriété φ peut être très rapideΓet ne pas converger pour une propriété φ' plus faible que φ.

10.4 Jeton circulant arborescent

Cet exemple est une extension du jeton circulaire vu à la section 10.1. On considère n processus $P_0\Gamma P_1\Gamma\ldots\ \Pi P_{n-1}$ qui se partagent une ressource en exclusion mutuelle. Les processus sont connectés dans une structure arborescente dans lequel un jeton peut circuler en profondeur.

Un processus ne peut utiliser la ressource que s'il a le jeton. Il possède un signal d'entrée et deux signaux de sortie :

- tk_{in} pour "token in" : le processus reçoit le jeton.
- tk_{out} pour "token out" : le processus émet le jeton.
- use pour "use resource" : le processus utilise la ressource.

Le processus est défini de la même manière qu'à l'exemple de la section 10.1Γpage 109.

$$\begin{array}{lcl} use & = & has\_tk.req \\ tk_{out} & = & has\_tk.\overline{req} \\ \textbf{next}\ has\_tk & = & tk_{in} \vee (has\_tk.\overline{tk_{out}}) \end{array}$$

Nous présentons deux versions de cet algorithme. Une où les processus sont situés sur les feuillesΓl'autre où les processus sont situés sur les noeuds de l'arbre.

10.4.1 Les processus sont sur les feuilles

Le jeton circule dans la structure arborescente. Les processus sont situés sur les feuilles de cet arbre (voir figure 10.10).

Chaque noeud a 3 signaux d'entrée et 3 signaux de sortieΓcorrespondant aux communications avec son pèreΓson fils droit et son fils gauche.

Lorsqu'un noeud reçoit un jeton de son pèreΓil le transmet à son fils gauche. Lorsqu'il le reçoit de son fils gaucheΓil le transmet à son fils droit. Et enfin lorsqu'il le reçoit de son fils droitΓil le rend à son père.

L'observateur d'exclusion mutuelle d'un noeud accepte 4 signaux supplémentaires en entrée :

- ν_{left} : la ressource est utilisée dans la branche gauche.

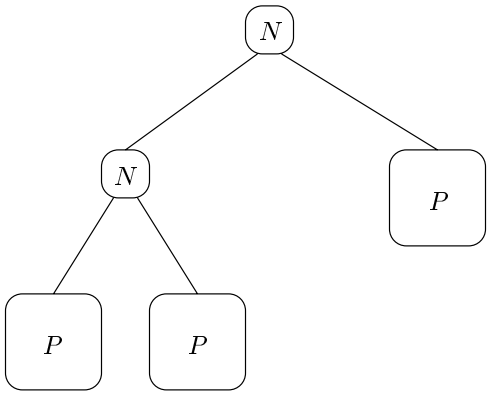


FIG. 10.10 – Première version du jeton arborescent. Les processus sont situés sur les feuilles de l'arbre. P représente un processusΓN un noeud.

- α_{left} : l'exclusion mutuelle est violée dans la branche gauche.
- ν_{right} : la ressource est utilisée dans la branche droite.
- α_{right} : l'exclusion mutuelle est violée dans la branche droite.

Il émet les deux signaux classiques

- α : la propriété d'exclusion mutuelle est violée.
- ν : la ressource est utilisée par un de ses fils.

Le comportement d'un noeud est défini de la manière suivante :

$$
\begin{aligned}
\textbf{next } tk_{out}^{left} &= tk_{in} \\
\textbf{next } tk_{out}^{right} &= tk_{out}^{left} \\
\textbf{next } tk_{out} &= tk_{out}^{right} \\
\nu_{out} &= \nu_{left} \vee \nu_{right} \vee tk_{in}^{left} \vee tk_{in}^{right} \vee use \\
\alpha &= \alpha_{left} \vee \alpha_{right} \vee (\nu_{left}.\nu_{right}) \vee (\nu_{left}.use) \vee (\nu_{right}.use)
\end{aligned}
$$

L'outil calcule l'invariant V^M en 5 itérationsΓsans utiliser d'élargissement. V^M a 829 états et 74013 transitions. Cet invariant est alors décomposé en utilisant l'algorithme 9.1Γpage 107. Suivant l'ordre dans lequel cette décomposition est effectuéeΓnous obtenons deux fois deux invariants G et D qui ont les tailles données par le tableau suivant :

	Première solution		Deuxième solution	
Fils	Nb états	Nb transitions	Nb états	Nb transitions
G	32	276	32	276
D	5	49	7	57

L'ensemble du calcul prend 9 minutes et 57 secondes. Le tableau suivant donne les tailles des automates V obtenus à chaque pas de calcul (le calcul a été arrêté au bout de 4h) :

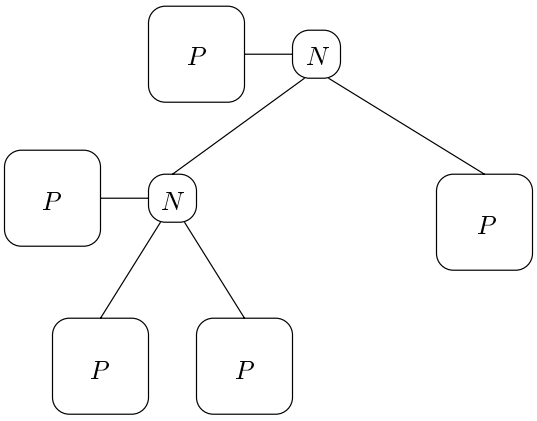


FIG. 10.11 – Deuxième version du jeton arborescent. Les processus sont situés sur les noeuds de l'arbre. P représente un processusΓN un noeud.

	Calcul en avant		Calcul en arrière	
Pas de calcul	Nb d'états	Nb de transitions	Nb d'états	Nb de transitions
0	2	6	5	1072
1	20	75	95	10084
2	>4h		296	30217
3	>4h		494	43814
4	>4h		554	50278
5	>4h		829	74013
6	>4h		Convergence	

10.4.2 Les processus sont sur les noeuds

On considère une deuxième version du jeton arborescentΓoù les processus sont situés sur les noeuds de l'arbre (voir figure 10.11).

Chaque noeud de la structure arborescente du réseau est associé à un processus avec lequel il peut communiquer : chaque noeud a donc 4 signaux d'entrée et 4 signaux de sortieΓcorrespondant aux communications avec son pèreΓson fils droitΓson fils gauche et son processus associé.

Lorsqu'un noeud reçoit un jeton de son pèreΓil le transmet à son fils gauche. Lorsqu'il le reçoit de son fils gaucheΓil le transmet à son fils droit. Et enfin lorsqu'il le reçoit de son fils droitΓil le rend à son père. A chaque passage du jeton dans le noeudΓil est également transmis au processus associé au noeudΓqui peut éventuellement le garder un certain temps pour utiliser la ressource.

La figure 10.12 montre le comportement d'un noeud. L'observateur d'exclusion mutuelle accepte en entrée un signal de plus que précédemment : *use* (la ressource est utilisée par son processus

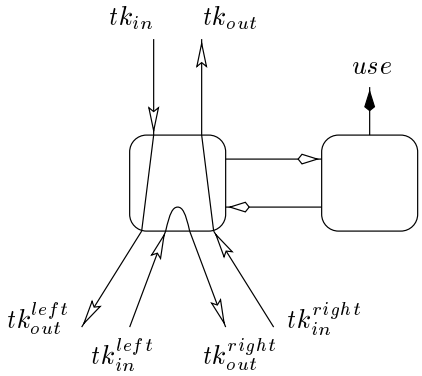


FIG. 10.12 – Un noeud du jeton circulant arborescent (deuxième version)

associé). Le comportement d'un noeud est défini de la manière suivante :

$$
\begin{array}{lcl}
(tk_{in}^{unit}, use) & = & \mathbf{Unit}(tk_{in} \vee tk_{in}^{left} \vee tk_{in}^{right}) \\
\mathbf{next}\left(tk_{from}^{father} \right) & = & \mathbf{next}\left(tk_{in} \right) \vee tk_{from}^{father}.\mathbf{next}\left(\overline{tk_{in}^{left}}.\overline{tk_{in}^{right}} \right) \\
\mathbf{next}\left(tk_{from}^{left} \right) & = & \mathbf{next}\left(tk_{in}^{left}.\overline{tk_{in}} \right) \vee tk_{from}^{left}.\mathbf{next}\left(\overline{tk_{in}}.\overline{tk_{in}^{right}} \right) \\
\mathbf{next}\left(tk_{from}^{right} \right) & = & \mathbf{next}\left(tk_{in}^{right}.\overline{tk_{in}}.\overline{tk_{in}^{left}} \right) \vee tk_{from}^{right}.\mathbf{next}\left(\overline{tk_{in}}.\overline{tk_{in}^{left}} \right) \\
tk_{out}^{left} & = & tk_{from}^{father}.tk_{in}^{unit} \\
tk_{out}^{right} & = & tk_{from}^{left}.tk_{in}^{unit} \\
tk_{out} & = & tk_{from}^{right}.tk_{in}^{unit} \\
\nu_{out} & = & \nu_{left} \vee \nu_{right} \vee tk_{in}^{left} \vee tk_{in}^{right} \vee use \\
\alpha & = & \alpha_{left} \vee \alpha_{right} \vee (\nu_{left}.\nu_{right}) \vee (\nu_{left}.use) \vee (\nu_{right}.use)
\end{array}
$$

La fonction **Unit** correspond à la traduction d'un processus. Les variables tk_{from}^{father}Γ tk_{from}^{left} et tk_{from}^{left} signifient respectivement que le noeud (ou son processus associé) possède le jetonΓet qu'il l'a reçu respectivement de son pèreΓde son fils gauche et de son fils droit.

L'outil calcule l'invariant V^M en 5 itérationsΓsans utiliser d'élargissement. V^M a 928 états et 72379 transitions. Cet invariant est alors décomposé en utilisant l'algorithme 9.1Γpage 107. Suivant l'ordre dans lequel cette décomposition est effectuéeΓnous obtenons deux fois deux invariants G et D qui ont les tailles données par le tableau suivant :

	Première solution		Deuxième solution	
Fils	Nb états	Nb transitions	Nb états	Nb transitions
G	27	266	27	266
D	15	111	15	105

L'ensemble du calcul prend 19 minutes et 15 secondes. Le tableau suivant donne les tailles des automates V obtenus à chaque pas de calcul (le calcul a été arrêté au bout de 4h) :

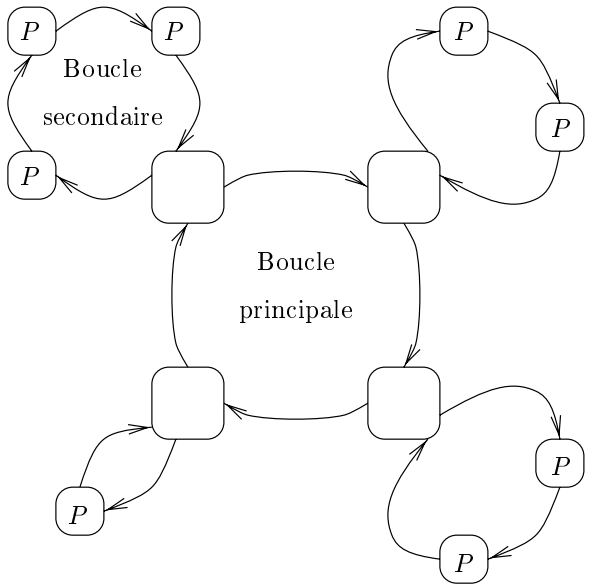


FIG. 10.13 – Réseau en pétales

	Calcul en avant		Calcul en arrière	
Pas de calcul	Nb d'états	Nb de transitions	Nb d'états	Nb de transitions
0	2	6	5	1072
1	23	90	88	8809
2	>4h		178	16882
3	>4h		511	42256
4	>4h		780	65146
5	>4h		928	72379
6	>4h		Convergence	

10.5 Réseau en pétales

Ce dernier exemple montre que la technique de calcul d'invariant pour les réseaux arborescents binaires peut s'appliquer à des réseaux asymétriques (où les fils droit et gauche sont définis différemment).

Considérons un réseau circulaire principalΓsur lequel peuvent se greffer des réseaux circulaires secondaires (voir figure 10.13).

Ce type de réseau peut être généré par la grammaire suivante :

$$\begin{aligned} \mathcal{S} &\rightarrow G\|_1 D \\ G &\rightarrow G\|_2 P \\ D &\rightarrow G\|_1 D \end{aligned}$$

(voir figure 10.14).

On considère un jeton circulant en profondeur dans toutes les branches de l'arbre. Un processus est défini comme à la section 10.1Γpage 109.

Lorsqu'un noeud principal $\|_1$ reçoit le jetonΓil le transmet à son fils gauche (i.e.Γune boucle secondaire associée) puis à son fils droit (i.e.Γau noeud principal suivant). Lorsqu'un noeud secondaire $\|_2$ reçoit le jetonΓil le transmet à son fils droit (i.e.Γun processus) puis à son fils gauche

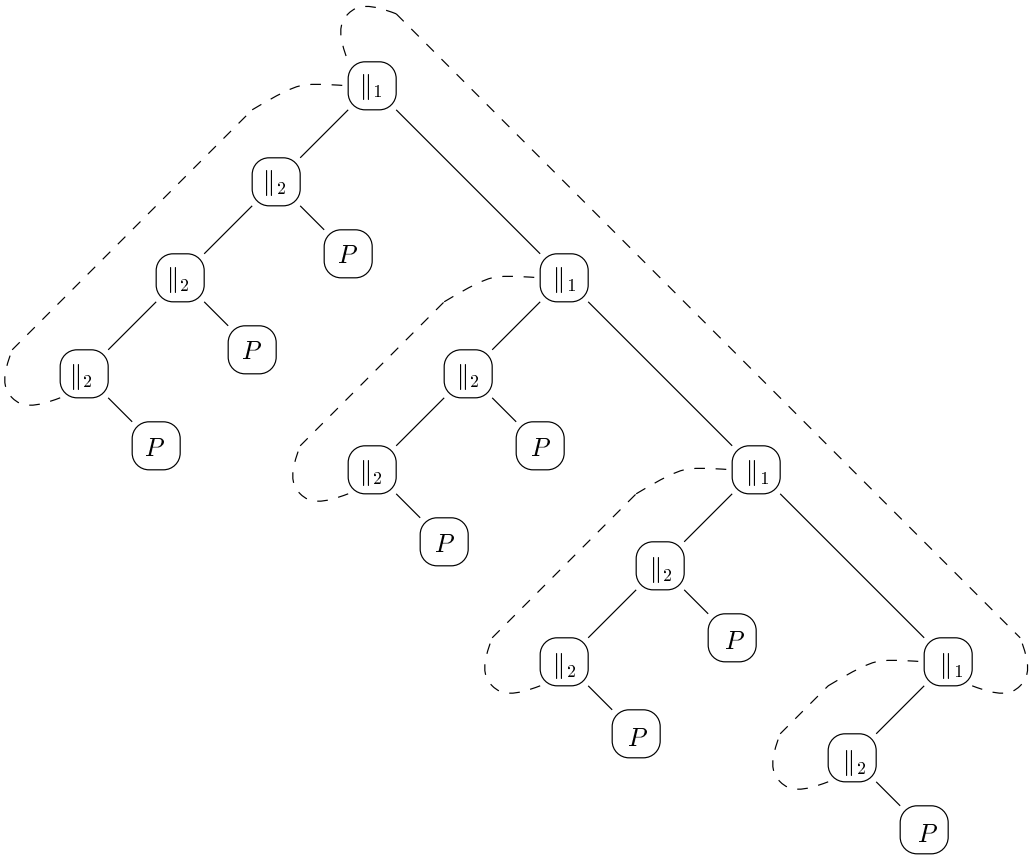


FIG. 10.14 – Construction d'un réseau en pétales à l'aide d'une grammaire arborescente binaire

(i.e. au noeud secondaire suivant). Ainsi le jeton parcourt l'ensemble des processus du réseau les uns après les autres.

Le calcul d'invariant ne converge pas pour cet exemple (la mémoire est saturée en quelques heures). Nous avons eu l'idée que cette non convergence pouvait venir de l'absence de règles d'initiation. Nous avons donc ajouter à la grammaire précédente les contraintes suivantes :

$$\begin{aligned} G &\rightarrow P \\ D &\rightarrow P \end{aligned}$$

i.e. les deux invariants G et D doivent être plus grands (dans le sens de l'inclusion des traces) qu'un processus. Toute solution des équations d'induction de cet grammaire sera également solution des équations de la grammaire précédente.

Pour ce nouveau réseau l'outil calcule l'invariant V^M en 3 itérations sans utiliser d'élargissement. V^M a 612 états et 50782 transitions. Cet invariant est alors décomposé en utilisant l'algorithme 9.1 page 107. Suivant l'ordre dans lequel cette décomposition est effectuée nous obtenons deux fois deux invariants G et D qui ont les tailles données par le tableau suivant :

	Première solution		Deuxième solution	
Fils	Nb états	Nb transitions	Nb états	Nb transitions
G	32	284	32	284
D	8	79	16	127

L'ensemble du calcul prend 18 minutes et 3 secondes. Le tableau suivant donne les tailles des automates V obtenus à chaque pas de calcul (le calcul a été arrêté au bout de 4h) :

	Calcul en avant				Calcul en arrière	
Pas de calcul	Nb d'états		Nb de transitions		Nb d'états	Nb de transitions
	G	D	G	D		
0	2	2	6	6	9	1888
1	13	13	45	45	335	33296
2	196	2435	979	13909	612	50814
3	>4h				612	50782
4	>4h				Convergence	

Dans la section 11.2.2Γpage 127Γnous présenterons une amélioration de cet algorithmeΓpour laquelle l'exclusion mutuelle sera prouvée à l'aide d'une abstraction supplémentaire.

Chapitre 11

Aide aux calculs d'invariants

11.1 Du calcul automatique au calcul semi-automatique

11.1.1 Introduction

Le problème du calcul d'un invariant pour un réseau de processus étant indécidable [AK86]Γil doit exister de nombreux exemples pour lesquels la convergence du plus grand point fixe n'est pas assuréeΓmême à l'aide d'extrapolation. Il existe en fait deux possibilités :

- Soit la suite d'extrapolations ne converge pas.
- Soit les extrapolations sont trop "fortes"Γet mènent à chaque application à la violation des propriétés initiales.

Dans ce chapitreΓnous allons montrer sur des exemples que dans le cas de non convergenceΓnotre technique peut néanmoins être d'une aide précieuse. Moyennant une (astucieuse) intervention humaineΓnotre outil est capable de trouver rapidement un invariant du réseau.

11.1.2 Principe

Considérons le cas des réseaux linéaires. Pour simplifierΓprenons un réseau qui ne possède qu'un processus terminalΓet reprenons l'inéquation d'induction

$$I \parallel P \preceq I$$

(voir section 7.1.3.2Γpage 75). L'ensemble des invariants I satisfaisant cette équation dépend de deux paramètres: $\parallel$ et P. L'idée de ce chapitre est d'effectuer un calcul d'invariant sur une inéquation dans laquelle on aurait changé un de ces deux paramètres. Le plus facile est de modifier P.

AinsiΓsi on choisit un processus P' tel que $P \preceq P'$ΓalorsΓtoute solution I' de la nouvelle inéquation

$$I' \parallel P' \preceq I'$$

est également une solution de l'inéquation d'induction. En effetΓ

$$I' \parallel P \preceq I' \parallel P' \preceq I'$$

S'il se trouve que I' est plus grand que P (ou que P')Γnous avons obtenu un invariant adéquat pour notre réseau.

