# Mémoire de Master 2 : Synthèse de contrôleur dans les systèmes distribués synchrones

Nathalie Sznajder

2 Septembre 2005

Stage effectué sous la direction de Paul Gastin et Marc Zeitoun

Laboratoire Spécifications et Vérification
ENS Cachan
61 avenue du Président Wilson
94230 CACHAN

# Table des matières

**Résumé**

La synthèse de contrôleur consiste, étant donné un système partiellement spécifié, à déterminer un programme dont la synchronisation avec le système vérifie une spécification donnée. Lorsque le système est distribué, il s'agit de construire des programmes locaux, un pour chaque processus du système.

Des résultats standard sur les jeux multi-joueurs à information incomplète impliquent que ce problème est indécidable pour des spécifications LTL. Pour des architecures de communication dans lesquelles les processus sont ordonnés de façon linéaire et pour lesquelles l'information est transmise entre processus dans un seul sens, il a par contre été montré que le problème était (non élémentairement) décidable pour des spécifications $\mu$-calcul.

Après avoir comparé différents modèles et avoir extrait les points importants des principaux résultats antérieurs, ce rapport met en évidence un critère de décidabilité pour des spécifications plus naturelles que celles considérées précédemment. Cette approche fournit des résultats de décidabilité pour des architectures indécidables dans d'autres modèles.

# Chapitre 1

# Introduction au problème du contrôle distribué

Durant la dernière décennie, la communauté informatique a fait d'énormes progrès dans le développement d'outils et techniques de vérification logicielle. Il existe dans ce domaine trois approches complémentaires : l'approche « Preuves » (qui essaie de prouver formellement sur le code source que le programme vérifie les spécifications), l'approche « Test »(qui fonctionne elle sur le code qui s'exécute - et non sur le code source) et l'approche « Model Checking ». Le *Model Checking* (voir comme livres de référence sur le sujet [3] et [4]) commence par faire une abstraction du programme – le modèle – sous forme d'un système de transitions $M$. On se donne ensuite une spécification $\phi$ (typiquement en logique temporelle), et le problème consiste à répondre à la question : est-ce que le modèle $M$ vérifie la spécification $\phi$ ? Outre des problèmes de complexité dus à l'explosion combinatoire de la taille des modèles (en partie résolus par les techniques de vérification symbolique), il a été reproché à cette méthode de vérification d'intervenir uniquement *à la fin* du développement du programme. Le contrôleur permet d'intervenir au moment de la conception pour *synthétiser* des programmes vérifiant la spécification.

Le problème de synthèse de contrôleur est donc plus général : il part d'un système, contrôlable et *non totalement spécifié* interagissant avec un environnement qui lui est non contrôlable. Étant donnée une spécification, on veut contraindre le système à faire les choix qui permettent de vérifier la spécification. Divers problèmes de contrôle ont depuis longtemps été étudiés dans le cadre séquentiel. Il est classique [21] que le problème est décidable pour des spécifications rationnelles. Le système à contrôler ne comporte alors qu'un seul processus. Souvent, le problème se modélise par un jeu à deux joueurs, dans lequel le système correspond à l'arène du jeu, le contrôleur correspond à un joueur (existentiel), l'environnement au joueur universel,

pour lequel la spécification correspond à la fonction de gain, et la synthèse du contrôleur revient à calculer une stratégie gagnante. Par la suite, plusieurs extensions ont été envisagées : le cas de systèmes à états infinis (automates à pile, temporisés, hybrides ... - voir par exemple [5] ou [2]), et le cas de systèmes à états finis, mais distribués. C'est à ce dernier type de systèmes que nous nous sommes intéréssés.