Nous avons envisagé d'effectuer une approximation supérieure du processus P de 3 manières.

1. En effaçant des signaux sans rapport avec la propriété à démontrer. Le concepteur d'un algorithme a introduit des communications pour des raisons précises. Par exempleΓpour

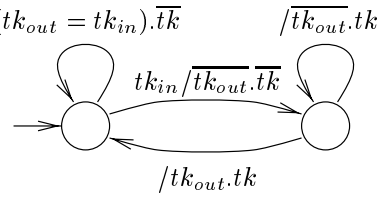


FIG. 11.1 – Un processus abstrait du jeton circulant de Dijkstra

vérifier une propriété d'exclusion mutuelle, il n'est pas nécessaire de considérer les signaux introduits pour gérer la priorité entre les processus.

2. En comprenant l'algorithme et en modifiant certains paramètres. Il est possible de modifier un algorithme qui vérifie toujours une propriété donnée et tel que les processus considérés soient plus grands que les processus du réseau initial. Cette technique dépend bien entendu uniquement de "l'intelligence" de l'utilisateur.

3. En utilisant des outils d'abstraction. Il est possible d'abstraire des processus contenant des données infinies et d'effectuer le calcul de point fixe sur cette abstraction (voir par exemple [Saï96, GS96, HMP95]).

Les sections suivantes montrent un exemple pour chacune de ces techniques.

11.2 Abstraction de signaux

11.2.1 Exemple du jeton circulant de Dijkstra

Cet exemple a déjà été entièrement traité section 10.2, page 113. Nous nous proposons ici de simplifier la preuve de l'exclusion mutuelle en simplifiant un processus. On rappelle qu'un processus peut communiquer à l'aide de 4 signaux :

- tk_{in} et tk_{out} qui correspondent respectivement à la réception et à l'émission du jeton.
- sg_{in} et sg_{out} qui correspondent respectivement à la réception et à l'émission d'un signal.

Seul le jeton sert à assurer l'exclusion mutuelle. Il est donc possible de simplifier ce réseau en abstrayant existentiellement les deux signaux sg_{in} et sg_{out} dans les description d'un processus de la fonction de composition et de l'environnement.

Le nouveau processus obtenu est présenté figure 11.1. Il ne possède plus que 2 états (contre 6 pour le processus concret) : l'état initial où le processus n'a pas le jeton et un état où le processus a le jeton. Notre outil calcule un invariant permettant de prouver

Il existe au plus un jeton dans le réseau.

Cet invariant ne fait que 4 états (contre 33 pour l'automate correspondant à l'invariant du réseau concret) : il est présenté figure 11.2.

Par contre, la propriété

Il existe un et un seul jeton dans le réseau.

n'est pas satisfaite par notre réseau abstrait. En effet, lorsqu'aucun processus ne demande la ressource, il est possible qu'aucun ne garde le jeton, et celui-ci est perdu. Dans le système concret, lorsque personne ne demande la ressource, le processus qui le possède est obligé de le garder.

Ce premier exemple très simple nous montre que certaines propriétés peuvent être vérifiées beaucoup plus simplement sur le système abstrait. Il nous montre aussi, que certaines propriétés peuvent ne plus être vraies.

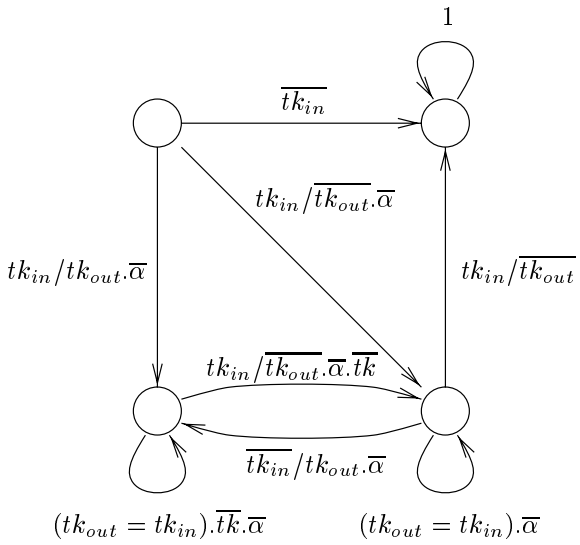


FIG. 11.2 – L'invariant du jeton circulant de Dijkstra abstrait

11.2.2 Réseau en pétales

Nous proposons une amélioration de l'algorithme d'exclusion mutuelle sur un réseau en pétales vu à la section 10.5Γpage 121. Cet amélioration est basée sur le principe de l'algorithme de Dijkstra (voir section précédente et section 10.2Γpage 113).

On rappelle que notre système est constitué d'un réseau circulaire principalΓsur lequel viennent se greffer des réseaux circulaires secondaires (voir figure 10.13Γpage 121). Il est décrit par la grammaire suivante :

$$\begin{aligned} \mathcal{S} &\rightarrow G\|_1 D \\ G &\rightarrow G\|_2 U \\ D &\rightarrow G\|_1 D \end{aligned}$$

Lorsqu'un processus situé sur une boucle secondaire a besoin de la ressourceΓil envoie un signal de requête vers le noeud principal correspondant. Lorsque le jeton arrive sur un noeud principal :

- Soit un signal de requête a été reçu (i.e.Γun processus de la boucle secondaire associée demande la ressource)Γet le jeton est envoyé dans cette boucle secondaire.

- Soit aucun signal de requête n'a été reçuΓet le jeton est transmis au noeud principal suivant.

La description d'un processus est proche de celle de l'algorithme de Dijkstra. Chaque processus a 1 signal d'entrée et 2 signaux de sortie :

- tk_{in} pour "token in" : le processus reçoit un jeton.

- tk_{out} pour "token out" : le processus émet un jeton.

- sg_{out} pour "signal out" : le processus émet un signal.

ainsi que 2 signaux internes *req* et *rel* correspondant à la requête ("request") et la libération

("release") du jeton. Un processus est décrit par les équations suivantes :

$$\begin{array}{lcl} \textbf{next}\ wait & = & (req \vee wait).\overline{tk_{in}}.\overline{has\_tk} \\ \textbf{next}\ has\_tk & = & (has\_tk \vee tk_{in}).\overline{tk_{out}} \\ tk_{out} & = & (tk_{in} \vee has\_tk).\overline{wait}.\overline{req \vee rel} \\ sg_{out} & = & \overline{has\_tk \vee tk_{in}}.\overline{wait}.req \end{array}$$

L'opérateur de composition $\|_1$ est définie de sorte que le jeton soit envoyé dans un réseau circulaire que si un signal de requête à été reçu. Il est défini de la manière suivante :

$$\begin{array}{rcl} \textbf{next}\ has\_sig & = & \textbf{next}\ sg_{in}^{left} \vee (has\_sig \wedge \overline{tk_{out}^{left}}); \\ \textbf{next}\ tk_{out}^{left} & = & \textbf{next}\ has\_sig \wedge tk_{in}; \\ \textbf{next}\ tk_{out}^{right} & = & tk_{in}^{left} \vee (\textbf{next}\ \overline{has\_sig} \wedge tk_{in}); \\ tk_{out} & = & tk_{in}^{right}; \end{array}$$

La définition de l'opérateur de composition $\|_2$ n'est pas modifiée.

Au bout de 24h de calculΓnotre outil n'a toujours pas terminé le second pas de calcul. Pour simplifier l'invariantΓnous effectuons comme précédemment une abstraction existentielle du signal de requête (qui n'est pas utilisé pour obtenir l'exclusion mutuelle). Sur le réseau abstrait ainsi obtenuΓle calcul se termine en 3h 18mnΓen trois pas. L'automate V^M a 777 états et 51711 transitions. Cet invariant est alors décomposé en utilisant l'algorithme 9.1Γpage 107. Suivant l'ordre dans lequel cette décomposition est effectuéeΓnous obtenons deux fois deux invariants G et D qui ont les tailles données par le tableau suivant :

	Première solution		Deuxième solution	
Fils	Nb états	Nb transitions	Nb états	Nb transitions
G	28	219	20	159
D	22	157	36	224

Le tableau suivant donne les tailles des automates V obtenus à chaque pas de calcul (le calcul a été arrêté au bout de 4h) :

	Calcul en avant				Calcul en arrière	
Pas de calcul	Nb d'états		Nb de transitions		Nb d'états	Nb de transitions
	G	D	G	D		
0	2	2	7	7	9	1680
1	17	16	85	87	1862	120658
2	>4h				761	53640
3	>4h				777	51711
4	>4h				Convergence	

11.3 Exemple de l'arbitre de McMillan

11.3.1 Description

Soient n processus se partageant une ressource en exclusion mutuelle.

L'objectif de cet arbitre est de permettre à chaque cycle d'horlogeΓl'accès à cette ressource pour un seul client à la fois. Les entrées du circuit sont un ensemble de signaux de requête ("request") $req_0, \ldots, req_{k-1}$Γet les sorties sont un ensemble de signaux d'acceptation ("acknowledge") $ack_0, \ldots, ack_{k-1}$. NormalementΓl'arbitre donne le signal d'acceptation au client le demandant qui a le plus petit numéro. NéanmoinsΓlorsqu'une requête devient trop fréquenteΠl'arbitre est construit

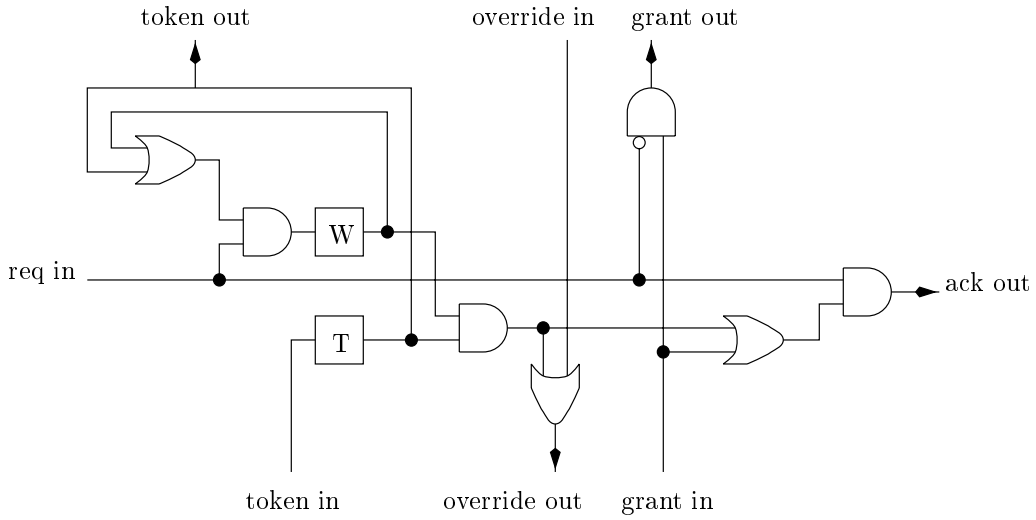


FIG. 11.3 – Une cellule de l'arbitre de McMillan

de manière à ce que toute requête finisse par être satisfaite. AinsiΓon associe à chaque unité un arbitre et on considère un jeton circulant dans l'anneau de ces arbitres. Le jeton se déplace à chaque cycle d'horloge. Si le client qui possède le jeton demande la ressourceΓelle lui est immédiatement attribuée.

L'observateur d'une cellule est donné figure 11.3. Cette cellule est répétée k foisΓcomme le montre la figure 11.4. Chaque cellule a une entrée *request* et une sortie *acknowledge*. La sortie *grant* de la cellule i est passée à la cellule $i+1$Γet indique qu'aucun client d'indice plus petit ou égal à i n'est demandeur. En conséquenceΓune cellule ne peut émettre un signal *acknowledge* que s'il reçoit le signal *grant*. Chaque cellule contient un registre T (pour "token") qui vaut vrai lorsque le jeton est présent. Elle contient de plus un registre W (pour "waiting") qui est mis à vrai lorsque l'entrée *request* est vraie et que le jeton est présent. Il reste à vrai tant que la requête persiste et jusqu'à ce que le jeton revienne. A ce momentΓles signaux *override* et *acknowledge* de la cellule sont émis. Le signal *override* se propage parmi les cellules suivantesΓjusqu'à ce qu'il annule l'entrée *grant* de la cellule 0Γet ainsi empêche les autres cellules d'utiliser la ressource à ce moment. Le circuit est initialisé de manière à ce que tous les registres W soient à 0 et exactement un registre T soit à 1.

11.3.2 Calcul d'un invariant

On cherche à montrer que l'exclusion mutuelle est assurée par cet arbitreΓen utilisant la méthode par induction décrite dans les chapitres précédents. Nous avons donc à trouver un invariant du réseau qui satisfait cette propriété.

11.3.2.1 Calcul automatique

Le méthode de calcul automatique d'une approximation inférieure du plus grand point fixe ne converge pas dans cet exemple. La mémoire est saturée après 15 pas de calcul en arrièreΓsans qu'aucun élargissement ne réussisse à faire converger la suite d'invariants.

11.3.2.2 Extrapolation manuelle

Selon l'idée de la section 11.1.1Γnous allons essayer de déterminer une cellule P' dont le comportement est plus grand que celui de P ($P \preceq P'$)Γet tel que le nouveau réseau ainsi créé satisfasse encore la propriété.

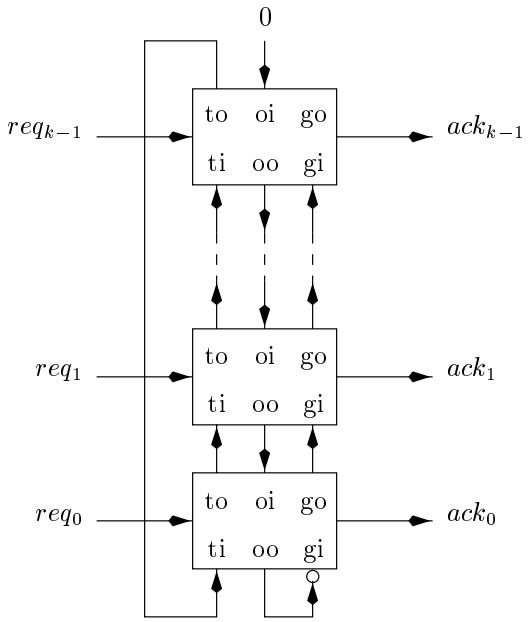


FIG. 11.4 – Configuration de l'arbitre de McMillan

Considérons l'automate d'une cellule figure 11.5 : nous ne nous intéressons qu'aux signaux relatifs au passage du jeton et aux requêtes; les signaux oo oi go gi ack ne sont donc pas représentés afin de simplifier le schéma. Essayons de comprendre la définition de chaque état de cet automate.

État 0 : C'est l'état initial la cellule ne possède pas le jeton et ne réclame pas la ressource. Les deux registres T et W sont positionnés à faux. Si la cellule ne reçoit pas de jeton elle reste dans le même état sinon elle passe dans l'état 2.

État 1 : La cellule ne possède pas le jeton mais demande la ressource les registres T et W sont positionnés respectivement à 0 et 1. Lorsqu'elle ne reçoit pas de jeton elle reste dans le même état jusqu'à ce qu'elle annule sa requête. Elle retourne alors dans l'état 0. Si elle reçoit le jeton soit elle maintient sa requête et passe dans l'état 3 soit elle ne la maintient pas et passe dans l'état 2.

État 2 : La cellule possède le jeton et ne demande pas la ressource les registres T et W sont positionnés respectivement à 1 et 0. Si la cellule libère le jeton soit elle demande la ressource et passe dans l'état 1 soit elle ne la demande pas et retourne dans l'état 0. La cellule ne peut garder le jeton que si elle en reçoit un nouveau. Si elle ne demande pas la ressource elle reste dans le même état sinon elle passe dans l'état 3.

État 3 : La cellule possède le jeton et demande la ressource les deux registres T et W sont positionnés à 1. Si la cellule libère le jeton soit elle demande la ressource et passe dans l'état 1 soit elle ne la demande pas et retourne dans l'état 0. La cellule ne peut garder le jeton que si elle en reçoit un nouveau. Si elle demande la ressource elle reste dans le même état sinon elle passe dans l'état 2.

Intuitivement la non convergence du calcul du plus grand point fixe vient du fait que pour k processus le jeton a besoin d'exactement k cycles d'horloge pour effectuer le tour du réseau. A chaque pas de calcul l'invariant calculé contient cette propriété ce qui empêche évidemment la convergence (puisque $k = k + 1$ uniquement si $k = \infty$). Pour forcer l'invariant à contenir

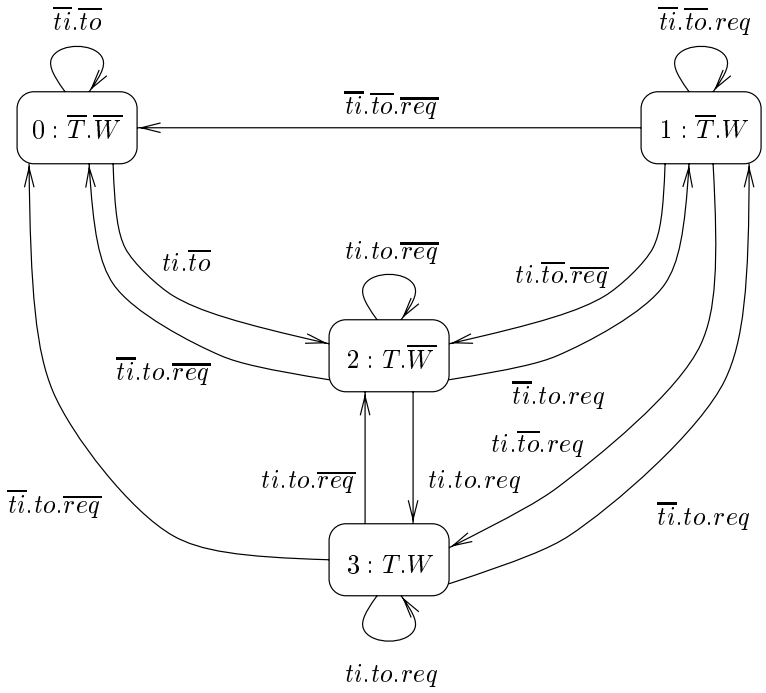


FIG. 11.5 – Automate décrivant une cellule. Les signaux ooΓoiΓgoΓgiΓack ne sont pas représentés

simplement la propriété que le jeton met un temps quelconque pour parcourir le réseau (temps qui ne dépend surtout pas de k)Γnous avons eu l'idée de modifier la cellule de façon à ce qu'elle puisque garder le jeton un temps quelconque.

Comme on l'a vuΓles états 2 et 3 de l'automate de la figure 11.5 sont ceux où la cellule possède la ressource. Considérons les transitions permettant de rester dans l'ensemble d'états $\{2, 3\}$Γi.e.Γ les transitions $2 \to 2$Γ$2 \to 3$Γ$3 \to 2$ et $3 \to 3$Γ

$$
\begin{array}{lcl}
2 & \xrightarrow{\overline{req}.ti.to.\overline{ack}.(oo=oi).(go=gi)} & 2 \\
2 & \xrightarrow{req.ti.\overline{go}.to.(oo=oi).(gi=ack)} & 3 \\
3 & \xrightarrow{req.ti.\overline{go}.to.ack.oo} & 3 \\
3 & \xrightarrow{\overline{req}.ti.to.\overline{ack}.oo.(go=gi)} & 2
\end{array}
$$

Si on ne garde que les signaux relatifs au passage du jetonΓsoit ti et toΓil nous reste

$$
\begin{array}{lcl}
2 & \xrightarrow{ti.to} & 2 \\
2 & \xrightarrow{ti.to} & 3 \\
3 & \xrightarrow{ti.to} & 3 \\
3 & \xrightarrow{ti.to} & 2
\end{array}
$$

En d'autres termesΓpuisque la cellule est obligée de transmettre le jeton qu'elle possèdeΓelle ne reste dans un état où elle a le jeton que si elle en reçoit un autre. Pour permettre à la cellule de garder le jeton plus d'un coup d'horlogeΓil suffit de lui permettre de ne pas émettre le jeton lorsqu'elle n'en reçoit pasΓi.e.Γde remplacer dans les transitions précédentes les étiquettes $ti.to$

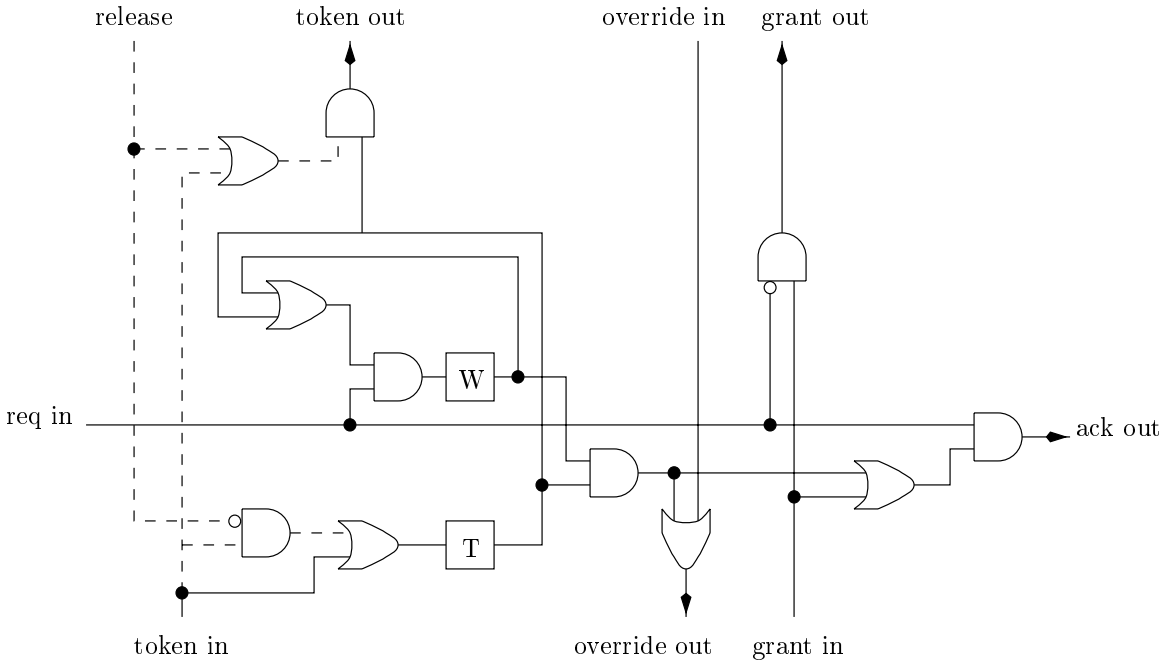


FIG. 11.6 – Une cellule élargie de l'arbitre de McMillan

par $ti = to$. On obtient ainsi

$$
\begin{array}{lcl}
2 & \xrightarrow{\overline{req}.(ti{=}to).\overline{ack}.(oo{=}oi).(go{=}gi)} & 2 \\
2 & \xrightarrow{req.(ti{=}to).\overline{go}.(oo{=}oi).(gi{=}ack)} & 3 \\
3 & \xrightarrow{req.(ti{=}to).\overline{go}.ack.oo} & 3 \\
3 & \xrightarrow{\overline{req}.(ti{=}to).\overline{ack}.oo.(go{=}gi)} & 2
\end{array}
$$

L'automate obtenu est non déterministe. Dans les états 2 et 3Γlorsqu'il ne reçoit pas de jetonΓil peut soit émettre un jetonΓsoit le garder. La figure 11.6 montre le schéma d'une cellule modifiée. Une nouvelle entrée *release* indique si la cellule rend le jeton (lorsqu'elle le possède) ou si elle le garde. Les liaisons en pointillé ont été rajoutées par rapport à la figure 11.3 : lorsque *release* est fixé à 1Γon retrouve le précédent schémaΓce qui confirme que la nouvelle cellule a un comportement plus grand que la précédente.

Pour le nouveau réseauΓle plus grand point fixe est calculé exactement (sans extrapolation) en 2 pas. Cet invariant vérifiant la propriété d'exclusion mutuelleΓil permet de prouver la même propriété sur le réseau initial.

11.4 Utilisation d'un démonstrateur de théorème

La technique présentée dans ce document ne peut s'appliquer qu'au cas booléenΓi.e.Γchaque processus a un nombre fini d'états (partie contrôle et partie donnée)Γet les communications entre processus s'effectuent par l'intermédiaire de signaux booléens.

Dans certains casΓla propriété qui nous intéresse ne dépend qu'indirectement de la partie donnée. C'est le cas dans l'exemple que nous présentons dans cette section. Nous allons considérer un système qui assure l'exclusion mutuelle à l'aide de signaux booléens (en faitΓpar passage d'un jeton)Γet une règle de priorité (à l'aide de signaux entiers). Si l'on abstrait de manière brutale les signaux entiersΓl'exclusion mutuelle est violée car il peut se créer plusieurs jetons dans

le réseau. Il est donc nécessaire de faire une abstraction plus fineΓet en particulier de générer certaines propriétés sur l'émission des jetonsΓdues aux contraintes linéaires. Pour celaΓnous allons utiliser le démonstrateur automatique ("theorem prover") PVS [OSR93] et utiliser des techniques développées par Graf et Saïdi [Saï96ΓGS96]. Le même résultat aurait pu être obtenu en utilisant des polyèdres [HPR94] (par exemple dans l'outil POLKA [HMP95]).

Le système considéré comprend n processus qui se partagent une ressource en exclusion mutuelle. Chaque processus peut demander la ressource avec une certaine priorité req (pour "request"). Si $req = 0$Γle processus ne demande pas la ressource. Si req est petitΓil demande la ressource avec une priorité faible; il n'est serviΓque si les autres processus ne demandent la ressource qu'avec une priorité plus faible. Si req est grandΓle processus demande fortement la ressource. Un jeton peut circuler dans le réseau. Il se déplace en fonction des requêtes des processus. Chaque processus possède donc 2 variables internes :

- req (pour "request") qui indique la priorité avec laquelle le jeton demande la ressource. Cette valeurΓune fois strictement positive (le processus demande la ressource)Γne peut plus être modifiéeΓjusqu'à ce que le processus reçoive le jeton.

- tk (pour "token") qui vaut vrai lorsque le processus possède le jeton.

Un processus peut recevoir 4 signaux et en émettre 5.

- req_{in}^{left} est la valeur entière maximale des requêtes des processus situés à gauche du processus recevant le signal.

- req_{in}^{right} est la valeur entière maximale des requêtes des processus situés à droite du processus recevant le signal.

- tk_{in}^{left} est vrai lorsque le processus reçoit le jeton par la gauche.

- tk_{in}^{right} est vrai lorsque le processus reçoit le jeton par la droite.

- req_{out}^{left} est une valeur entière calculée par le processus et émise vers sa gauche. Elle est égale au maximum de sa requêteΓet des requêtes qu'il reçoit des autres processus.

- req_{out}^{right} est une valeur entière calculée par le processus et émise vers sa droite. Elle est égale à req_{out}^{left}.

- tk_{out}^{left} est vrai lorsque le processus émet le jeton vers la gauche. C'est le cas lorsque la requête reçue par la gauche est supérieur à celle du processus et celle reçue par la droite.

- tk_{out}^{right} est vrai lorsque le processus émet le jeton vers la droite. C'est le cas lorsque la requête reçue par la droite est supérieur à celle du processus et celle reçue par la gauche.

- use est vrai lorsque le processus utilise la ressource.

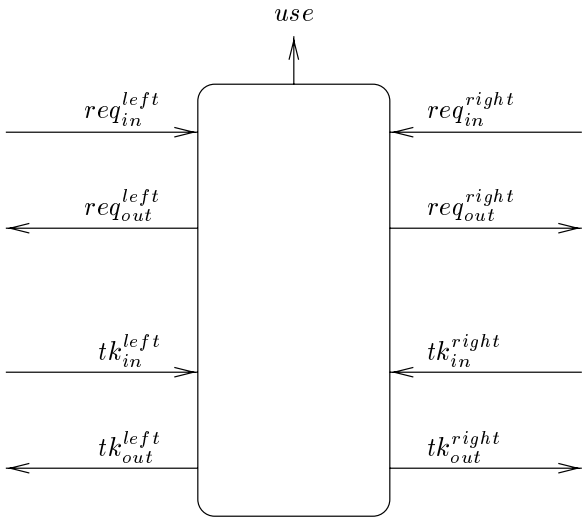


FIG. 11.7 – Communication d'un processus avec ses voisin

(voir figure 11.7). Un processus est décrit par les équations suivantes :

$$
\begin{array}{lcl}
\textbf{next}\ tk & = & \textbf{next}\ \left(\ \overline{tk_{out}^{left}} \wedge \overline{tk_{out}^{right}}\ \right) \wedge (tk \vee tk_{in}^{left} \vee tk_{in}^{right}); \\
\textbf{next}\ use & = & \left(\ \textbf{next}\ tk\ \right) \wedge (req > 0) \wedge tk; \\
\textbf{next}\ req & = & if\ req = 0\ then\ req^{new} \\
 & & else\ if\ use\ then\ req^{new} \\
 & & else\ req; \\
\textbf{next}\ tk_{out}^{left} & = & tk \wedge (req_{in}^{left} > req) \wedge (req_{in}^{left} > req_{in}^{right}); \\
\textbf{next}\ tk_{out}^{right} & = & tk \wedge (req_{in}^{right} > req) \wedge (req_{in}^{right} > req_{in}^{left}); \\
\textbf{next}\ req_{out}^{left} & = & if\ (req > req_{in}^{left}) \wedge (req > req_{in}^{right})\ then\ req \\
 & & else\ if\ (req_{in}^{left} > req_{in}^{right})\ then\ req_{in}^{left} \\
 & & else\ req_{in}^{right}; \\
\textbf{next}\ req_{out}^{right} & = & req_{out}^{left};
\end{array}
$$

Le compilateur LUSTRE peut traduire ce programme vers un automate booléen présenté figure 11.8. Les variables entières ont été abstraites. Chaque test sur les entiers provoque donc un choix non déterministe qui est modélisé par une variable d'entrée booléenne : cette variable est vraie lorsque le test est évalué à vrai et fausse dans le cas contraire. Le rôle de ces variables se rapproche des "oracles" introduits section 4.2.1 page 43 pour modéliser le non déterminisme. Dans notre cas les valeurs de ces variables étaient aléatoires. Dans le cas des automates booléens générés par le compilateur LUSTRE il s'agit plutôt de valeurs inconnues (i.e. qui dépendent de variables abstraites). Dans cet exemple 5 variables ont été introduites équivalentes aux conditions

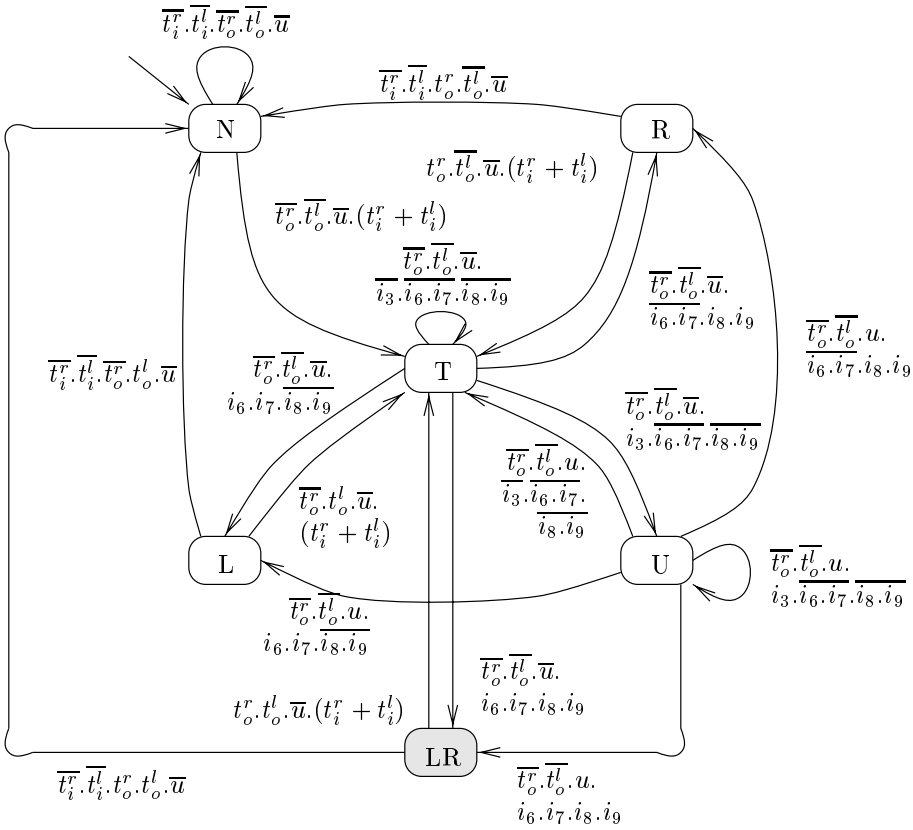


FIG. 11.8 – Le processus de l'anneau à priorité Γabstrait par le compilateur LUSTRE. Les signaux tk_{in}^{left} Γ tk_{in}^{right} Γ tk_{out}^{left} et tk_{out}^{right} Γsont abrégés respectivement t_i^l Γ t_i^r Γ t_o^l et t_o^r. Les tests sur les entiers sont remplacés par les oracles i_3 Γ i_6 Γ i_7 Γ i_8 et i_9

suivantes :

$$
\begin{aligned}
i_3 &= req > 0;\\
i_6 &= req_{in}^{left} > req;\\
i_7 &= req_{in}^{left} > req_{in}^{right};\\
i_8 &= req_{in}^{right} > req;\\
i_9 &= req_{in}^{right} > req_{in}^{right};
\end{aligned}
$$

L'interprétation de l'automate de la figure 11.8 est la suivante :

État N : ("Neutral") Le processus n'a pas le jeton et n'utilise pas la ressource.

État T : ("Token") Le processus possède le jeton. A partir de cet état Γil doit décider ce qu'il va en faire.

État U : ("Use") Le processus a la plus forte priorité. Il décide d'utiliser la ressource.

État L : ("Left") La plus forte priorité vient de la gauche. Le processus s'apprête donc à émettre le jeton vers la gauche.

État R : ("Right") La plus forte priorité vient de la droite. Le processus s'apprête donc à émettre le jeton vers la droite.

État LR : ("Left/Right") La plus forte priorité vient à la fois de la gauche et de la droite. Le processus s'apprête donc à émettre le jeton des deux cotés à la fois.

L'état **LR** (grisé sur le schéma) n'est normalement pas accessible. En effetΓpour y parvenirΓla condition $i_7 \wedge i_9 = (req_{in}^{left} > req_{in}^{right}) \wedge (req_{in}^{right} > req_{in}^{right})$ doit être vérifiée: cette condition est trivialement fausseΓmais le compilateur LUSTRE ne s'en aperçoit pas. AinsiΓle processus peut émettre deux jetons à la foisΓce qui conduit à la violation de la propriété d'exclusion mutuelle.

L'outil INVARIANT CHECKER développé par Saïdi permet de construire un graphe abstrait de ce système et trouve automatiquement que l'état **LR** n'est pas accessible. Le même résultat aurait pu être obtenu en utilisant des polyèdres (par exemple dans l'outil POLKA [HMP95]). Notre méthode permet alors de prouver automatiquement la propriété d'exclusion mutuelle grâce à la construction d'un invariant adéquat pour le système abstrait.

Troisième partie

Implémentation

Chapitre 12

BANG : A Boolean Automaton Network Grammar checker

Cette partie décrit l'implémentation des résultats des parties I et II.

A l'origine, la synthèse de processus avait été implémentée à partir de l'outil BAC [Hal94] de Halbwachs. BAC (pour "Boolean Automata Checker") permet de vérifier des propriétés de sûreté sur les circuits booléens, il implémente, entre autre, les algorithmes de la post et de la précondition, de la résolution des boucles dans la définition des variables locales et le calcul de l'assertion extensible maximale. Les circuits booléens utilisés peuvent être automatiquement produit par le compilateur LUSTRE [HCRP91] (la plupart des exemples décrits dans ce document ont ainsi été écrits en LUSTRE). Dans un second temps, les outils nécessaires aux calculs d'invariants ont été implémentés. Notre logiciel est alors devenu

BANG (pour "Boolean Automata Network Grammars checker" [Les97a]).

BANG est un outil prototype. De ce fait, il évolue au fil des expériences et de la conception de nouveaux algorithmes.

12.1 Introduction

12.1.1 Algorithmes implémentés

BANG implémente les algorithmes suivants :

- La synthèse de processus : à partir d'un observateur implémenté par un circuit booléen, BANG synthétise les BDD implémentant les ensembles de configurations $\mathcal{I}_P$ et F^l_P (voir chapitre 3, page 33). Il détermine ensuite l'ensemble des solutions déterministes à l'aide d'oracles (voir chapitre 4, page 43). L'exécution de ces solutions peut être simulée.
- La synthèse d'invariant : BANG peut lire la description d'une grammaire de réseau, et calculer un invariant en utilisant les heuristiques vues dans la partie II.
- Des calculs généraux sur les automates : BANG reprend les algorithmes symboliques de BAC [Hal94] (résolution des boucles dans la définition des variables locales, pré et post-condition) en les optimisant (voir chapitre 14); de nouveaux algorithmes sont implémentés, aussi bien en symbolique qu'en énumératif (composition, minimisation, abstraction, élargissement).

  Un mode en ligne permet d'appliquer ces algorithmes à la demande.

12.1.2 Choix de l'implémentation

Deux implémentations ont été mises en oeuvre. Dans BAC les automates booléens étaient décrits en symbolique. Le calcul des abstractions existentielle et universelle n'étant pas suffisamment efficace dans ce format (voir chapitre 13 page 151) une implémentation énumérative a été entreprise. Cette implémentation est particulièrement efficace sur les petits exemples. Il semble pourtant que dans l'avenir il faille inventer des algorithmes symboliques réellement efficaces de façon à pouvoir traiter des problèmes de taille conséquente.

12.1.2.1 Implémentation énumérative

Un essai a été effectué pour synthétiser des invariants à l'aide d'ALDEBARAN [FGK+96] et éviter ainsi un long travail d'implémentation. Bien que les algorithmes implémentés dans ALDEBARAN soient plus efficaces que ceux implémentés dans BANG (du moins pour ce qui est des algorithmes énumératifs) son utilisation n'a pas été possible à cause du nombre important de symboles de signaux. En effet pour un exemple comme l'arbitre de McMillan (voir section 11.3 page 128) le nombre de signaux utiles est de 9. La définition d'une fonction de composition d'arité 2 utilise donc $3*9 = 27$ signaux (9 signaux 9 renommages en ' et 9 renommages en ") soit $2^{27} = 1.34 \times 10^8$ événements. Il est en fait indispensable de développer des algorithmes spécifiques utilisant des BDD pour coder les événements. Une future version d'ALDEBARAN allant dans ce sens sera peut-être utilisable.