Plus précisément, on se donne un système distribué sur une architecture définissant les communications possibles. Ce système est ouvert : son comportement est influencé par l'environnement dans lequel il s'exécute. Les systèmes considérés sont distribués : ils consistent en plusieurs processus qui communiquent par mémoire partagée. L'architecture du système, supposée connue et fixe au cours du temps, indique les communications qui peuvent avoir lieu. Un contrôleur distribué est un système respectant la même architecture de communication que le système de départ, donc composé de contrôleurs *locaux*, un par processus, à espace d'états fini. Chaque contrôleur local s'exécute sur l'un des sites du système, synchronisé avec son processus associé. Les contrôleurs peuvent communiquer (en respectant l'architecture), et inhiber certaines actions suivant la vue du système qu'ils peuvent avoir. Cette synchronisation restreint ainsi l'ensemble des actions locales possibles, en fonction de l'état courant du contrôleur. Par ailleurs, on se donne une spécification, i.e., un ensemble de comportements corrects du système distribué. Le problème est alors de déterminer s'il existe un contrôleur distribué qui, une fois synchronisé avec le système, ne produit que des comportements satisfaisant la spécification pour n'importe quels choix de l'environnement. Si c'est le cas, la question est alors de trouver des algorithmes pour construire un tel contrôleur. Il ne s'agit évidemment pas de construire un contrôleur centralisé qui aurait une vue globale.

Le cas distribué connaît également de nombreuses variantes, tant sur le mode de communication des processus (synchrone - cas que nous allons étudier : toutes les variables sont écrites en même temps, ou asynchrone - avec une communication par envoi de messages, voir par exemple [15], ou [8]), que sur le type de spécifications (LTL, CTL*, $\mu$-calcul, spécifications portant sur tous les canaux du système -comme dans [11], [6]- ou bien uniquement sur les canaux communiquant avec l'environnement - comme dans [20], ou encore spécifications locales, qui ne portent que sur ce que « voit » chaque processus - voir pour cela par exemple [14]), ou encore sur la prise en compte ou non d'un délai dans la transmission de l'information. Des tentatives d'unification de tous ces modèles ont été effectuées dans [16] et [1].

Comme dans le cas séquentiel, il existe un parallèle naturel avec les jeux distribués, donc les techniques de résolution couramment utilisées sont celles utilisées pour les jeux et les automates d'arbres. Les travaux de [20] se basant

sur [19] ont montré que le problème était indécidable en général.

Dans ce mémoire, nous nous plaçons dans le cadre d'une sémantique *synchrone*, avec des programmes à mémoire locale, et nous considérerons différents types de spécifications et de délais. Notons que nous considérerons des contrôleurs n'ayant pas de restrictions sur leurs mouvements.

L'objet du stage se base sur un résultat publié par Finkbeiner et Schewe [6], proposant un critère de décidabilité pour des spécifications du $\mu$-calcul portant sur tous les canaux du système. Ce critère ne nous semblant pas suffisamment fin (ce type de spécifications rend plus facile l'indécidabilité) et pas suffisamment naturel. En effet, il nous paraît plus proche de la réalité de considérer que la spécification ne porte que sur les *inputs* et *outputs* du système (les canaux communiquant avec l'environnement), laissant le programmeur libre de faire passer l'information sur les canaux internes de l'architecture comme il l'entend. Nous avons donc tenté de développer un autre critère, pour des spécifications portant uniquement sur les canaux externes. L'apport de ce mémoire se situe à plusieurs niveaux : tout d'abord une certaine unification des différents formalismes car, comme nous l'avons vu, les modèles sont multiples. Puis, la compréhension, et parfois la simplification de certaines preuves présentées succintement dans des articles de conférences. Enfin, la découverte de nouveaux résultats, en particulier d'un nouveau critère pour des spécifications portant uniquement sur les canaux de communication avec l'environnement, et pour un nouveau type d'architectures : les architectures *à bande passante maximale*.

Nous présenterons à la partie suivante les différentes notations et techniques qui seront utilisées dans ce rapport, ainsi que les principaux résultats de décidabilité et d'indécidabilité sur lesquels nous nous sommes basés, avant de présenter dans le chapitre 3 notre nouveau critère de décidabilité.

# Chapitre 2

# État de l'art

Nous allons dans ce chapitre définir le cadre formel dans lequel nous allons nous placer. Nous présenterons ensuite les principaux résultats existants à partir desquels nous avons travaillé. D'abord, l'exhibition d'une première architecture indécidable [20], rendant le problème indécidable en général, ensuite, la preuve de décidabilité de l'architecture de type pipe-line exposée par O. Kupferman et M.Y. Vardi dans [11], et enfin, un critère nécessaire et suffisant d'indécidabilité dans un cadre étendu [6].

## 2.1 Mise en place des définitions

Il existe dans la littérature plusieurs modèles pour représenter les architectures distribuées ; nous allons en présenter quelques uns, ainsi qu'un nouveau (les graphes de variables) qui a été défini durant le stage, et qui nous semblait plus adapté à notre cadre.