Dans le futur les *guided tree automata* développés par [BMK+96] pourraient également fournir des outils particulièrement efficaces pour calculer des invariants.

Pour le moment il était de toute manière nécessaire de développer un modèle d'automates énumératifs pour implémenter certains algorithmes spécifiques tels les extrapolations de plus petits points fixes (voir chapitre 7 page 71) de plus grands points fixes (voir chapitre 8 page 87) ou de décomposition d'invariant (voir chapitre 9 page 99).

12.1.2.2 Implémentation symbolique

L'implémentation symbolique des automates booléens est réalisée par les circuits booléens (voir section 1.1.4 page 17) ou chaque fonction de $\mathbb{B}^n \to \mathbb{B}$ est définie par un BDD.

L'implémentation à l'aide d'une relation a également été envisagée. Elle consiste à définir une variable booléenne par variable d'état ainsi qu'une autre variable booléenne pour la version prime de chaque variable d'état. La fonction de transition est alors décrite par une seule formule booléenne des variables d'état des variables d'état prime et des entrées. L'intérêt évident de ce codage est de permettre des fonctions de transition non déterministes. L'inconvénient majeure est la taille du BDD codant la relation.

En effet pour un circuit booléen à n variables d'état et m entrées codé par une fonction il faut stocker n BDD à $n+k$ variables soit utiliser dans le pire des cas une mémoire de $n.2^{n+k}$. Pour ce même automate codé par une relation il faut stocker un BDD à $2n+k$ variables soit utiliser dans le pire des cas une mémoire de 2^{2n+k}. Dans le cas des automates de grande taille l'utilisation d'une relation semble donc à proscrire.

Bibliothèque de BDD: L'outil BANG a utilisé dans un premier temps la bibliothèque de BDD décrite dans [Rat92]. Par la suite il a été récrit avec la bibliothèque TIGER [CMT93]. Cette bibliothèque implémente tous les opérateurs booléens classiques: $=, \neq, \vee, \wedge, \forall, \exists$ etc ... . Elle possède également les deux opérateurs "restrict" et "constrain" (voir section 14.1.2 page 159).

La bibliothèque TIGER permet également le réordonnancement automatique des variables. La taille du BDD d'une formule dépend en effet fortement de l'ordre des variables. Ainsi le BDD de $(x_0 = x_1) \wedge (x_1 = x_2) \wedge \ldots \wedge (x_{n-2} = x_{n-1})$ est de taille $O(n)$ pour l'ordre $x_0 < x_1 < \ldots < x_{n-1}$ mais de taille $O(2^n)$ pour l'ordre $x_1 < x_{n-1} < x_2 < x_{n-2} < \ldots$. La technique de réordonnancement automatique consiste à modifier dynamiquement l'ordre des variables des

BDD de façon à minimiser leur taille (voir [Rau96] pour plus de détails). Nous avons vu dans les différentes expérimentations de cet ouvrage que cette technique était particulièrement efficace.

Dans ce chapitre nous décrivons la syntaxe des automates booléens (symboliques et énumératifs) ainsi que les commandes de BANG. Le chapitre 13 décrit des algorithmes d'abstraction de signaux. Dans le chapitre 14 nous comparerons différents algorithmes de calcul de la post et de la précondition.

12.2 Syntaxe des automates booléens

Il existe 4 manières de décrire un automate booléen: de façon symbolique ou énumérative de façon compactée ou non compactée. Ces descriptions correspondent aux utilisations suivantes:

Type	Extension	Utilisation
Symbolique	.ba	Automate E/S et observateur
Symbolique compacté	.cba	Automate E/S et observateur
Énumératif	.aut	Observateur
Énumératif compacté	.caut	Observateur

Dans cette section les 2 syntaxes non compactées vont être présentées.

12.2.1 Automates booléens symboliques

La structure générale de la description d'un automate booléen symbolique est la suivante:

⟨Symbolic_boolean_automaton⟩ ::= ⟨declarations⟩ ⟨initial⟩ ⟨equations⟩
[⟨assertion⟩] [⟨invariant⟩] [⟨finals⟩]

5 classes de variables peuvent apparaître dans un automate booléen: les variables d'état les variables d'entrée les variables oracles (voir chapitre 4 page 43) les variables locales et les variables de sortie. Chacune de ces quatre dernières classes peut être vide. Chaque classe existante est déclarée par un mot clé approprié suivi par une liste d'identificateurs de variables et terminé par un point virgule. Les identificateurs suivent la syntaxe du C.

⟨declarations⟩ ::= ⟨state_var_declarations⟩ [⟨input_declarations⟩]
[⟨oracle_declarations⟩] [⟨local_declarations⟩]
[⟨output_declarations⟩]
⟨state_var_declarations⟩ ::= **states** ⟨ident_list⟩ “;”
⟨input_declarations⟩ ::= **inputs** ⟨ident_list⟩ “;”
⟨oracle_declarations⟩ ::= **oracles** ⟨ident_list⟩ “;”
⟨local_declarations⟩ ::= **localstates** ⟨ident_list⟩ “;”
⟨output_declarations⟩ ::= **outputs** ⟨ident_list⟩ “;”
⟨ident_list⟩ ::= ⟨ident⟩
| ⟨ident_list⟩ “,” ⟨ident⟩

Chaque variable qui n'est pas une entrée ou un oracle est définie par une équation. Il y a deux types d'équations: les *transitions* définissent les valeurs des variables d'état dans l'état suivant; l'identificateur de la partie gauche est suivi d'un prime; les *définitions* définissent les variables locales et de sortie:

⟨equations⟩ ::= **transitions** ⟨transition_list⟩ [**definitions** ⟨definition_list⟩]
⟨transition_list⟩ ::= ⟨transition⟩
| ⟨transition_list⟩ ⟨transition⟩
⟨transition⟩ ::= ⟨ident⟩"'" "=" ⟨expression⟩ ";"
⟨definition_list⟩ ::= ⟨definition⟩
| ⟨definition_list⟩ ⟨definition⟩
⟨definition⟩ ::= ⟨ident⟩ "=" ⟨expression⟩ ";"

Les expressions sont des expressions booléennes constituées de constantes d'identificateurs et d'opérateurs.

⟨expression⟩ ::= "0"
| "1"
| ⟨ident⟩
| "(" ⟨expression⟩ ")"
| **not** ⟨expression⟩
| ⟨expression⟩ ⟨bin_op⟩ ⟨expression⟩
| "#" "(" ⟨expression_list⟩ ")"
| **if** ⟨expression⟩ **then** ⟨expression⟩ **else** ⟨expression⟩ **fi**
⟨bin_op⟩ ::= **or**
| **and**
| **xor**
| **eq**

"eq" est l'opérateur "égal" "xor" est le "ou exclusif" ou l'opérateur "non égal". "#" est l'opérateur "d'exclusion" : il retourne "true" si et seulement si au plus un de ses arguments est vrai. La priorité des opérateurs est standard : "not" a une priorité plus forte que "eq" et "xor" qui ont une priorité plus forte que "and" qui a une priorité plus forte que "or". Les identificateurs primés doivent être des variables d'état.

Pour rendre plus compactes les représentations des expressions booléennes une syntaxe compactée équivalente a été définie :

not ≡ ! ≡ −
or ≡ +
and ≡ .
eq ≡ =
if x_1 **then** x_2 **else** x_3 ≡ (x_1 ? x_2 : x_3)

La syntaxe utilisée pour **not** **or** et **and** provient de l'outil BDDC (pour "BDD-Calculator") de Raymond.

L'ensemble des états initiaux $Init$ l'assertion $\mathcal{A}$ l'invariant $\mathcal{I}$ et l'ensemble des états finals F sont définis par une expression précédée par un mot clé approprié :

⟨initial⟩ ::= **initial** ⟨expression⟩ ";"
⟨assertion⟩ ::= **assertion** ⟨expression⟩ ";"
⟨invariant⟩ ::= **invariant** ⟨expression⟩ ";"
⟨finals⟩ ::= **finals** ⟨expression⟩ ";"

Habituellement les fichiers décrivant des automates booléens symboliques ont l'extension ".ba". Il existe une forme compactée des automates booléens symboliques utilisée de façon interne par BANG. Elle est décrite par des fichiers avec l'extension ".cba".

Exemple 12.1 :

Cet exemple décrit le comportement d'un processus de l'exemple du jeton circulant (voir section 10.1 page 109).

states
token;
inputs
inΓreq;
outputs
useΓout;
initial
not token;
transitions
token' = in **or** (token **and not** out);
definitions
use = req **and** token;
out = (**not** req) **and** token;

Un observateur de ce processus E/S peut être implémenté par le circuit booléen suivant :

states
token;
inputs
inΓreqΓuseΓout;
initial
not token;
transitions
token' = in **or** (token **and not** out);
invariant
(use = (req **and** token)) **and**
(out = (**not** req) **and** token);

□

12.2.2 Automates booléens énumératifs

La structure générale de la description d'un automate booléen énumératif est la suivante :

⟨Enumerative_boolean_automaton⟩	::=	⟨declaration⟩ ⟨transitions⟩
⟨declaration⟩	::=	**des** "(" ⟨init⟩ "**,**"⟨trans⟩ "**,**" ⟨states⟩ ")"
⟨transitions⟩	::=	ϵ
	|	⟨transition⟩ ⟨transitions⟩
⟨transition⟩	::=	"(" ⟨source⟩ "**,**" "**"** ⟨expression⟩ "**"** "**,**" ⟨target⟩ ")"

⟨init⟩ représente l'état initial de l'automate : actuellementΓil doit obligatoirement être égal à 0. ⟨trans⟩ et ⟨states⟩ représentent respectivement le nombre de transitions et le nombre d'états de l'automate. ⟨expression⟩ est une expression booléenne représentant la garde de la transition.

HabituellementΓles fichiers décrivant des automates booléens énumératifs ont l'extension ".aut"Γ i.e.Γla même que pour les fichiers utilisés par ALDEBARAN [FGK+96]. La forme utilisée par BANG est néanmoins plus compacte car chaque transition est étiquetée par une garde au lieu d'un événement. La commande "-ald" génère un fichier compatible avec ALDEBARAN. AinsiΓsi l'ensemble des signaux de l'automate est $\{a, b\}$Γla garde $\overline{a}$ est expansée en $\{\ \{\overline{a}.b\}, \{\overline{a}.\overline{b}\}\ \}$.

Il existe une forme compactée des automates booléens énumératifsΓutilisée de façon interne par BANG. Elle est décrite par des fichiers avec l'extension ".caut".

Exemple 12.2 :

L'automate énumératif suivant décrit le même observateur que celui de l'exemple 12.1 :

```
des (0Γ4Γ2)
   (   0   Γ        "!in.!out.!use"        Γ   0   )
   (   0   Γ        "in.!out.!use"         Γ   1   )
   (   1   Γ        "!in.out.!use"         Γ   0   )
   (   1   Γ   "(out?!use.in:use=req)"     Γ   1   )
```

□

12.3 Commandes en ligne

BANG possède un mode en ligneΓqu'il est possible de lancer en tapant la commande :

bang -

L'utilisateur peut alors lancer des commandes de calcul sur les automates booléens. Ces commandes se divisent en différents sous-groupes.

12.3.1 Commandes générales

help :

Affiche la liste des commandes.

end :

Termine l'exécution de BANG. Cette commande est équivalente à un *End Of File*.

list :

Affiche la liste des automates actuellement en mémoire.

ident :

Affiche la liste de tous les identificateurs actuellement utilisés. Ces identificateurs peuvent représenter des automates booléens ou des variables booléennes (signaux d'entrée/sortieΓou variables d'état). Cette commande a été définie dans une optique de débogage.

bdd :

Affiche l'ensemble des étiquettes de BDD actuellement utilisées. Cette commande a été définie dans une optique de débogage.

reset :

Réinitialise le système. Efface tous les automates booléens et réinitialise le "BDD-manager".

12.3.2 Commandes d'affectation

let ⟨**ident**⟩ "=" "\"" ⟨**file_name**⟩ "\"" ["["⟨**renames**⟩ "]" ";"] :

Charge un automate booléen à partir du fichier nommé ⟨file_name⟩Γet l'affecte à l'identificateur ⟨ident⟩. Les noms de fichier décrivant des automates symboliques compactés doivent avoir l'extension ".cba". Les autres formats sont automatiquement reconnus par l'analyseur syntaxiqueΓquelle que soit leur extension. Les variables d'entrée et de sortie peuvent être renommées par l'expression ⟨renames⟩. Cette expression suit la syntaxe suivante :

```
⟨renames⟩   ::=   ⟨rename⟩
              |   ⟨renames⟩ "," ⟨rename⟩
 ⟨rename⟩   ::=   ⟨ident1⟩ ">" ⟨ident2⟩
```

⟨ident1⟩ doit être l'identificateur d'une variable d'entrée ou de sortie. ⟨ident2⟩ doit être un identificateur non utilisé dans la description de l'automate booléen. La variable de nom ⟨ident1⟩ est alors renommée en ⟨ident2⟩.

save ⟨**ident**⟩ "**"** ⟨**file_name**⟩ "**"** [**"["** ⟨**renames**⟩ **"]"** ";"] :

Sauvegarde l'automate ⟨ident⟩ dans le fichier de nom ⟨file_name⟩ en renommant éventuellement les signaux d'entrée/sortie. Le format de sauvegarde dépend de l'extension du nom de fichier (".ba"Γ".cba"Γ".aut" ou ".caut").

let ⟨**ident**⟩ **"="** ⟨**expression**⟩ [**"["** ⟨**renames**⟩ **"]"** ";"] :

Calcule l'expression d'automates ⟨expression⟩Γet l'affecte à l'identificateur ⟨ident⟩ avec l'éventuel renommage ⟨renames⟩. Une expression d'automates suit la syntaxe suivante :

⟨expression⟩ ::= ⟨expression⟩ **and** ⟨expression⟩
| ⟨expression⟩ **or** ⟨expression⟩
| "(" ⟨expression⟩ ")"
| ⟨ident⟩

Les opérateurs **and** et **or** calculent respectivement l'intersection et l'union de langages.

let ⟨**ident**⟩ **"="** **forall "("** ⟨**ident_list**⟩ **")"** ⟨**expression**⟩ **";"** :

Calcule l'abstraction universelle de l'expression ⟨expression⟩ par les variables ⟨ident_list⟩ et l'affecte à l'identificateur ⟨ident⟩.

let ⟨**ident**⟩ **"="** **exist "("** ⟨**ident_list**⟩ **")"** ⟨**expression**⟩ **";"** :

Calcule l'abstraction existentielle de l'expression ⟨expression⟩ par les variables ⟨ident_list⟩ et l'affecte à l'identificateur ⟨ident⟩.

minimize "(" ⟨**ident**⟩ **")" ";"** :

Minimise l'automate booléen ⟨ident⟩ selon l'équivalence de traces.

let ⟨**ident**⟩ **"="** **minimize "("** ⟨**expression**⟩ **")" ";"** :

Calcule l'expression d'automates ⟨expression⟩Γla minimise et l'affecte à l'identificateur ⟨ident⟩.

destroy ⟨**ident**⟩ **";"** :

Libère la mémoire utilisée par l'automate booléen ⟨ident⟩.

12.3.3 Vérification

verify ⟨**expression**⟩ **";"** :

Calcule l'expression d'automates ⟨expression⟩Γet vérifie que le signal d'alarme α n'est jamais émisΓi.e.Γque la propriété spécifié par l'invariant est satisfaite. L'algorithme utilisé est un algorithme de calcul en avant des états accessibles.

backward ⟨**ident**⟩ **";"** :

Vérifie que l'automate ⟨ident⟩ implémente le langage X^∞. L'algorithme utilisé est un algorithme en arrière; il n'est implémenté que symboliquement (i.e.Γpour les circuits booléens)Γ et est en général moins efficace que le calcul en avant.

impl "(" ⟨**expression1**⟩ **","** ⟨**expression**⟩ **")" ";"** :

Calcule les expressions ⟨expression1⟩ et ⟨expression2⟩ et vérifie que le langage reconnu par la première est inclus dans le langage reconnu par la seconde.

let ⟨**ident**⟩ **"="** **shrink "("** ⟨**ident1**⟩ **","** ⟨**ident2**⟩ [**","** ⟨**d1**⟩ **","** ⟨**d2**⟩] **")" ";"** :

Calcule l'élargissement (voir section 8.3Γpage 90) de l'automate ⟨ident1⟩ par l'automate ⟨ident2⟩Γafin d'abstraire une approximation du plus grand point fixe d'une équation. ⟨d1⟩ et ⟨d2⟩ sont les valeurs des deux paramètres d_1 et d_2. Cet élargissement nécessite que le langage reconnu par l'automate ⟨ident1⟩ soit inclus dans celui reconnu par le langage ⟨ident2⟩.

Exemple 12.3 :

L'exemple suivant charge un fichier "test.caut"Γcontenant un automate énumératif compactéΓle minimiseΓet le sauvegarde sous forme de circuit booléen non compacté :

bang -

bang: Boolean Automaton Network Grammar checker
Version for SunOS 5.5 Generic (Wed Apr 9 11:23:31 MET DST 1997)
Boolean Automaton Calculator:
> **let x="test.caut";**
> **minimize(x);**
> **save x "test.ba";**
> **end**

□

12.3.4 Grammaire de réseau

Les grammaires de réseau sont un formalisme permettant de construire des réseaux infinis de processus. La structure générale de la description d'une grammaire de réseau d'automates booléens est la suivante :

⟨Network_grammar⟩ ::= [⟨inputs⟩] [⟨outputs⟩] ⟨terminals⟩ ⟨non-terminals⟩ ⟨start⟩
⟨composition-rules⟩ ⟨property⟩ ⟨production_rules⟩

Les sections ⟨inputs⟩ et ⟨outputs⟩ déclarent respectivement les variables de communication d'entrée et de sortie du réseau.

⟨inputs⟩ ::= **inputs** ⟨ident_list⟩ ";"
⟨outputs⟩ ::= **outputs** ⟨ident_list⟩ ";"
⟨ident_list⟩ ::= ⟨ident⟩
| ⟨ident⟩ "," ⟨ident_list⟩

Les sections ⟨terminals⟩ et ⟨non-terminals⟩ déclarent respectivement les symboles terminaux et non terminaux utilisés. A chaque symbole terminal est associé un automate booléenΓqui doit être défini dans le fichier ⟨file_name⟩.

⟨terminals⟩ ::= **terminals** ⟨terminal_list⟩
⟨terminal_list⟩ ::= ⟨terminal⟩
| ⟨terminal⟩ ⟨terminal_list⟩
⟨terminal⟩ ::= ⟨ident⟩ "=" "¤" ⟨file_name⟩ "¤" ";"
⟨non_terminals⟩ ::= **nonterminals** ⟨non_terminal_list⟩
⟨non_terminal_list⟩ ::= ⟨non_terminal⟩
| ⟨non_terminal⟩ ⟨non_terminal_list⟩
⟨non_terminal⟩ ::= ⟨ident⟩ ";"

Les sections ⟨start⟩ et ⟨property⟩ définissent respectivement le symbole de départ et la propriété à vérifier.

⟨start⟩ ::= **start** ⟨ident⟩ ";"
⟨property⟩ ::= **property** ⟨ident⟩ "=" "¤" ⟨file_name⟩ "¤" ";"

Une règle de composition est définie par un symboleΓun automate booléen et son arité. Cet automate booléen reconnaît un langage sur l'alphabet $\{X, X0, X1, \ldots, X(n-1)\}$ où X est la liste de signaux ⟨inputs⟩ et n l'arité de la composition.

⟨compositions_rules⟩ ::= **compositions** ⟨compositions_rule_list⟩
⟨compositions_rule_list⟩ ::= ⟨composition_rule⟩
| ⟨composition_rule⟩ ⟨compositions_rule_list⟩
⟨composition_rule⟩ ::= ⟨ident⟩ "=" "¤" ⟨file_name⟩ "¤" "(" ⟨composition_rule_arity⟩ ")" ";"

Par exemple, si l'ensemble des signaux de communication du réseau est $\{x, y, z\}$, et l'arité de la composition est 3, l'automate booléen a $\{x, y, z, x0, y0, z0, x1, y1, z1, x2, y2, z2\}$ comme ensemble de signaux d'entrée.

La déclaration des règles de production a la syntaxe suivante :

⟨production_rules⟩ ::= ⟨production_rule⟩
| ⟨production_rule⟩ ⟨production_rules⟩
⟨production_rule⟩ ::= ⟨ident⟩ ">" ⟨ident⟩ "(" ⟨ident_list⟩ ")" ";"
⟨ident_list⟩ ::= ⟨ident⟩
| ⟨ident⟩ "," ⟨ident_list⟩

L'identificateur de gauche doit être un symbole non terminal. Il représente le but de la règle de production. Le deuxième identificateur doit être celui d'une composition. La liste d'identificateurs doit être une liste de symboles terminaux ou non terminaux représentant les arguments de la composition. De plus, le symbole de départ doit être le but de la première règle de production, et ne doit pas apparaître dans les règles suivantes.

Exemple 12.4 :

Cet exemple décrit l'anneau à jeton circulant de la section 10.1, page 109. Il est utilisé afin d'assurer l'exclusion mutuelle d'une ressource pour n processus. Les signaux de communication du réseaux sont les suivants :

in	Un jeton est reçu par le réseau.
u	Un processus du réseau utilise la ressource.
ex	L'exclusion mutuelle de la ressource est vérifiée par le réseau.
out	Un jeton est émis par le réseau.

La grammaire de réseau décrivant cet exemple est la suivante :

```
inputs
    in;
outputs
    u, ex, out;
terminals
    U = "unit.caut";
nonterminals
    R;
start
    S;
compositions
    C = "composition.caut"(2);
    E = "environment.caut"(1);
property
    "property.caut";
productions
    S > E(R);
    R > C(U, R);
    R > U;
```

Cette grammaire permet de générer les réseaux suivants :

$$
\begin{array}{lcl}
E(U) & \equiv & E \parallel U \\
E(C(UIU) & \equiv & E \parallel U \parallel U \\
E(C(UIC(UIU))) & \equiv & E \parallel U \parallel U \parallel U \\
E(C(UIC(UIC(UIU)))) & \equiv & E \parallel U \parallel U \parallel U \parallel U
\end{array}
$$

□

12.4 Résumé des commandes

BANG prend comme paramètres un (pour les opérations autres que les élargissements) ou deux (pour les élargissements) fichiersΓet un certain nombres de paramètres. La commande

bang -h

affiche un résumé des commandes et options disponibles.

La syntaxe générale d'une commande de BANG est

bang ⟨**file_name1**⟩ [⟨**file_name2**⟩] ⟨**operation**⟩ [⟨**options**⟩]

Les fichiersΓles opérations et les options peuvent apparaître dans n'importe quel ordreΓsauf pour les élargissementsΓoù les fichiers doivent apparaître dans le bon ordre (i.e.Γcelui de la suite dont il faut abstraire la limite).

12.4.1 Opérations

BANG admet les opérations suivantes :

12.4.1.1 Opérations de traduction

-l : Produit un fichier Lustre.
-atg : Produit un fichier atg.
-ba : Produit un fichier ba.
-ald : Produit un fichier aut compatible avec ALDEBARAN.
-aut : Produit un fichier aut.
-ec : Produit un fichier ec pour visualiser les états à l'aide de l'outil SIMLUS de Raymond.

12.4.1.2 Opérations de vérification

-c : Calcule une assertion extensible.
-r : Calcule l'ensemble des états accessibles.
-f : Vérification en avant d'un invariant.
-b : Vérification en arrière d'un invariant.
Cet algorithme n'est implémenté que symboliquement (i.e.Γpour les circuits booléens).
-net : Vérification d'une grammaire de réseau.

12.4.1.3 Opérations de calcul

-exp : Calcule une expression d'automates à partir d'un fichier.
- : Commandes en ligne.
-m : Minimisation d'un automate selon l'équivalence de traces.
-wd : Élargissement (voir options plus loin).
-int : Interprétation d'un automate booléen (voir chapitre 4Γpage 43).

12.4.1.4 Opérations de synthèse

-as : Synthèse complète de processus.
-sas : Synthèse partielle de processus.
-os : Synthèse d'un observateur.

12.4.2 Options

BANG admet les options suivantes :

12.4.2.1 Options de sortie

-o ⟨file_name⟩ : Nom du fichier de sortie.
-sort : Trie les transitions d'un automate énumératif.
-ext1 : Visualisation des formules booléennes sans condition (if ... then ... else).
-ext2 : Visualisation étendue des formules booléennes (compatible avec LUSTRE).

12.4.2.2 Options d'élargissement

-dep1 : Premier paramètre de profondeur.
-dep2 : Deuxième paramètre de profondeur.
-fwd : Nombre de pas de calcul en avantΓavant le calcul en arrière.
-n : Nombre de pas de calcul avant élargissement.

12.4.2.3 Options générales

-d : Affiche un diagnostic.
-sift : Autorise le réordonnancement automatique des variables.
-symb : Utilise uniquement des algorithmes symboliques.
-mem : Minimise la mémoire utilisée en sauvegardant certains résultats intermédiaires sur disque.

12.4.2.4 Algorithmes de précondition

Un certain nombre d'algorithmes de calcul de la précondition ont été implémentés. Ils sont décrits dans le chapitre 14Γpage 159.

-ratel :

Utilise l'algorithme de précondition de Ratel (par sous but).

-ratel_restrict :

Utilise l'algorithme de précondition de Ratel (par sous but) avec l'opérateur restrict ↑.

-raymond :

Utilise l'algorithme de précondition de Raymond (par conjonction).

-raymond_restrict :

Utilise l'algorithme de précondition de Raymond (par conjonction) avec l'opérateur restrict ↑.

-coudert :

Utilise l'algorithme de précondition de Coudert (par disjonction).

-coudert_restrict :

Utilise l'algorithme de précondition de Coudert (par disjonction) avec l'opérateur restrict ↑.

-coudert_info :

Utilise l'algorithme de précondition de Coudert (par disjonction) avec décoration des noeuds.

Chapitre 13

Calcul des abstractions

La complexité du calcul des invariants vu dans la partie II est due principalement aux abstractions existentielle et universelle de signaux. Ce chapitre présente et discute des algorithmes de calcul de ces abstractions.

13.1 Introduction

13.1.1 Problème

Soit X un ensemble de signauxΓet soit $T \subseteq X^\infty$ un langage sur X. Supposons que X se partitionne en un ensemble X_v de signaux visibles et un ensemble X_{nv} de signaux non visibles :

$$X = X_v \cup X_{nv} \quad \text{et} \quad X_v \cap X_{nv} = \emptyset$$

On peut alors se poser deux problèmes :

- Trouver le processus le plus général tel qu'il existe un processus externeΓdit coopérantΓqui peut émettre des signaux non visibles de façon à satisfaire T. Le langage des traces de ce processus est appelé l'abstraction existentielle de T et est noté $\exists X_{nv}, T$.
- Trouver le processus le plus général tel que pour tout processus externeΓdit non coopérantΓ émettant des signaux non visiblesΓla propriété T est satisfaite. Le langage des traces de ce processus est appelé l'abstraction universelle de T et est noté $\forall X_{nv}, T$.

Plus précisémentΓon rappelle les définitions suivantes (voir section 7.2.3Γpage 77) :

Définition 13.1 :

Les abstractions existentielle et universelle de T par rapport aux variables de $X_{nv}\Gamma$ notées respectivement $\exists X_{nv}, T$ et $\forall X_{nv}, T\Gamma$sont définies par

$$\begin{aligned} \exists X_{nv}, T &= \{\ \tau_v \in X_v^\infty \mid \exists \tau_{nv} \in X_{nv}^{|\tau_v|},\ \tau_v \odot \tau_{nv} \in T\ \} \\ \forall X_{nv}, T &= \{\ \tau_v \in X_v^\infty \mid \forall \tau_{nv} \in X_{nv}^{|\tau_v|},\ \tau_v \odot \tau_{nv} \in T\ \} \end{aligned}$$

□

Exemple 13.1 :

Soit $X = \{a, b\}$. Utilisons des notations booléennes pour écrire les ensembles de sous-ensembles de X (par exemple$\Gamma\{\ \}, \{b\}\ \}$ est écrit $\overline{a}$) et les notations standards des expressions régulières pour décrire les ensembles de traces sur X. Soit $T = (\overline{a})^* + (\overline{a}\overline{b}.ab)^*$. Alors

$$\exists b, T = (\overline{a})^* + (\overline{a}.a)^* \quad \text{et} \quad \forall b, T = (\overline{a})^*$$

(voir figure 13.1Γpage 156). □

Le problème de l'abstraction de signaux a été traité entre autres par la communauté des DES ("Discrete Event Systems" décrite dans [RW87Γ RW89]Γ voir section 2.2Γ page 30) dans le cadre asynchrone. Les sections suivantes présentent les travaux de Rudie et WonhamΓ puis de Bergeron. Dans la suiteΓ nous proposerons des algorithmes pour calculer les abstractions existentielle et universelle de langages définis par des automates énumératifsΓ ou des circuits booléens.

13.1.2 Contrôle décentralisé

13.1.2.1 Introduction

Dans [RW92]Γ Rudie et Wonham partitionnent donc l'ensemble X des signaux d'entrée en l'ensemble X_v des signaux visibles (ou observables)Γ et l'ensemble X_{nv} des signaux non visibles (ou non observables). Ils définissent alors la projection canonique $P : X^* \to X_v^*$ par

$$\begin{aligned} P(\varepsilon) &= \varepsilon \\ \forall x \in X_v,\ P(x) &= x \\ \forall x \in X_{nv},\ P(x) &= \varepsilon \\ \forall x \in X, \forall \tau \in X^*,\ P(\tau.x) &= P(\tau).P(x) \end{aligned}$$

i.e.Γ P efface tous les événements non observables (on rappelle que cette théorie est définie dans le cadre asynchrone). Pour un langage $T \subseteq X^*$Γ on note $P(T) = \{P(\tau) | \tau \in T\}$.

Rudie et Wonham se posent alors deux problèmes :

1. **Global Problem (GP)** : Étant donnés A un automateΓ T_1 et T_2 deux langages préfixes-clos tels que $T_1 \subseteq T_2 \subseteq T_A$Γ et des ensembles $X_{1,c}, X_{2,c}, X_{1,o}, X_{2,o} \subseteq X$Γ construire deux superviseurs locaux $\mathcal{S}_1$ et $\mathcal{S}_2$ tels que $T_1 \subseteq T_{\mathcal{S}_1 \wedge \mathcal{S}_2 / A} \subseteq T_2$.

   On rappelle que de notre point de vueΓ les ensembles de signaux contrôlables $X_{1,c}$ et $X_{2,c}$ représentent les sorties des superviseurs $\mathcal{S}_1$ et $\mathcal{S}_2$.

2. **Global Problem with Zero Tolerance (GPZT)** : Étant donnés A un automateΓ T un langage préfixe-clos tel que $T \subseteq T_A$Γ et des ensembles $X_{1,c}, X_{2,c}, X_{1,o}, X_{2,o} \subseteq X$Γ construire deux superviseurs locaux $\mathcal{S}_1$ et $\mathcal{S}_2$ tels que $T_{\mathcal{S}_1 \wedge \mathcal{S}_2 / A} = T$.

13.1.2.2 Résolution de GPZT

Pour résoudre GPZTΓ Rudie et Wonham introduisent la notion de coobservabilité. Pour celaΓ ils définissent la fonction $nextact_T$ par

$$\begin{aligned} &\forall x \in X,\ \tau, \tau' \in X^*, \\ ((\tau, x, \tau') \in nextact_T) \ - \ &(\tau'.x \in T \wedge \tau \in T \wedge \tau.x \in T_A \Rightarrow \tau.x \in T) \end{aligned}$$

IntuitivementΓ $nextact_T(\tau, x, \tau')$ signifie que x est accepté ou refusé de façon identique par τ et τ'.

Définition 13.2 :

> Étant donnés un automate AΓ les ensembles $X_{1,c}, X_{2,c}, X_{1,o}, X_{2,o} \subseteq X$Γ les projections $P_1 : X^* \to X_{1,o}^*$ et $P_2 : X^* \to X_{2,o}^*$Γ un langage $T \subseteq T_A$ est coobservable si
>
> $$\begin{aligned} &\forall \tau, \tau', \tau'' \in X^*,\ P_1(\tau) = P_1(\tau') \wedge\ P_2(\tau) = P_2(\tau'') \\ &\Rightarrow \begin{cases} \forall x \in X_{1,c} \cap X_{2,c},\ (\tau, x, \tau') \in nextact_k \vee (\tau, x, \tau'') \in nextact_k & \wedge \\ \forall x \in X_{1,c} \setminus X_{2,c},\ (\tau, x, \tau') \in nextact_k & \wedge \\ \forall x \in X_{2,c} \setminus X_{1,c},\ (\tau, x, \tau'') \in nextact_k & \wedge \\ (\tau, \tau') \in markact_T \vee (\tau, \tau'') \in markact_T \end{cases} \end{aligned}$$
>
> □

Cette définition signifie qu'à tout moment, au moins une des deux observations permet de décider correctement de la marche à suivre.

Propriété 13.1 :

Il existe deux superviseurs complets $\mathcal{S}_1$ et $\mathcal{S}_2$ résolvant GPZT si et seulement si T est contrôlable avec respect de A et coobservable avec respect de A, P_1 et P_2. □

Intuitivement, les deux superviseurs sont construits de la manière suivante : les états du superviseur i seront les classes d'équivalence de la relation d'équivalence $\tau \sim \tau' \Leftrightarrow P_i(\tau) = P_i(\tau')$, notées $[\tau]_i$. La fonction de transition est définie par

$$\delta_i([\tau]_i, x) = [\tau.x]_i$$

et la fonction de contrôle par

$$\Phi([\tau]_i, x) = \begin{cases} 0 & \text{si } \forall \tau' \in [\tau]_i,\ \tau'.x \notin T \\ 1 & \text{sinon} \end{cases}$$

13.1.2.3 Résolution de GP

Pour résoudre GP, Rudie et Wonham introduisent

$$\underline{CCCo}(T) \quad = \quad \{\, T' \mid T' \supseteq T,\ T' \text{ est préfixe-clos, contrôlable et coobservable} \,\}$$

Propriété 13.2 :

Si $T_1 \neq \emptyset$, GP est soluble si et seulement si $inf\ \underline{CCCo}(T_1) \subseteq T_2$. □

Commentaires : Cette approche a l'avantage de fournir des résultats théoriques d'existence de superviseur. Les preuves de ces résultats fournissent des algorithmes de calcul. En pratique, ces algorithmes restent peu efficaces car ils utilisent un codage énumératif des automates. Dans la suite, nous proposerons des algorithmes similaires, mais adaptés au modèle synchrone et codés en symbolique, qui permettent de considérer des automates beaucoup plus gros.

13.1.3 Observation linéaire

Bergeron, dans [Ber94a, Ber94b, Ber95], tout en reprenant les résultats de Rudie et Wonham [RW92], introduit en plus la notion d'observabilité linéaire.

Soit P un processus. Une spécification pour le processus P est modélisée par un automate Z tel que $T_Z \subseteq T_P$. Étant donnés n sites, chacun ayant l'ensemble de signaux non observables $X_{nv,i}$, elle cherche à construire n automates $C_1, \ldots, C_n$ tels que $T_Z = T_{P \times \Pi C_i}$. Chaque automate C_i est appelé automate d'observation par rapport à $X_{nv,i}$.

Définition 13.3 :

Un automate O est un automate d'observation par rapport à X_{nv} si pour tout $s \in X^*$ et $x \in X_{nv}$ tels que $s \cdot x$ soit défini, on ait $s \cdot x = s$. □

Une spécification Z est dite observable si et seulement s'il existe n automates $C_1, \ldots, C_n$ tels que $T_Z = T_{P \times \Pi C_i}$.

Proposition 13.1 :

Étant donnés un automate A et un ensemble d'événements non observables $X_{nv} \subseteq X$, il existe un automate d'observation minimal plus grand que A. Il est noté $\Theta(A)$ et est appelé l'observateur minimal de A par rapport à X_{nv}. □

Construction de l'observateur minimal: Les états de $\Theta(A)$ sont les parties non vides $\mathcal{P}^+(S_A)$ de S_A. L'état initial est défini par

$$I \quad = \quad \{\, i \cdot u \mid i \cdot u \text{ est défini et } u \in X_{nv}^* \,\}$$

Sa fonction de transition

$$\circ \quad : \quad \mathcal{P}^+(S_A) \times X \to \mathcal{P}^+(S_A)$$

est définie par

$$\forall (S, x) \in \mathcal{P}^+(S_A) \times X,$$
$$S \circ X \;=\; \begin{cases} \text{indéfini} & \text{si } \forall s \in S, s \cdot X \text{ est indéfini} \\ \{\, s \cdot Xu \mid s \in S \Gamma s \cdot Xu \text{ est défini et } u \in X_{nv}^* \,\} & \text{sinon} \end{cases}$$

Ses états marqués sont définis par

$$\{\, S \subseteq S_A \mid \exists s \in S,\ s \in F_A \,\}$$

Propriété 13.3 :

Soient n sites avec des événements non observables $X_{nv,i}\Gamma$une spécification Z sur un processus P et $\Theta_i(Z)$ les observateurs minimaux par rapport à $X_{nv,i}$. Alors Z est observable si et seulement si $T_Z = TP \times \Pi\ \Theta_i(Z)$. □

Les observateurs ainsi calculés peuvent avoir un nombre d'états exponentiel de celui de l'automate de départ. La définition suivante est un cas particulier où ce nombre est limité.

Définition 13.4 :

Étant donnés n sites avec des événements non observables $X_{nv,i}\Gamma$une spécification Z sur un processus P est linéairement observable s'il existe des automates $C_1, \ldots, C_n$ tels que C_i soit un automate d'observation par rapport à $X_{nv,i}$ et tels que $Z \leftrightarrow P \times \Pi C_i$. □

Propriété 13.4 :

Si Z est linéairement observableΓet si Z a k étatsΓZ est linéairement observable par des automates ayant au plus k états. □

Commentaires : La définition d'un automate "à la Bergeron" est identique à celle d'un générateur "à la Wonham". La notion d'observabilité de Bergeron correspond à celles de contrôlabilité et de coobservabilité de Rudie et Wonham.

L'apport de Bergeron se situe dans la notion d'observabilité linéaire. Sous ces conditionsΓ l'abstraction de signaux produit des automates de taille linéaire par rapport à l'automate initial.