### 2.1.1 Modèles pour le problème de synthèse distribuée

**Architectures.** Une *architecture distribuée* définit des interactions entre processus communiquant par échange de messages ou mémoire partagée. On peut la représenter par un tuple : $(P, V, r, w)$ avec $P$ un ensemble de processus, $V$ un ensemble de variables et, si on note $env$ un processus spécial représentant l'environnement, $r : V \to P \cup \{env\}$ et $w : V \to P \cup \{env\}$ respectivement les fonctions de lecture et d'écriture sur ces variables. On pose $V = X \uplus Y \uplus T$ avec $X = w^{-1}(env)$ l'ensemble des variables d'entrée et $Y = r^{-1}(env)$ l'ensemble des variables de sortie du système. Les variables de $T$ sont dites *internes*, celles de $X \cup Y$ *externes*.

On définit un graphe étiqueté $(P, \to)$ ainsi : si $r(x) = q$ et $w(x) = p$, on

note $p \xrightarrow{x} q$. Intuitivement, $p \xrightarrow{x} q$ signifie que $p$ envoie de l'information à $q$ par le canal étiqueté par la variable $x$. On note que deux arêtes distinctes sont étiquetées par des variables distinctes. Si $r(z) = env$ et $w(z) = p$ on écrit $p \xrightarrow{z}$. De même, si $r(x) = p$ et $w(x) = env$ on note $\xrightarrow{x} p$.

On impose par ailleurs qu'il n'y ait pas d'arc de $env$ vers $env$. La figure 2.1 donne l'exemple d'une architecture, dite pipe-line.

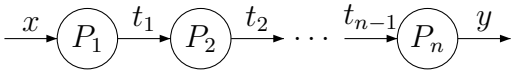



FIG. 2.1 – L'architecture pipe-line

On peut abstraire une architecture distribuée en ne considérant plus un graphe de processus (qui est la représentation classique) mais un *graphe de variables*. On définit le graphe $(V, A)$ dans lequel les sommets sont les variables, et les arêtes sont définies ainsi : $(x, y) \in A$ si $w(y) = r(x) \in P$. On peut facilement représenter un graphe de processus $\mathcal{P}$ par un graphe de variables $\mathcal{V}$. Les noeuds minimaux de $\mathcal{V}$ sont les variables $X$ de l'architecture distribuée, les noeuds maximaux les variables $Y$ et les arcs sont créés d'après la définition.

La figure 2.2 représente le graphe des variables du pipe-line de la figure 2.1.

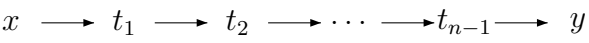



FIG. 2.2 – L'architecture pipe-line sous forme de graphe de variables

Les figures 2.3 et 2.4 représentent la même architecture sous forme de graphe de processus et sous forme de graphe de variables.

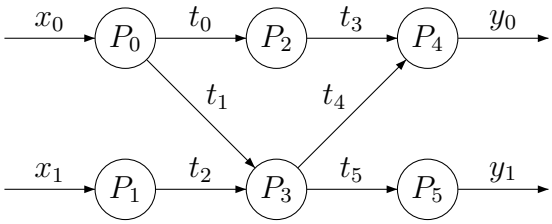



FIG. 2.3 – Architecture : graphe de processus

On peut par contre avoir un graphe de variables qui ne corresponde pas à un graphe de processus tel qu'on l'a défini. Par exemple, le graphe de variables

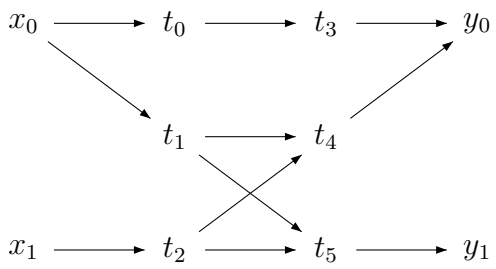



FIG. 2.4 – Architecture : graphe de variables

de la figure 2.5 est parfaitement correct, mais n'est pas issu d'un graphe de processus valide. En effet, soit le graphe de processus ainsi défini aurait deux processus différents ayant la variable $x_0$ en entrée, ce que le modèle ne permet pas[1]. Soit, le processus $p$ lisant $x_0$ et le processus $q$ lisant $x_1$ écrivent tous deux sur $z_1$. Ceci est interdit dans le modèle si $p \neq q$ (voir figure 2.6). Enfin, si $p = q$, on devrait avoir un arc $(x_1, z_0)$ dans le graphe des variables.