Dans notre approcheΓnous n'allons pas cacher des événementsΓmais des signaux. Cela est équivalent en fait à confondre certains événements. Par exempleΓsi le signal b est abstraitΓcela est équivalent à confondre les événements $a.b$ et $a.\overline{b}$. Dans les sections suivantesΓnous allons proposer des algorithmes

- qui adaptent les algorithmes asynchrones de RudieΓWonham et Bergeron au cas synchrone.
- qui étendent l'abstraction existentielle (effacement d'entrée) à l'abstraction universelle.
- qui utilisent des méthodes symboliques pour éviter l'explosion du nombre des états.

13.2 Algorithme d'abstraction énumératif

Soit $\Omega = (Q, q^0, X, \{\alpha^s\}, \delta^Q, \delta^O)$Γun observateur déterministe d'un langage $T \subseteq X^\infty$. On suppose que T décrit une propriété de sûreté (voir section 1.2.1Γpage 19)Γi.e.Γque T est un langage préfixe-clos. Dans cette sectionΓnous proposons des algorithmes énumératifs permettant de calculer les abstractions existentielle $\exists X_{nv}, T$ et universelle $\forall X_{nv}, T$

Supposons que Ω ait un état puits uniqueΓnoté q^sΓtel que le signal d'alarme α^s ne soit émis que par les transitions l'atteignant :

$$\alpha^s \in \delta^O(q,x) \;-\; \delta^Q(q,x) = q^s$$

$\exists X_{nv}, \Omega$ est simplement construit en effaçant tous les signaux de X_{nv} des étiquettes des transitions de Ω. On obtient un automate non déterministe qui peut être déterminisé par l'algorithme classique suivant :

$\exists X_{nv}, \Omega = (2^Q, \{q^0\}, X_v, \{\alpha^s\}, \delta^Q_{\exists X_{nv},\Omega}, \delta^O_{\exists X_{nv},\Omega})$ où

$$\delta^Q_{\exists X_{nv},\Omega} :\; 2^Q \times 2^{X_v} \to 2^Q \quad \text{et} \quad \delta^O_{\exists X_{nv},\Omega} :\; 2^Q \times 2^{X_v} \to \{\emptyset, \{\alpha^s\}\}$$

$$\delta^Q_{\exists X_{nv},\Omega}(\tilde{q}, x_v) \;=\; \{\; q' \mid \exists q \in \tilde{q}, \exists x_{nv} \in X_{nv},\; q' = \delta^Q(q, x_v \cup x_{nv}) \;\}$$

$$\delta^O_{\exists X_{nv},\Omega}(\tilde{q}, x_v) \;=\; \begin{cases} \{\alpha^s\} & \text{si } \delta^Q_{\exists X_{nv},\Omega}(\tilde{q}, x_v) = \{q^s\} \\ \emptyset & \text{sinon} \end{cases}$$

i.e.Γle signal d'alarme α^s n'est émis que lorsque l'état $\{q^s\}$ est atteint.

$\forall X_{nv}, \Omega$ accepte une suite $\tau^v = (\tau^v_0, \tau^v_1, \dots)$ d'ensemble de signaux visibles si et seulement si pour toute séquence $\tau^{nv} = (\tau^{nv}_0, \tau^{nv}_1, \dots)$ d'ensemble de signaux non visiblesΓΩ accepte la suite $\tau = (\tau^v_0 \cup \tau^{nv}_0, \tau^v_1 \cup \tau^{nv}_1, \dots)$. AinsiΓune suite d'événements est refusée si et seulement si $\exists X_{nv}, \Omega$ atteint un état qui contient l'état puits q^s. $A_{\forall X_{nv},\Omega} = (2^Q, \{q^0\}, X_v, \{\alpha^s\}, \delta^Q_{\exists X_{nv},\Omega}, \delta^O_{\forall X_{nv},\Omega})$ où

$$\delta^O_{\forall X_{nv},\Omega}(\tilde{q}, x_v) \;=\; \begin{cases} \{\alpha^s\} & \text{if } q^s \in \delta^Q_{\exists X_{nv},\Omega}(\tilde{q}, x_v) \\ \emptyset & \text{sinon} \end{cases}$$

i.e.Γtel que le signal d'alarme α^s est émis lorsqu'un état contenant l'état puits q^s est atteint.

La figure 13.1 illustre ces deux algorithmes.

Dans le pire des casΓles abstractions de signaux non visibles peuvent produire des automates de taille exponentielle.

13.3 Algorithme d'abstraction symbolique

Cette section présente des algorithmes symboliques permettant de calculer les abstractions existentielle et universelle de langages codés par des circuits.

13.3.1 Automates sans *and* ni *not*

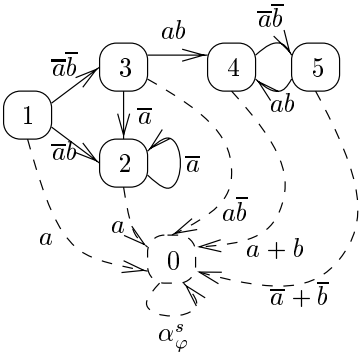
On considère un circuit booléen $C = (Init_C, \delta_C, \mathcal{I}_C)$ tel que

- La fonction de transition ait la forme suivante :

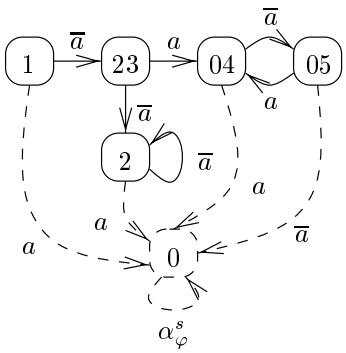
$$\forall l,\; 0 \le l \le n-1,\; \forall (q,i) \in \mathbb{B}^{n+m},\; \delta^l_C(q,i) \;=\; \bigvee_{0 \le j \le n-1} q_j . g_{l,j}(i)$$

  où $g_{l,j}$ est une fonction booléenne.

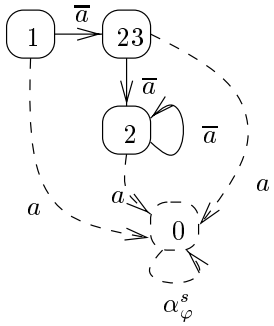
- Il existe un état puits tel que le signal d'alarme n'est émis que dans les transitions l'atteignant.



Automate Ω



$\exists\{b\},\ \Omega$



$\forall\{b\},\ \Omega$

FIG. 13.1 – Algorithme d'abstraction

– Cet état puits soit codé par le vecteur de variables d'état $(0,\ldots,0,1)$ et tel que dans tout autre étatΓq_{n-1} soit fausse. En d'autres termesΓ

$$\begin{aligned} \forall (q,i) \in \mathbb{B}^{n+m},\ \mathcal{I}_C(q,i) &- \left(\bigvee_{0\le l\le n-2} \delta^l_C(q,i) \right) \\ \overline{\mathcal{I}_C(q,i)} &- \delta^{n-1}_C(q,i) \end{aligned}$$

L'invariant $\mathcal{I}_C$ du circuit peut donc s'écrire sous la forme

$$\begin{aligned} \mathcal{I}_C &= \overline{q_{n-1}} \\ &= \bigvee_{0\le l\le n-2} q_l \end{aligned}$$

Alors les abstractions existentielle et universelle$\Gamma\ \exists X_{nv},\ C = (Init_C, \delta', \mathcal{I}_\exists)$ et $\forall X_{nv},\ C = (Init_C, \delta', \mathcal{I}_\forall)\Gamma$sont définies par

$$\begin{aligned} \forall l,\ 0\le l\le n-1,\ \forall (q,i)\in\mathbb{B}^{n+m},\ \delta'_l(q,i) &= \exists I',\ \delta_l(q,i) \\ \mathcal{I}_\exists(q,i) &= \bigvee_{0\le j\le n-2} q_j \\ \mathcal{I}_\forall(q,i) &= \overline{q_{n-1}} \end{aligned}$$

Preuve :

Nous allons définir une sémantique $\mathcal{S}$ sur les automates booléensΓen utilisant une syntaxe simplifiée de LUSTRE [HCRP91]. Soit $\mathcal{P}l$ (pour "*Petit* LUSTRE") le langage suivant :

$$\begin{array}{rcl} T & := & pre(T).i \\ & | & T \textbf{ or } T \\ & | & i \end{array}$$

où i est un vecteur booléen d'entrée de taille m. pre est l'opérateur prédécesseur de LUSTRE. $\mathcal{S}$ est définie par

$$\begin{array}{rcl} \mathcal{S}(i) & = & \{(i)\} \\ \mathcal{S}(pre(T).i) & = & \{(\tau_0, \ldots, \tau_{|\tau|-1}, i) \mid (\tau_0, \ldots, \tau_{|\tau|-1}) \in T\} \\ \mathcal{S}(T_1 \textbf{ or } T_2) & = & T_1 \cup T_2 \end{array}$$

Les opérateurs $\exists X_{nv}$ et $\forall X_{nv}$ sont définis sur les langages et sur les automates booléens comme dans les sections précédentes.

$$\begin{array}{rcl} \mathcal{S}(\exists X_{nv},\ i) & = & \exists X_{nv},\ i \\ \mathcal{S}(\exists X_{nv},\ T_1 \textbf{ or } T_2) & = & (\exists X_{nv},\ \mathcal{S}(T_1)) \cup (\exists X_{nv},\ \mathcal{S}(T_2)) \\ \mathcal{S}(\exists X_{nv},\ pre(T).\tau) & = & pre(\exists X_{nv},\ T).(\exists X_{nv},\ \tau) \end{array}$$

DoncΓle diagramme suivant commute

$$\begin{array}{ccc} A & \stackrel{\exists X_{nv}}{\longrightarrow} & \exists X_{nv},\ A \\ \mathcal{S} \downarrow & & \downarrow \mathcal{S} \\ T & \stackrel{\exists X_{nv}}{\longrightarrow} & \exists X_{nv},\ T \end{array}$$

La forme utilisée pour appliquer l'abstraction universelle est $\overline{\mathcal{P}l}$. Comme $\forall X_{nv},\ \overline{T} = \overline{\exists X_{nv},\ T}$Γle diagramme suivant commute

$$\begin{array}{ccc} A & \stackrel{\forall X_{nv}}{\longrightarrow} & \forall X_{nv},\ A \\ \mathcal{S} \downarrow & & \downarrow \mathcal{S} \\ T & \stackrel{\forall X_{nv}}{\longrightarrow} & \forall X_{nv},\ T \end{array}$$

□

Commentaire : Cet algorithme est très efficace sur la forme particulière des automates sans *and* ni *not*. En pratiqueΓnous n'avons pas d'algorithme pour calculer cette formeΓnous préférons donc utiliser l'algorithme de la section suivante.

13.3.2 Codage 1 parmi n

Le codage *un* parmi n permet d'obtenir un circuit booléen sans *and* ni *not*. Il consiste à coder chaque état de l'automate par une variable d'état en exclusion mutuelle (le circuit a donc autant de variables d'état que l'automate a d'états). L'algorithme énumératif de la section 13.2 peut alors s'appliquer. En effetΓaprès effacement des variables non visiblesΓplusieurs variables d'état peuvent être vraies en même temps. Un état du nouvel automate peut être considéré comme un ensemble d'états de l'ancienΓcomme dans l'algorithme classique de déterminisation.

Exemple 13.2 :

Reprenons l'exemple de la figure 13.1. Codons chaque état par une variable d'état. On obtient le circuit $C = (Init_C, \delta_C, \mathcal{I}_C)$ où

$$\begin{aligned}
Init_C(q,i) &= \overline{q_0}.q_1.\overline{q_2}.\overline{q_3}.\overline{q_4}.\overline{q_5} \\
\delta^0_C(q,i) &= q_0 \vee (q_1.a) \vee (q_2.a) \vee (q_3.a.\overline{b}) \vee \left(\; q_4.(a \vee b) \; \right) \vee \left(\; q_5.(\overline{a} \vee \overline{b}) \; \right) \\
\delta^1_C(q,i) &= 0 \\
\delta^2_C(q,i) &= (q_1.\overline{a}.b) \vee (q_2.\overline{a}) \\
\delta^3_C(q,i) &= q_1.\overline{a}.\overline{b} \\
\delta^4_C(q,i) &= (q_3.a.b) \vee (q_5.a.b) \\
\delta^5_C(q,i) &= q_4.\overline{a}.\overline{b} \\
\mathcal{I}_C(q,i) &= \overline{q_0} \\
&= \bigvee_{0 \leq i \leq n-1} q_i
\end{aligned}$$

Pour calculer les abstractions existentielle et universelleΓon efface b de la définition de δ_C. On obtient

$$\begin{aligned}
\delta^0_C(q,i) &= q_0 \vee (q_1.a) \vee (q_2.a) \vee (q_3.a) \vee q_4 \vee q_5 \\
\delta^1_C(q,i) &= 0 \\
\delta^2_C(q,i) &= (q_1.\overline{a}) \vee (q_2.\overline{a}) \\
\delta^3_C(q,i) &= q_1.\overline{a} \\
\delta^4_C(q,i) &= (q_3.a) \vee (q_5.a) \\
\delta^5_C(q,i) &= q_4.\overline{a}
\end{aligned}$$

et

$$\begin{aligned}
\mathcal{I}_\forall(q,i) &= \overline{q_0} \\
\mathcal{I}_\exists(q,i) &= \bigvee_{0 \leq i \leq n-1} q_i
\end{aligned}$$

□

Commentaire : Cet algorithme est optimal dans le pire des casΓmais assez mauvais en général. En effetΓsi on considère un automate à n étatsΓdans le pire des casΓle nombre d'états de l'automate abstrait est de taille exponentielle par rapport à celui d'origineΓi.e.Γe^n. Il est donc codable par un circuit à $log(e^n) = n$ variables d'étatΓqui est exactement la taille de notre solution.

Chapitre 14

Calcul de la post et de la précondition

Ce chapitre décrit des algorithmes symboliques calculant les fonctions de postcondition et de précondition.

14.1 Introduction

14.1.1 Définitions

Étant donnée δ une fonction de transition d'un automate booléenΓles fonctions *post* et *pre* calculent respectivement l'ensemble des états successeurs d'un ensemble de configurationsΓet l'ensemble des configurations prédécesseurs d'un ensemble d'états (voir figure 14.1 et section 1.1.3.1Γ page 16). Soient $\Gamma \subseteq \mathbb{B}^{n+m}$ et $\mathcal{Q} \subseteq \mathbb{B}^n$ respectivement un ensemble de configurations et un ensemble d'états. Les fonctions *pre* et *post* sont définies par

$$\begin{aligned} post(\Gamma) &= \{\ q' \in \mathbb{B}^n \mid \exists (q,i) \in \Gamma,\ q \xrightarrow{i} q'\ \} \\ pre(\mathcal{Q}) &= \{\ (q,i) \in \mathbb{B}^{n+m} \mid \exists q' \in \mathcal{Q},\ q \xrightarrow{i} q'\ \} \end{aligned}$$

Les algorithmes symboliques implémentant ces deux fonctions utilisent les deux opérateurs "restrict" ($\uparrow$) et "constrain" ($\downarrow$) introduit par CoudertΓBerthet et Madre dans [CBM89].

14.1.2 Les opérateurs "restrict" ($\uparrow$) et "constrain" ($\downarrow$)

Les opérateurs restrict (noté "$\uparrow$") et constrain (noté "$\downarrow$") sont des opérateurs binaires. Notons "Δ" un de ces deux opérateurs. Soient $\mathcal{F}_1$ et $\mathcal{F}_2$ deux formules booléennes. IntuitivementΓ"$\mathcal{F}_1 \Delta \mathcal{F}_2$" peut se traduire par "la valeur de $\mathcal{F}_1$ lorsque $\mathcal{F}_2$ est vraie". Plus précisémentΓlorsque $\mathcal{F}_2$ est vraieΓ $\mathcal{F}_1 \Delta \mathcal{F}_2$ est égale à $\mathcal{F}_1$. Lorsque $\mathcal{F}_2$ est fausseΓla valeur de $\mathcal{F}_1 \Delta \mathcal{F}_2$ est quelconque : elle est choisie de façon à simplifier la formule $\mathcal{F}_1$.

Les opérateurs "$\uparrow$" et "$\downarrow$" possèdent donc la propriété suivante :

$$(\mathcal{F}_1 \Delta \mathcal{F}_2) \wedge \mathcal{F}_2 = \mathcal{F}_1 \wedge \mathcal{F}_2$$

qui peut également s'écrire :

$$\mathcal{F}_2 \Rightarrow (\mathcal{F}_1 = \mathcal{F}_1 \Delta \mathcal{F}_2)$$

Plus généralementΓces opérateurs vérifient

$$\begin{array}{lcll} \mathcal{F}_2 \wedge \mathcal{F}_1 & \Rightarrow & \mathcal{F}_1 \Delta \mathcal{F}_2 & \text{et} \\ \mathcal{F}_2 \wedge \overline{\mathcal{F}_1} & \Rightarrow & \overline{\mathcal{F}_1 \Delta \mathcal{F}_2} & \text{d'où} \end{array}$$

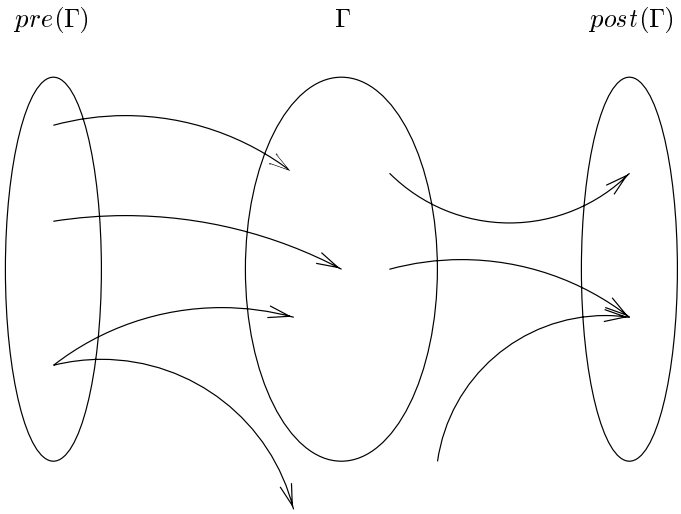


FIG. 14.1 – Les fonctions *post* et *pre*

$$\mathcal{F}_2 \wedge \mathcal{F}_1 \;\Rightarrow\; \mathcal{F}_1 \Delta \mathcal{F}_2 \;\Rightarrow\; \overline{\mathcal{F}_2} \vee \mathcal{F}_1$$

En pratique, si l'ensemble de configurations $\mathcal{F}_1 \cap \mathcal{F}_2$ est un ensemble "plus petit" que $\mathcal{F}_1$, la formule booléenne $\mathcal{F}_1 \wedge \mathcal{F}_2$ peut être beaucoup "plus grosse" que la formule $\mathcal{F}_1$. Par contre, la formule $\mathcal{F}_1 \uparrow \mathcal{F}_2$ est en général "plus petite" que la formule $\mathcal{F}_1 \wedge \mathcal{F}_2$. Dans certains cas, on peut donc optimiser une formule booléenne en remplaçant "$\wedge$" par "$\uparrow$" quand l'information perdue par l'opération peut être retrouvée.

L'opérateur "$\downarrow$" possède d'autres propriétés qui seront présentées dans le calcul du post (voir section suivante).

14.2 Calcul de la fonction *post*

On rappelle l'algorithme proposé par Coudert, Berthet et Madre [CBM89]. Cette méthode de calcul de la fonction *post* tire parti de la nature vectorielle de la fonction de transition. Soit $\Gamma \subseteq \mathbb{B}^{n+m}$ un ensemble de configurations. Notons $\delta(\Gamma) = \{\ \delta(q,i) \mid (q,i) \in \Gamma\ \}$. On utilise la propriété suivante de l'opérateur "constrain" :

$$\delta(\Gamma) \;=\; (\delta \downarrow \Gamma)(\mathbb{B}^{n+m})$$

Cette propriété permet de remplacer le calcul de l'image de Γ par δ par celui de l'image de $\mathbb{B}^{n+m}$ par $\delta \downarrow \Gamma$ (voir figure 14.2). On a alors

$$\begin{aligned} post(\Gamma) &= \{\ q' \in \mathbb{B}^n \mid \exists (q,i) \in \Gamma,\ q' = \delta(q,i)\ \} \\ &= \delta(\Gamma) \\ &= (\delta \downarrow \Gamma)(\mathbb{B}^{n+m}) \end{aligned}$$

Soit $\delta_i \in \mathbb{B}^{n+m} \to \mathbb{B}$ la $i^{\text{ème}}$ composante de la fonction de transition δ ($\delta \in \mathbb{B}^{n+m} \to \mathbb{B}^n$). On utilise une autre propriété de l'opérateur "$\downarrow$" :

$$[\delta_i] \downarrow \Gamma \;=\; [\delta_i \downarrow \Gamma]$$

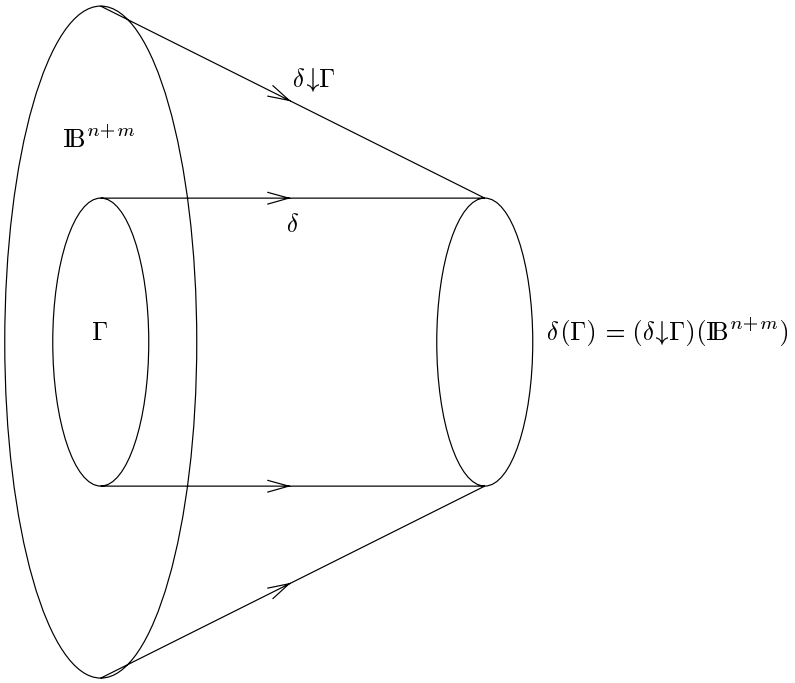


FIG. 14.2 – L'opérateur "$\downarrow$"

Notons $\delta_i^\Gamma = \delta_i\downarrow\Gamma$. On a

$$q' \in post(\Gamma) \quad - \quad \exists (q,i) \in \mathbb{B}^{n+m} \ (q'_1 = \delta_1^\Gamma(q,i), \ldots, q'_n = \delta_n^\Gamma(q,i))$$

$$- \quad \exists (q,i) \in \mathbb{B}^{n+m} \begin{cases} q'_1 = 1 \wedge \delta_1^\Gamma(q,i) = 1 \wedge q'_2 = \delta_2^\Gamma(q,i) \\ \quad \ldots q'_n = \delta_n^\Gamma(q,i) \\ \vee \\ q'_1 = 0 \wedge \delta_1^\Gamma(q,i) = 0 \wedge q'_2 = \delta_2^\Gamma(q,i) \\ \quad \ldots q'_n = \delta_n^\Gamma(q,i) \end{cases}$$

$$- \quad \begin{cases} q'_1 = 1 \wedge \exists (q,i) \in \mathbb{B}^{n+m}, \delta_1^\Gamma(q,i) = 1 \wedge q'_2 = \delta_2^\Gamma(q,i) \\ \quad \ldots q'_n = \delta_n^\Gamma(q,i) \\ \vee \\ q'_1 = 0 \wedge \exists (q,i) \in \mathbb{B}^{n+m}, \delta_1^\Gamma(q,i) = 0 \wedge q'_2 = \delta_2^\Gamma(q,i) \\ \quad \ldots q'_n = \delta_n^\Gamma(q,i) \end{cases}$$

$$- \quad \begin{cases} q'_1 = 1 \wedge \exists (q,i) \in \mathbb{B}^{n+m}, q'_2 = (\delta_2^\Gamma\downarrow\delta_1^\Gamma)(q,i) \\ \quad \ldots q'_n = (\delta_n^\Gamma\downarrow\delta_1^\Gamma)(q,i) \\ \vee \\ q'_1 = 0 \wedge \exists (q,i) \in \mathbb{B}^{n+m}, q'_2 = (\delta_2^\Gamma\downarrow\overline{\delta_1^\Gamma})(q,i) \\ \quad \ldots q'_n = (\delta_n^\Gamma\downarrow\overline{\delta_1^\Gamma})(q,i) \end{cases}$$

Ceci donne une méthode itérative de calcul de la fonction *post* par applications successives de l'opérateur "$\downarrow$".

14.3 Calcul de la fonction *pre*

Il existe au moins trois techniques pour calculer la précondition d'un ensemble d'états :

1. par disjonction : les configurations de la précondition sont ajoutées une à une au résultat.
2. par conjonction : les configurations qui n'appartiennent pas à la précondition sont enlevées une à une du résultat.
3. par sous-but.

Soit $\mathcal{Q}$ un ensemble d'états. Les algorithmes suivants utilisent les trois fonctions sur les BDD suivantes : soit $b = x.b[x/1] \vee \overline{x}.b[x/0]$ la décomposition de Shannon d'un BDD (où x est la variable racine du BDD). Alors

root(b)	=	x	Variable racine
high(b)	=	b[x/1]	Partie positive
low(b)	=	b[x/0]	Partie négative

14.3.1 Par sous-but

14.3.1.1 Version simple [Rat92]

Soit i un indice de variable d'état. Cette technique utilise la propriété suivante :

$$pre(q_i.\mathcal{Q}[q_i/1] \vee \overline{q_i}.\mathcal{Q}[q_i/0]) \quad = \quad \delta_i.pre(\mathcal{Q}[q_i/1]) \vee \overline{\delta_i}.pre(\mathcal{Q}[q_i/0])$$

Le calcul de la précondition peut ainsi s'effectuer par un simple parcours du BDD.

Algorithme 14.1 :

```
BDD pre_rec(BDD but)
begin:
  if but=0 ∨ but=1 then return but;
  root = root(but); high = high(but); low = low(but);
  if δ_root=1 then but = high;
                   goto begin;
  if δ_root=0 then but = low;
                   goto begin;
  high = pre_rec(high);
  low = pre_rec(low);
  return δ_root∧high∨¬δ_root∧low;
End of pre_rec

BDD Pre(BDD but)
  return pre_rec(but);
End of Pre
```

□

14.3.1.2 Version avec restrict

Cet algorithme de la précondition utilise l'optimisation proposée par Ratel [Rat92]. Soit Γ une formule des variables d'état et des variables d'entrée. On a

$$pre(\Gamma) = \exists i', \Gamma[(q,i)/(\delta(q,i), i')]$$

L'opération chère est le calcul de $\Gamma[(q,i)/(\delta(q,i),i')]\Gamma$ qui requiert la composition de fonctions. Soient $\delta = [\delta_i]$ et $X' = \exists i', \Gamma$. Notons $\Gamma(q') = \Gamma[q/q']$. Le calcul itératif suit le schéma suivant :

$$\begin{aligned} X'(\delta(q,i)) &= (\delta_1(q,i) \wedge X'(1, \delta_2(q,i), \ldots, \delta_n(q,i))) \\ &\quad \vee(\overline{\delta_1(q,i)} \wedge X'(0, \delta_2(q,i), \ldots, \delta_n(q,i))) \\ &= (\delta_1(q,i) \wedge X'(1, (\delta_2 \uparrow \delta_1)(q,i), \ldots, (\delta_n \uparrow \delta_1)(q,i))) \\ &\quad \vee(\overline{\delta_1(q,i)} \wedge X'(0, (\delta_2 \uparrow \overline{\delta_1})(q,i), \ldots, (\delta_n \uparrow \overline{\delta_1})(q,i))) \end{aligned}$$

Une bonne stratégie permet d'éviter de recalculer les fonctions $\delta_i \uparrow \delta_{i-1} \ldots \uparrow \delta_1$ en utilisant un cache.

Algorithme 14.2 :

```
BDD pre_rec(BDD but,BDD δ[ ])
begin:
  if but=0 ∨ but=1 then return but;
  root = root(but); high = high(but); low = low(but);
  if δ_root=1 then but = high;
                   goto begin;
  if δ_root=0 then but = low;
                   goto begin;
  for v∈StateVars do
    if v≠root ∧ δ_v ≠0 ∧ v∈high then δ'_v = δ_v↑δ_root;
                                else  δ'_v = 0;
  high = pre_rec(high,δ');
  for v∈StateVars do
    if v≠root ∧ δ_v ≠0 ∧ v∈low then δ'_v = δ_v↑¬δ_root;
                               else  δ'_v = 0;
  low = pre_rec(high,δ');
  return δ_root∧high∨¬δ_root∧low;
End of pre_rec

BDD Pre(BDD but)
  for v∈StateVars do
    if v∈but then δ'_v = δ_v;
             else  δ'_v = 0;
  return pre_rec(but,δ');
End of Pre
```

□

14.3.2 Par disjonction

14.3.2.1 Version simple

Cette technique proposée par Coudert Berthet et Madre parcourt le BDD de Q tout en retenant le chemin utilisé pour arriver à un noeud. Lorsque l'algorithme parvient à un 1 il ajoute au résultat le chemin y parvenant.

```
pre_rec(BDD path,BDD but)
  if but=0 then return;
  if but=1 then res = res∨path;
```

```
                return;
  root = root(but); high = high(but); low = low(but);
  if high≠0 then pre_rec(path∧δ_root⊓high);
  if low≠0 then pre_rec(path∧δ̄_root⊓low);
End of pre_rec

Bdd Pre(Bdd but)
  res = 0;
  pre_rec(1⊓but);
  return res;
End of Pre
```

14.3.2.2 Version avec restrict

Comme précédemment, il est possible d'utiliser l'opérateur restrict pour réduire la taille de la fonction de transition au fur et à mesure que l'algorithme progresse dans le parcours de l'arbre.

14.3.2.3 Version avec sauvegarde de l'information

Cette version de l'algorithme utilise la structure du Bdd pour éviter de parcourir deux fois la même branche. Pour cela, il décore chaque noeud du Bdd avec l'information suivante :

```
PathInfo = int nb;
           Bdd set;
```

où nb est égal au nombre de chemin atteignant ce noeud, et "set" est égal à l'ensemble des chemins arrivant à ce noeud. Initialement, les "set" sont tous mis à faux. L'algorithme précédent est modifié de la manière suivante :

Algorithme 14.3 :

```
pre_rec(Bdd path⊓Bdd but)
  if but=0 then return;
  if but=1 then res=res∨path;
  root = root(but); high = high(but); low = low(but);
  info = info(but);
  info.nb = info.nb-1;
  info.set = info.set∨path;
  if info.nb=0 then if high≠0 then pre_rec(info.set∧δ_var⊓high);
                    if low≠0 then pre_rec(info.set∧δ̄_var⊓low);
End pre_rec
```

□

14.3.3 Par conjonction

14.3.3.1 Version simple

Cet algorithme a été proposé par Raymond, en analogie avec le précédent. Au lieu de partir de l'ensemble vide (i.e. le Bdd 0) et d'ajouter les configurations appartenant à la précondition, il part de l'ensemble de toutes les configurations (i.e. le Bdd 1) et supprime les configurations n'appartenant pas à la précondition.

Algorithme 14.4 :

```
Bdd pre_rec(Bdd path⊓Bdd but)
  if but=1 then return 1;
```

```
  if path=0 then return 1;
  root = root(but); high = high(but); low = low(but);
  if low=0 then return (path⇒ δ_root) ∧ pre_rec(path⊓high);
  if high=0 then return (path⇒ δ̄_root) ∧ pre_rec(path⊓low);
  if high≠1 then high = pre_rec(path∧δ_root⊓high);
  if low≠1 then low = pre_rec(path∧δ̄_root⊓low);
  return high∧low;
End of pre_rec

Bdd Pre(Bdd but)
  return pre_rec(1⊓but);
End of Pre
```

□

14.3.3.2 Version avec restrict

Comme précédemment, l'opérateur restrict peut être utilisé pour tenter de simplifier le calcul.

14.4 Tests

Ces trois algorithmes ont été testés dans leurs différentes versions sur l'exemple de l'arbitre (voir section 10.3, page 115) avec différents nombres de processus. Les tableaux suivants résument les résultats obtenus, sans réordonnancement automatique des variables (voir section 12.1.2.2, page 140), et avec (on a noté $\wedge$, $\vee$ et $\uparrow$ pour respectivement "par conjonction", "par disjonction" et "avec restrict") :

Sans réordonnancement automatique :

Processus	Sous but	Sous but↑	∧	∧↑	∨	∨↑	∨Info
5	<1"	<1"	1"	1"	3"	3"	3"
10	7'17"	6'34"	saturation				
20	saturation						

Sans réordonnacement, la technique par sous-but semble la plus efficace. L'optimisation de cette technique par l'utilisation de l'opérateur restrict pour simplifier la fonction de transition semble également appréciable. Il est à noter que, pour le cas à 20 processus, la mémoire est saturée par le chargement du circuit booléen, avant même le calcul de la précondition.

Avec réordonnancement automatique :

Processus	Sous but	Sous but↑	∧	∧↑	∨	∨↑	∨Info
5	<1"	<1"	<1"	<1"	<1"	<1"	14"
10	<1"	1"	1"	1"	2"	2"	3"
20	30"	26"	1'17"	1'17"	2'27"	2'27"	20'32"
30	1'42"	1'17"	11'34"	11'32"	24'14"	24'10"	saturation
40	5'8"	12'30"	2h50'34"	2h53'2"	saturation		

Commentaires : Ces derniers résultats montrent tout d'abord l'extraordinaire efficacité du réordonnancement automatique des variables : les performances sont améliorées sans modification de l'algorithme. Deuxièmement, ces mesures confirment la supériorité de l'algorithme par sous-but.

L'algorithme avec décoration des feuilles est de loin le plus mauvaisΓcar il interdit le réordonnancement des variables lors du calcul de la précondition. Il est également à noter que l'optimisation du calcul par sous-but à l'aide de l'opérateur restrict est efficace pour les petits automatesΓmais pas pour les gros : cela vient du fait qu'à partir d'un certain momentΓla simplification de la fonction de transition prend plus de temps que le calcul de la précondition elle-même.

Conclusion

Bilan

Nous nous sommes posés dans l'introduction de ce document le problème de la vérification de propriétés sur les systèmes synchrones de grande taille. L'étude de la vérification modulaire nous a amenés à nous poser le problème de la synthèse de spécification pour un sous-programmeΓpuis de la synthèse de processus et d'invariants.

Synthèse de processus

Le problème de la synthèse a été résolu pour des propriétés régulières (exprimées à l'aide d'automates de Büchi)Γen se limitant à des propriétés de sûreté sur l'environnement. Nous avons proposés des algorithmes pour

- La synthèse d'invariant : un invariant permet à tout moment de choisir les sorties possibles du processusΓpour qu'une certaine propriété de sûreté soit satisfaite. AinsiΓsi les sorties du processus sont choisies de façon à vérifier cet invariantΓl'existence d'une stratégie gagnante dans le futur est assurée.

- La synthèse de chemin : un algorithme a été proposé pour synthétiser un invariantΓquiΓs'il est vérifiéΓpermet de conduire inéluctablement dans des états finals.

  Ces deux résultats permettent de caractériser la solution du problème de la synthèse de processus. De façon à satisfaire la propriété demandéeΓnotre processus doit se maintenir dans l'invariantΓet infiniment souvent suivre le chemin conduisant à un état final.

- La synthèse de processus : à partir des résultats précédentsΓplusieurs algorithmes (dont un symbolique très efficace) ont été proposés pour synthétiser l'ensemble des programmes déterministes satisfaisant une propriété exprimée à l'aide d'un automate de Büchi.

  Le résultat est un programme exécutableΓqui peut être simulé de façon interactive avec l'utilisateur.

On obtient ainsi l'ensemble des solutions *déterministes* du problème de la synthèse. Le non déterminisme est modélisé par l'ajout de variables oracles permettant de choisir entre différents comportements possibles.

Des exemples ont été présentésΓdont un de plus de 16 millions d'états. Ces résultats ne sont possibles que grâce à l'utilisation d'algorithmes symboliques utilisant les BDD. La technique du réordonnancement automatique des variables dans les BDD nous a permis de traiter des relativement gros exemples.

Synthèse d'invariant

La synthèse de processus a été étendue à la synthèse d'invariant afin de vérifier des réseaux paramétrés de n processus. Ces réseaux ont été exprimés par des grammaires de réseauΓet nous avons adapté la preuve par induction à ce cas particulier. Nous avons défini et utilisé une sémantique de traces pour réexprimer les équations d'induction. L'invariant d'un réseau généré par

une grammaire a alors été exprimé dans le cas général par un plus petit point fixe. Un théorème d'existence a été donné. Afin de résoudre le problème de convergenceΓcet invariant a également été exprimé par un plus grand point fixe :

- dans le cas linéaire : un théorème d'existence du plus grand point fixe a été donnéΓet des heuristiques ont été proposées pour en calculer une approximation régulière. Le cas linéaire a été aisément étendu aux réseaux en anneau par l'utilisation d'un environnement.

- dans le cas arborescent binaire : nous avons exprimé les fils gauche et droit d'un noeud à l'aide d'un seul vecteur d'invariantΓqui a été exprimé à l'aide d'un plus grand point fixe. Un théorème d'existence a été donné. Des heuristiques ont été proposées pour décomposer ce vecteur d'invariants en des invariants adéquats.

  Ce cas a été étendu théoriquement au cas général des grammaires de réseauΓmais seul le cas binaire a été implémenté.

Un certain nombre d'exemples ont été traités. Dans certains casΓnos techniques ont permis de prouver automatiquement qu'un réseau paramétré de processus satisfait une propriété donnéeΓ en exhibant un invariant du réseau. Même dans les exemples apparemment simples (tel le jeton circulant)Γcet invariant n'est pas trivial. Nous n'avons pas ainsi été capable d'en déterminer un à la main !

Dans d'autres exemplesΓnous avons montré que ces techniques apportaient une aide appréciable à la découverte d'invariant. En effetΓnotre technique n'a pas véritablement l'ambition de résoudre automatiquement tout type de problèmes. D'un point de vue plus réalisteΓnous pensons qu'elle apporte une aide réellement efficace à la détermination d'invariants.