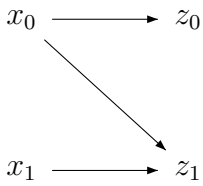



FIG. 2.5 – Exemple de graphe de variables

La représentation par graphe de variables est donc plus générale que la représentation par graphe de processus. Par ailleurs, les spécifications ne considèrent que les valeurs des variables, et les stratégies que l'on essaiera de synthétiser sont également associées aux variables, et non aux processus. Il est donc plus naturel de faire abstraction des processus et de travailler directement sur le graphe des variables, ce qui n'ajoute aucune difficulté technique.

**Domaines et bande passante.** Chaque variable $z$ de $V$ peut prendre ses valeurs dans un ensemble fini particulier $D(z)$. On considère que $|D(z)| > 1$. En effet, si une variable ne peut prendre qu'une seule valeur, elle peut être ignorée, car elle n'apporte aucune information.

On appelle *architecture à bande passante maximale* une architecture dans

[1]D'autres modèles [6] l'autorisent, afin de modéliser le multicast, cf. section 2.4.

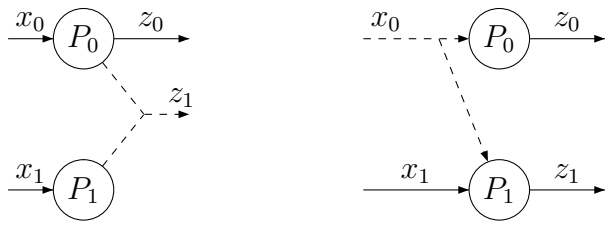



FIG. 2.6 – Graphes de processus non valides correspondant à la figure 2.5

laquelle, pour toute variable interne $t \in T$, on a

$$D(t) = \prod_{x \in X} D(x). \tag{2.1}$$

Ainsi, les domaines de définition de tous les canaux internes sont identiques, et chaque canal permet de faire suivre la valeur des canaux d'entrée.

**Exécutions (comportements, runs).** L'état du système à un instant fixé est la valeur des variables, et est décrit par une fonction d'interprétation

$$s : V \to \bigcup_{z \in V} D(z)$$

telle que $s(z) \in D(z)$ pour tout $z \in V$. On désigne par $s[Z]$ la restriction de $s$ à $Z \subseteq V$. On remarque qu'on peut identifier $s[Z]$ à un tuple indexé par $Z$. On définit un run (infini) comme une suite d'états $s_0 \cdot s_1 \dots s_n \dots$.

**Calculs et délai.** On travaille dans un cadre synchrone et temps discret. À chaque instant, l'environnement fournit de nouvelles valeurs pour les variables d'entrée. Chaque processus calcule alors de nouvelles valeurs qu'il écrit sur ses variables de sortie, pour obtenir l'état atteint à l'instant suivant. Nous considérerons deux variantes pour ce calcul :

1. sémantique *sans délai* (ou 0-délai). Dans cette sémantique, chaque processus reçcoit en entrée à l'instant $i$ l'ensemble des valeurs déjà calculées par ses prédécesseurs (dans le graphe des processus) *à l'instant $i$ également*. Ceci suppose le graphe des processus (ou des variables) acyclique, puisqu'entre l'instant $i$ et l'instant $i + 1$, le calcul se propage des entrées vers les sorties. L'environnement fournit donc des valeurs pour les variables d'entrée, qui sont utilisées par les processus minimaux du graphe pour eux-même calculer leurs sorties, etc.

2. sémantique *avec délai* (ou 1-*délai*). Dans cette sémantique, les valeurs des variables sont supposées connues à l'instant initial. Pour passer de l'instant $i$ à l'instant $i+1$, chaque processus utilise dans son calcul les valeurs disponibles à l'instant $i$ sur les variables qu'il lit et assigne les variables sur lesquelles il peut écrire. Autrement dit, dans la sémantique avec délai, les valeurs lues par chaque processus proviennent de ce qui a été écrit à l'instant précédent.

Une exécution particulière sur une telle architecture sera donc un run $s_0 \cdot s_1 \dots s_n \dots$, identifiable à un mot sur l'alphabet $\prod_{z \in V} D(z)$.

**Spécifications.** En plus de l'architecture, on se donne une spécification qui décrit un ensemble de comportements. La spécification peut être donnée par exemple par une formule du $\mu$-calcul, en CTL* ou en LTL sur l'alphabet des différents tuples de valeurs possibles formés par les variables du système, i.e. $\Sigma = \prod_{z \in Z \subseteq V} D(z)$. La spécification porte donc sur les mots constitués par les runs du système. Le langage de spécification est

1. *externe* s'il ne permet de spécifier que sur la valeur des variables externes. En particulier, la validité d'une formule sur une exécution ne dépend que de la projection de l'exécution sur $Z = X \cup Y$.
2. *interne* s'il permet de spécifier sur la valeur de toutes les variables : $Z = V$.

**Stratégies.** On définit à présent les programmes associés à l'écriture de chaque variable, programmes qu'on identifie à des stratégies dans un jeu environnement contre l'équipe des processus. Pour un graphe de variables $(V, A)$, soit $z \in V$. On note

$$R(z) = A^{-1}(z)$$

son ensemble de lecture. Une stratégie pour la variable $z$ est une fonction

$$g_z : (\prod_{t \in R(z)} D(t))^+ \to D(z)$$

si l'on est en 0-délai. On dit dans ce cas-là que le run $s_0 s_1 \cdots$ est un $g_z$-*run* si, pour tout $i \geq 0$,

$$s_i(z) = g_z(s_0[R(z)] \cdot \ldots \cdot s_i[R(z)]).$$

Dans une sémantique 1-délai, on définit également la stratégie lorsque le canal d'entrée est vide, ce qui nous donne en fait la valeur de $z$ à l'instant

initial. Une stratégie pour $z$ est donc une fonction

$$g_z : (\prod_{t \in R(z)} D(t))^* \to D(z).$$

On dit que le run $s_0 s_1 \cdots$ est un $g_z$-*run* dans une sémantique 1-délai si, pour tout $i \geq 1$,

$$\begin{aligned} s_i(z) &= g_z(s_0[R(z)] \cdot \ldots \cdot s_{i-1}[R(z)]) \text{ et} \\ s_0(z) &= g_z(\epsilon). \end{aligned}$$

Par la suite, on utilisera la notation $\overline{s}_i[X]$ pour $s_0[X] \cdots s_i[X]$, c'est-à-dire la version « avec mémoire » de l'interprétation des variables.

On appellera *stratégie distribuée* un tuple $g = (g_z)_{z \in T \cup Y}$ de stratégies locales (qu'on notera parfois sous forme ensembliste). Étant donné une stratégie distribuée $g = (g_z)_{z \in T \cup Y}$, on dit qu'un run est un $g$-*run* si c'est un $g_z$-run pour tout $z \in T \cup Y$. On dit qu'une telle stratégie $g$ *réalise* une propriété LTL $\phi$ si tout $g$-run satisfait $\phi$. On dira aussi que $g$ est *gagnante* (vis-à-vis de la propriété $\phi$, qui sera souvent sous-entendue),

**Le problème de synthèse.** On définit enfin le problème de synthèse (ou de réalisabilité) de la façon suivante : étant donnée une architecture $\mathfrak{A}$, une formule de spécification $\psi$, on cherche s'il existe des stratégies $(f_z)_{z \in T \cup Y}$ pour les variables internes et de sortie, telles que tous les runs respectant ces stratégies satisfassent la spécification $\psi$, et si c'est le cas, on veut synthétiser de telles stratégies à mémoire finie.