Implémentation

Tous ces résultats ont été implémentés dans un outilΓappelé BANG. Des algorithmes énumératifs et symboliques ont été développésΓsur lesquels des tests de performances ont été effectués de façon à sélectionner les meilleurs.

Algorithmes énumératifs : Les algorithmes de base sur les automates ont été implémentés : post et préconditionΓcompositionΓinclusion des langagesΓminimisationΓetc ... De plusΓdes algorithmes spécifiques au calcul d'invariant ont été implémentés : les abstractions de signauxΓles élargissementsΓet les heuristiques de décomposition d'un vecteur d'invariants.

Algorithmes symboliques : Les mêmes algorithmes de base ont été implémentés que dans le cas énumératif : compositionΓinclusion des langagesΓminimisationΓabstraction de signaux ... En effort particulier a été fourni pour l'implémentation des post et précondition. Différents algorithmes ont été testés. Les performances de tous ces algorithmes ont été fortement améliorées par l'utilisation du réordonnancement automatique des variables des BDD.

D'autre partΓun petit simulateur permet de tester les processus synthétisés. Il est ainsi possible de jouer une partie de jeu de Nim avec l'ordinateur (bien que ce ne soit pas très intéressantΓ l'ordinateur gagnant à tous les coups !).

Dans ces deux casΓnous avons proposé 4 algorithmes (2 énumératifs et 2 symboliques) pour calculer la propriété minimale que doit satisfaire un sous-programme pour qu'un système satisfasse une propriété donnée. Ces algorithmes abstraient les signaux non visibles de deux manières différentes :

- de manière existentielle : si le programme et le sous-programme fonctionnent de façon coopérante.

- de manière universelle : si le programme et le sous-programme fonctionnent de façon non coopérante.

Perspectives

Ce travail peut servir de point de départ à un grand nombre de recherches.

Sur la synthèse de processus

- Nous nous sommes restreints à des propriétés de sûreté sur l'environnement. Il faudrait généraliser aux propriétés de vivacité.
- Dans certains cas simplesΓil doit être possible d'étendre la synthèse aux cas des processus à variables entières. A l'aide d'approximation et d'abstractionΓon peut envisager de synthétiser quelques solutions (mais sûrement pas toutes).
- Nous avons écrit un simulateur pour les programmes synthétisés. Il serait également nécessaire de générer un programme en langage classique (LUSTREΓou éventuellement C). Le seul problème qui se pose est la taille du fichier généré. Nos solutions ont été sauvegardées sous une forme compactéeΓun programme LUSTRE occuperait 10 à 100 fois plus d'espace.

Sur la synthèse d'invariants

- En premier lieuΓpour que cette technique puisse être plus largement utiliséeΓil semble nécessaire de développer un algorithme de calcul des abstractions existentielle et surtout universelle encore plus efficace. Des exemples plus gros pourraient ainsi être traitésΓpar exemple dans le non linéaire.
- Les élargissements proposés pour extrapoler un plus grand point fixe ont été proposés sur des bases expérimentales. Il est évidentΓque s'ils marchent sur les exemples présentésΓils ne fonctionneront pas forcément sur d'autres plus complexes.

  Il serait nécessaire d'effectuer un ensemble plus large d'études de cas de façon à fournir à l'utilisateur une batterie d'élargissementsΓparmi lesquels il pourrait choisir. En particulierΓ il faudrait étudier la convergence des invariants vers les grammaires de graphes [QJ95].
- La découverte d'élargissement pour le calcul en avant reste également un défi important. Un tel élargissementΓs'il apparaissait efficaceΓpermettrait de trouver des propriétés générales sur le réseau paramétré. Il est possible que cet élargissement doive être paramétré par la propriété que l'utilisateur souhaite démontrer.
- Il manque également à cette méthode une technique efficace de génération de diagnostic. A prioriΠl'approche en arrière ne permet pas de fournir un tel diagnostic (la méthode échoue en général lorsqu'on obtient un invariant nul). Il est peut être préférable de relancer un calcul en avantΓpour déterminer à partir de combien de processus la propriété est violée.
- Dans ce documentΓnous nous sommes limités à des processus d'états finisΓne possédant aucune donnée infinie. Beaucoup de programmes réalistes n'entrent pas dans cette catégorie. Il serait intéressant d'étudier des processus ayant une partie contrôle finieΓmais une partie donnée éventuellement infinie. Des techniques d'abstraction (tels les polyèdres [HPR94]) ou des démonstrateurs automatiques de théorème [OSR93] pourraient alors être utiles.

Bibliographie

[AK86] K. R. Apt and D. C. Kozen. Limits for automatic verification of finite-state concurrent systems. *Information Processing Letters*Γ22:307–309Γ1986.

[Arn92] A. Arnold. *Systèmes de transitions finis et sémantique des processus communicants.* MassonΓ1992.

[Ber94a] A. Bergeron. Easy problems in partial observation. Technical reportΓLACIMΓUniversité du Québec à MontréalΓ1994.

[Ber94b] A. Bergeron. Sharing out control in distributed processes. Technical reportΓLACIMΓ Université du Québec à MontréalΓ1994.

[Ber95] A. Bergeron. Sharing out control in distributed processes. *TCS*Γ139:163–186Γ1995.

[BG92] G. Berry and G. Gonthier. The Esterel synchronous programming language: DesignΓ semanticsΓimplementation. *Science of Computer Programming*Γ19(2):87–152Γ1992.

[BMK+96] BiehlΓMortenΓKlarlundΓNilsΓRauheΓand Theis. Algorithms for guided tree automata. In *Proc. WIA '96, Lecture Notes in Computer Science*. Springer VerlagΓ1996.

[CBM89] O. CoudertΓC. BerthetΓand J. C. Madre. Verification of sequential machines using boolean functional vectors. In L. J. M. ClaesenΓeditorΓ *Formal VLSI Correctness Verification*. North-HollandΓnovembre 1989.

[CC77] P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *4th ACM Symposium on Principles of Programming Languages, POPL'77*ΓLos AngelesΓjanvier 1977.

[CC92] P. Cousot and R. Cousot. Comparing the Galois connection and widening/narrowing approaches to abstract interpretation. In M. Bruynooghe and M. WirsingΓeditorsΓ *PLILP'92*ΓLeuven (Belgium)Γjanvier 1992. LNCS 631ΓSpringer Verlag.

[CES86] E. M. ClarkeΓE. A. EmersonΓand A. P. Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM TOPLAS*Γ8(2)Γ1986.

[CGB86] E. M. ClarkeΓO. GrumbergΓand M. C. Browne. Reasonning about networks with many identical finite-state processes. In *5th ACM Symposium on Principles of Distributed Computing, Calgary (Alberta)*Γpages 240–248Γaoût 1986.

[CGJ95] E. M. ClarkeΓO. GrumbergΓand S. Jha. Verifying parameterized networks using abstraction and regular languages. In *CONCUR'95*. LNCS 962ΓSpringer VerlagΓaoût 1995.

[CMT93] O. CoudertΓJ. C. MadreΓand H. Touati. TiGeR version 1.0 user guide. Technical reportΓDigital Paris Research LaboratoryΓnovembre 1993.

[CPS89] R. CleavelandΓJ. ParrowΓand B. Steffen. The concurrency workbench. In *Automatic Verification Methods for Finite State Systems*. J. Sifakis (ed)ΓSpringer-VerlagΓLNCS 407Γ1989.

[Eil74] S. Eilenberg. *Automata, Languages, and Machines*. Academic PressΓ1974.

[EN95] E. A. Emerson and K. S. Namjoshi. Reasoning about rings. In *Proc. 22th ACM Conf. on Principles of Programming Languages, POPL'95*ΓSan FranciscoΓjanvier 1995.

[EN96] E. A. Emerson and K. S. Namjoshi. Automatic verification of parameterized synchronous systems. In R. Alur and T. HenzingerΓeditorsΓ*8th International Conference on Computer Aided Verification, CAV'96*ΓRutgers (N.J.)Γ1996.

[FGK+96] J. C. FernandezΓH. GaravelΓA. KerbratΓR. MateescuΓL. MounierΓand M. Sighireanu. Cadp (cæsar/aldebaran development package): A protocol validation and verification toolbox. In Rajeev Alur and Thomas A. HenzingerΓeditorsΓ*Proceedings of the 8th Conference on Computer-Aided Verification (New Brunswick, New Jersey, USA)*Γvolume 1102 of *LNCS*Γpages 437–440. Springer VerlagΓAugust 1996.

[GGB87] T. GauthierΓP. Le GuernicΓand L. Besnard. SignalΓa declarative language for synchronous programming of real-time systems. In *Proc. 3rd. Conf. on Functional Programming Languages and Computer Architecture*. LNCS 274ΓSpringer VerlagΓ1987.

[GS96] S. Graf and H. Saïdi. Verifying invariants using theorem proving. In *Conference on Computer Aided Verification CAV'96*ΓLNCS 1102ΓJuly 1996.

[Hal93] N. Halbwachs. *Synchronous programming of reactive systems*. Kluwer Academic Pub.Γ 1993.

[Hal94] N. Halbwachs. BACΓa Boolean Automaton Checker. Technical reportΓVerimag LaboratoryΓfévrier 1994.

[Har87] D. Harel. Statecharts: A visual approach to complex systems. *Science of Computer Programming*Γ8(3)Γ1987.

[HCRP91] N. HalbwachsΓP. CaspiΓP. RaymondΓand D. Pilaud. The synchronous dataflow programming language LUSTRE. *Proceedings of the IEEE*Γ79(9):1305–1320Γseptembre 1991.

[HLR92a] N. HalbwachsΓF. LagnierΓand C. Ratel. An experience in proving regular networks of processes by modular model checking. *Acta Informatica*Γ29(6/7):523–543Γ1992.

[HLR92b] N. HalbwachsΓF. LagnierΓand C. Ratel. Programming and verifying critical systems by means of the synchronous data-flow programming language LUSTRE. *IEEE Transactions on Software Ingeneering, Special Issue on the Specification and Analysis of Real-Time Systems*ΓSeptember 1992.

[HLR93] N. HalbwachsΓF. LagnierΓand P. Raymond. Synchronous observers and the verification of reactive systems. In M. NivatΓC. RattrayΓT. RusΓand G. ScolloΓeditorsΓ*Third Int. Conf. on Algebraic Methodology and Software Technology, AMAST'93*ΓTwenteΓ juin 1993. Workshops in ComputingΓSpringer Verlag.

[HMP95] N. HalbwachsΓF. MaraninchiΓand Y. E. Proy. The railroad crossing problemΓmodeling with Hybrid Argos - Analysis with Polka. In *Second European Workshop on Real-Time and Hybrid Systems*ΓGrenoble (France)Γjuin 1995.

[HPR94] N. HalbwachsΓY. E. ProyΓand P. Raymond. Verification of linear hybrid systems by means of convex approximations. In B. LeCharlierΓeditorΓ*International Symposium on Static Analysis, SAS'94*ΓNamur (Belgique)ΓSeptember 1994. LNCS 864ΓSpringer Verlag.

[HW92] G. Hoffmann and H. WongToi. Symbolic synthesis of supervisory controllers. In *American Control Conference, Chicago*Γjuin 1992.

[ID93] C. N. Ip and D. L. Dill. Better verification through symmetry. In *11th International Symposium on Computer Hardware Description Languages and Their Application*Γ avril 1993.

[ID96] C. N. Ip and D. L. Dill. Better verification through symmetry. *Formal Methods in System Design*Γ9(1/2Γpp. 41-75)Γaoût 1996.

[KG95] R. Kumar and V. K. Garg. *Modeling and Control of Logical Discrete Event Systems.* Kluwer Academic PublishersΓ1995.

[KM89] R. P. Kurshan and K. McMillan. A structural induction theorem for processes. In *8th ACM Symposium on Principles of Distributed Computing*Γpages 239–247ΓEdmonton (Alberta)Γaoût 1989.

[KMM+97] Y. KestenΓO. MalerΓM. MarcusΓA. PnueliΓand E. Shahar. Symbolic model checking with rich assertional languages. In *submitted in 9th Conference on Computer Aided Verification, CAV'97*Γjuillet 1997.

[Les94] D. Lesens. Synthèse de programmes réactifs. Rapport de DEAΓEcole PolytechniqueΓ ParisΓjuillet 1994.

[Les97a] D. Lesens. The boolean automaton network grammar checker home page. http://www.imag.fr/VERIMAG/PEOPLE/David.Lesens/BANGΓ1997.

[Les97b] D. Lesens. Invariants of parameterized binary tree networks as greatest fixpoints. In *Sixth International Conference on Algebraic Methodology and Software Technology, AMAST'97*ΓSydneyΓdécembre 1997.

[LHR96] D. LesensΓN. HalbwachsΓand P. Raymond. Automatic construction of network invariants. In *International Workshop on Verification of Infinite State Systems, Infinity'96*ΓPisaΓaoût 1996.

[LHR97] D. LesensΓN. HalbwachsΓand P. Raymond. Automatic verification of parameterized linear networks of processes. In *24th ACM Symposium on Principles of Programming Languages, POPL'97*ΓParisΓjanvier 1997.

[Mar85] A. J. Martin. Distributed mutual exclusion on a ring of processes. *Science of Computer Programming*Γ5:265–276Γ1985.

[Mar92] F. Maraninchi. Operational and compositional semantics of synchronous automaton compositions. In *CONCUR'92*ΓStony BrookΓaoût 1992. LNCS 630ΓSpringer Verlag.

[MG91] R. Marelly and O. Grumberg. Gormel-grammar oriented model checker. Technical reportΓThe TechnionΓ1991.

[Mil80] R. Milner. *A Calculus of Communicating Systems.* LNCS 92ΓSpringer VerlagΓ1980.

[NS94] X. Nicollin and J. Sifakis. The algebra of timed processes ATP: theory and application. *Information and Computation*Γ114(1):131–178ΓOctober 1994.

[OSR93] S. OwreΓN. ShankarΓand J. M. Rushby. A tutorial on specification and verification using PVS. Technical reportΓComputer Science LaboratoryΓSRI InternationalΓ1993.

[QJ95] Y. M. Quemener and T. Jéron. Model-checking of infinite Kripke structures defined by simple graph grammars. In *SEGRAGRA'95, Joint COMPUGRAPH/SEMAGRAPH Workshop on Graph Rewriting and Computation.* A. CorradiniΓU. Montanari (réd.)Γ Elsevier Science B. V – ENTCSΓseptembre 1995.

[QS82] J. P. Queille and J. Sifakis. Specification and verification of concurrent systems in CESAR. In *International Symposium on Programming*. LNCS 137ΓSpringer VerlagΓ avril 1982.

[Rac78] C. Rackoff. The covering and boundedness problems for vector addition systems. *Theoretical Computer Science*Γ6(2):223–231Γavril 1978.

[Rat92] C. Ratel. *Définition et réalisation d'un outil de vérification formelle de programmes* LUSTRE *: le système* LESAR. PhD thesisΓUniversité Joseph FourierΓGrenoble Γjuillet 1992.

[Rau96] A. Rauzy. Utilisation du calcul propositionnel en sûreté de fonctionnementΓalgorithmique sur les diagrammes binaires de décision. Technical reportΓActes de l'école d'été MOVEP'96ΓMOdélisation et VErification des Processus parallèlesΓ1996.

[RW87] P. J. Ramadge and W. M. Wonham. Supervisory control of a class of discrete event processes. *SIAM J. Control and Optimization*Γ25(1)Γjanvier 1987.

[RW89] P. J. Ramadge and W. M. Wonham. The control of discrete event systems. *Proceedings of the IEEE*Γ77(1)Γjanvier 1989.

[RW92] K. Rudie and W. M. Wonham. Think globallyΓact locally: decentralized supervisory control. *IEEE Transactions on automatic control*Γ37(11)Γnovembre 1992.

[Saï96] H. Saïdi. A tool for proving invariance properties of concurrent systems automatically. In *Int. Workshop on Tools and Algorithms for the Construction and Analysis of Systems, TACAS'96*ΓLNCS 1055ΓMarch 1996.

[SG87] A. P. Sistla and S. M. German. Reasoning with many processes. In *Symp. on Logic in Computer Science, Ithaca*Γpages 138–152Γjuin 1987.

[SG89] Z. Shtadler and O. Grumberg. Network grammarsΓcommunication behaviors and automatic verification. In *Workshop on Automatic Verification Methods for Finite State Systems, Grenoble*. LNCS 407ΓSpringer VerlagΓjuin 1989.

[SV94] B. K. Szymanski and J. M. Vidal. Automatic verification of a class of symmetric parallel programs. In *Proc. 13th IFIP World Computer Congress*Γ1994.

[Ull84] J. D. Ullman. Computational aspects of VLSI. *Computer Science Press*Γ1984.

[WL89] P. Wolper and V. Lovinfosse. Verifying properties of large sets of processes with network invariants. In *International Workshop on Automatic Verification Methods for Finite State Systems, Grenoble*. LNCS 407ΓSpringer VerlagΓ1989.

Résumé : Cette thèse a pour cadre la vérification des systèmes réactifs de grande taille. Les systèmes réactifs sont des systèmes informatiques qui réagissent continûment à leur environnement physiqueΓà une vitesse déterminée par cet environnement. Ils sont particulièrement utilisés dans le cadre du contrôle des systèmes critiques : transportΓnucléaireΓcommande de processus industrielsΓ communication ... De par leur fonctionΓils doivent impérativement satisfaire des contraintes strictes de fonctionnement.

La vérification des systèmes réactifs de grande taille peut être effectuée en décomposant modulairement le programme. Des algorithmes de synthèse sont proposésΓqui permettent de déterminer les sous-programmes les plus généraux Γtels que le système complet satisfasse une propriété donnée. La vérification modulaire a été étendue aux cas des réseaux paramétrés de processus. L'utilisation d'observateurs synchrones permet de spécifier des propriétés sur un nombre quelconque de processus. La technique proposée consiste alors à exprimer un invariant de ce réseau à l'aide d'un plus petit ou d'un plus grand point fixeΓet d'utiliser des techniques d'extrapolation pour en calculer une approximation régulière. Les cas des réseaux linéaires (ou en anneau) et des réseaux arborescents ont particulièrement été étudiés.

L'ensemble des techniques et des algorithmes développés ont été implémentés dans un outil : *a Boolean Automaton Network Grammar checker*.

Mots-clés : Langages synchronesΓvérificationΓsynthèseΓréseaux paramétrés de processusΓinvariant.

Abstract: This thesis concern is the verification of large reactive systems. Reactive systems are computer systems which react continuously to their physical environmentΓat a speed fixed by this environment. They are especially used for the control of critical systems: transportΓnuclearΓ command of industrial processesΓcommunication ... For these usesΓthey must satisfy some critical working constraints.

The verification of large reactive systems can be performed using modular decomposition of the program. Synthesis algorithms are proposedΓallowing the construction of the most general subprogramsΓsuch that the complete system satisfies a given property. Modular verification has been extended to the case of parameterized networks of processes. The use of synchronous observers allows to specify properties on any number of processes. The proposed technique consists then in expressing an invariant of this network as a least or a greatest fixpointΓand in using extrapolation techniques to compute a regular approximation of these fixpoints. The cases of linear networks (or ring networks) and of tree networks have been particularly studied.

All these techniques and algorithms have been implemented in a tool: *a Boolean Automaton Network Grammar checker*.

Keywords: Synchronous languagesΓverificationΓsynthesisΓparameterized networks of processesΓ invariants.