**Exemple.** Sur l'architecture représentée sur la figure 2.4, le problème de synthèse distribuée 0-délai va donc être, étant donnée une formule $\phi$ sur l'alphabet $\Sigma = D(x_0) \times D(x_1) \times D(y_0) \times D(y_1)$, de trouver les programmes/stratégies

$$\begin{aligned} g_{t_0} &: D(x_0)^+ \to D(t_0) \\ g_{t_1} &: D(x_0)^+ \to D(t_1) \\ g_{t_2} &: D(x_1)^+ \to D(t_2) \\ g_{t_3} &: D(t_0)^+ \to D(t_3) \\ g_{t_4} &: (D(t_1) \times D(t_2))^+ \to D(t_4) \\ g_{t_5} &: (D(t_1) \times D(t_2))^+ \to D(t_5) \\ g_{y_0} &: (D(t_3) \times D(t_4))^+ \to D(y_0) \\ g_{y_1} &: D(t_5)^+ \to D(y_1) \end{aligned}$$

telles que pour tout run respectant ces stratégies locales, c'est-à-dire tout run $s_0 \cdot s_1 \cdots$ tel que pour tout $i$, $s_i(z) = g_z(\overline{s}_i(R(z)))$, $s_0[X \cup Y] \cdot s_1[X \cup Y] \cdots$ satisfait $\phi$.

### 2.1.2 Constructions sur les arbres

Les stratégies se représentent tout naturellement sous forme d'arbres. Dans le cas 1-délai, une stratégie est un arbre par définition. Dans le cas 0-délai, la stratégie n'est pas définie sur le mot vide. On étend une telle stratégie $f_z : (\prod_{t \in R(z)} D(t))^+ \to D(z)$ en un arbre en posant $f_z(\epsilon) = \bot$, où $\bot \notin X \cup Y \cup Z$. De la même façon, un ensemble de runs se représente par un arbre.

On introduit à présent des constructions sur les arbres, qui permettent en particulier de passer d'un arbre de stratégie à l'arbre des comportements induits par la stratégie.

- Pour $t$ un $(X, Y)$-arbre, on note $xray(t)$ le $(X, (X \cup \bot) \times Y)$-arbre tel que $xray(t)(\epsilon) = (\bot, t(\epsilon))$, et, pour tout $\sigma \in X^*$, $x \in X$, $xray(t)(\sigma \cdot x) = (x, t(\sigma \cdot x))$.
- Pour $X$, $Y$ et $Z$ ensembles finis, et pour $t$ un $(X, Y)$-arbre, on construit $wide_Z(t)$ un $(X \times Z, Y)$-arbre tel que pour tout $(\sigma_X, \sigma_Z) \in (X \times Z)^*$, $(wide_Z(t))(\sigma_X, \sigma_Z) = t(\sigma_X)$. Les étiquettes de $wide_Z(t)$ ne tiennent donc pas compte de la valeur dans $Z$ du nœud.
- Pour $t$ un $(X, Y)$-arbre, on note $delay(t)$ le $(X, Y)$-arbre tel que pour tout $\sigma \in X^*$, $x \in X$, $delay(t)(\sigma \cdot x) = t(\sigma)$ et $delay(t)(\epsilon) = \epsilon$.

On remarque que dans le cadre 0-délai, on peut exprimer un arbre de run $s$ suivant la stratégie $f$ en fonction de l'arbre de $f$ ainsi : $s = xray(f)$. En effet, les nœuds de l'arbre de stratégie étant les variables d'entrée, l'arbre de $f$ donne les valeurs sur les variables de sortie aux mêmes instants. Pour lire le run sur les étiquettes de l'arbre, il faut les enrichir avec la valeur des variables d'entrée, donc des nœuds. Dans le cas 1-délai, on a $s = xray(delay(f))$.

### 2.1.3 Rappels sur les automates d'arbres

Cette analogie entre les arbres et les stratégies mène tout naturellement à utiliser des techniques d'automates d'arbres alternants ou non déterministes pour résoudre le problème de synthèse distribuée. Les automates d'arbres alternants ont été introduits dans [17] comme une généralisation des automates non déterministes. Nous présenterons ici ces derniers comme un cas particulier des premiers.

Un automate d'arbres alternant $\mathcal{A} = (Y, Q, q_0, \delta, \alpha)$ accepte, pour un ensemble $X$ donné, des $(X, Y)$-arbres complets. L'ensemble $Q$ est l'ensemble des états de l'automate, avec $q_0$ l'état initial, $\delta$ la fonction de transition, et $\alpha$ une condition d'acceptation (qui définit un sous-ensemble de $Q^\omega$). La fonction $\delta : Q \times Y \to \mathcal{B}^+(X \times Q)$ (où $\mathcal{B}^+(X \times Q)$ est l'ensemble des formules booléennes positives sur $X \times Q$) associe à un état de l'automate et à une lettre

une formule donnant les nouvelles configurations possibles de l'automate. Par exemple, si $X = \{0, 1\}$, $\delta(q, y) = (0, q_1) \wedge (0, q_2) \vee (0, q_2) \wedge (1, q_2) \wedge (1, q_3)$ signifie que lorsque l'automate est dans l'état $q$ et lit la lettre $y$, il peut soit envoyer deux copies, dans l'état $q_1$ et dans l'état $q_2$, dans la direction 0, soit envoyer une copie dans l'état $q_2$ dans la direction 0, et deux copies, dans l'état $q_2$ et dans l'état $q_3$ dans la direction 1.

Un run sur un automate d'arbres alternant $\mathcal{A}$ sur un $(X, Y)$-arbre $t$ est un $(D, X^* \times Q)$-arbre $\rho$, où $D$ est un ensemble fini. Chaque nœud de $\rho$ correspond à un nœud de $t$, mais plusieurs nœuds de $\rho$ peuvent correspondre au même nœud de $t$. Un nœud de $\rho$ étiqueté par $(x, q)$ définit une copie de l'automate se trouvant dans l'état $q$ et lisant le nœud $x$ de $t$ (c'est-à-dire $t(x)$). Un nœud de l'arbre de run et ses enfants doivent satisfaire la fonction de transition : $\bigwedge_{d \in D} \rho(\sigma_D \cdot d) \vDash \delta(\rho(\sigma_D)_{|Q}, t(\rho(\sigma_D)_{|X^*}))$.

Un chemin infini est étiqueté par la suite infinie détats visités. Un run est acceptant si tous ses chemins infinis satisfont la condition d'acceptation $\alpha$.

Il existe d'autres définitions des automates d'arbres alternants, les représentant comme des jeux (les états étant divisés en deux groupes pour représenter les joueurs universel et existentiel) (voir par exemple [9]).

Un automate d'arbres non déterministe est un automate d'arbres alternant dans lequel la fonction de transition est définie de telle sorte qu'elle n'envoie qu'une seule copie de l'automate dans chaque direction. Un run d'un automate non déterministe sur un $(X, Y)$-arbre $t$ peut être vu comme un arbre $(X, Y \times Q)$ qui serait l'arbre $t$ auquel on a rajouté dans l'étiquette de chaque nœud l'état visité par l'automate à ce moment.

On rappelle le théorème suivant :

**Théorème 2.1.1 (*[18]*)**

*Un automate d'arbres alternant de Rabin avec $m$ états et $k$ paires peut être transformé en un automate non déterministe équivalent, avec $m^{O(mk)}$ états et $O(mk)$ paires.*

## 2.2 Indécidabilité du problème en général

Pnueli et Rosner ont exhibé dans [20] plusieurs architectures pour lesquelles le problème de synthèse distribuée était indécidable pour des spécifications LTL. La preuve se base sur une réduction du problème de l'arrêt d'une machine de Turing. L'architecture indécidable de base est l'architecture totalement déconnectée, c'est-à-dire constituée de deux processus ayant chacun une variable d'entrée et une variabled de sortie, et ne communiquant pas

entre eux (voir figure 2.7). La preuve repose sur une réduction du problème de l'arrêt d'une machine de Turing.

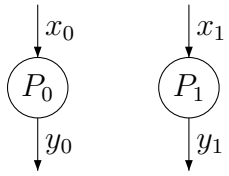



FIG. 2.7 – Architecture $\mathfrak{A}_0$

En effet, étant donnée $M$ une machine de Turing quelconque, on peut définir une spécification LTL $\Phi_M$ telle que $\Phi_M$ est réalisable sur $\mathfrak{A}_0$ si et seulement si $M$ s'arrête sur bande vide. Plus précisément, supposons que les domaines $D(x_0)$ et $D(x_1)$ valent $\{0, 1\}$ et que les domaines $D(y_0)$ et $D(y_1)$ permettent de coder les différentes configurations de $M$ ainsi qu'un symbole spécial \$. Alors on peut écrire une formule de spécification LTL pour laquelle la seule réalisation possible soit pour chaque processus d'écrire la suite de configurations de la machine de Turing $M$. Nous allons la décrire informellement. Cette spécification est de la forme

$$\Phi_M = \psi_1 \rightarrow (\psi_2 \wedge \psi_3) \tag{2.2}$$

avec :

- la formule $\psi_1$ impose que les canaux d'entrée (i.e. l'environnement) n'écrivent un symbole 1 que si les deux canaux de sortie ont écrit un symbole \$ à l'instant précédent, et que la différence entre le nombre de 1 écrit sur chaque fil n'excède jamais 1. Par la forme de $\Phi_M$, l'environnement perd s'il viole cette contrainte.
- la deuxième formule ($\psi_2$) assure
  - que chaque canal de sortie n'écrive que des \$ jusqu'à ce que son canal d'entrée écrive 1, moment à partir duquel il écrit le code de la première configuration de la machine de Turing $M$, suivi d'une suite non vide de \$.
  - que, par la suite, à chaque fois qu'un processus reçoit un 1 sur son canal d'entrée, il écrive sur son canal de sortie une configuration légale de la machine de Turing $M$,
  - que, à chaque fois qu'il y a un 1 sur *les deux canaux d'entrée*, les configurations $C_0$ et $C_1$ sorties sur les canaux $y_0$ et $y_1$ respectent les conditions suivantes :
    - $C_0 = C_1$ si le nombre de 1 est le même sur les deux canaux d'entrée,

- $C_0 \vdash C_1$ si il y a un 1 de plus sur le canal d'entrée $x_0$ que sur le canal d'entrée $x_1$,
- $C_1 \vdash C_0$ si il y a un 1 de plus sur le canal d'entrée $x_1$ que sur le canal d'entrée $x_0$,

avec $C_0 \vdash C_1$ signifiant que les configurations $C_0$ et $C_1$ sont deux configurations successives de la machine de Turing $M$.

- et enfin, $\psi_3$ impose que si il y a un nombre infini de 1 sur les canaux d'entrée, alors sur les canaux de sortie sera codée une configuration finale de la machine de Turing.

On peut montrer que la seule stratégie distribuée réalisant la spécification $\psi_{0,M} = \psi_1 \rightarrow \psi_2$ fait écrire des \$ sur chaque canal de sortie jusqu'à ce que le canal d'entrée correspondant écrive un 1. Puis, quand on lit le $k$-ième 1 sur le canal d'entrée, elle fait écrire sur le canal de sortie la $k$-ième configuration de la machine de Turing $M$. C'est également la seule stratégie gagnante pour $\Phi_M$, qui assure en plus la convergence de $M$. On a donc le résultat suivant :

**Lemme 2.2.1**

*La spécification $\Phi_M$ est réalisable sur $\mathfrak{A}_0$ si et seulement si $M$ s'arrête sur bande vide.*

On a donc l'indécidabilité du problème de synthèse sur l'architecture $\mathfrak{A}_0$.

On constate que la partie importante de la formule $\Phi_M$ est la sous-formule $\psi_2$ qui impose aux deux configurations sur les deux canaux de sortie d'être « ordonnées » correctement, alors que les stratégies que l'on cherche ne peuvent pas savoir ce qu'a reçu l'autre processus. On voit là la source principale d'indécidabilité : une spécification *globale* est trop forte pour des stratégies *locales*.

## 2.3 Décidabilité du pipe-line

La preuve de décidabilité de cette architecture a été présentée dans [20] pour les spécifications LTL, puis reprise et généralisée dans [11] pour des spécifications CTL* qui s'autorisent, contrairement au papier originel, à contraindre les variables internes (i.e. les variables dans $T$). En effet, O. Kupferman et M. Vardi s'intéressaient dans ce papier à la *décidabilité* des architectures. Or, si l'on sait décider le problème de synthèse dans le cas de formules spécifiant toutes les variables du système, on sait également le décider pour des formules ne s'autorisant à contraindre que les variables d'entrée et de sortie, le second ensemble de formules étant inclus dans le premier.

Nous nous plaçons ici, comme dans [11], dans une sémantique 1-délai, et avec des spécifications CTL*.

### 2.3.1 Préliminaires

On considère que si deux variables $y$ et $z$ ont le même ensemble de lecture et sont en lecture des mêmes variables ($R(y) = R(z)$ et $A(y) = A(z)$), alors on les fusionne en une seule variable $x$ dont le domaine de définition est $D(x) = D(y) \times D(z)$. On définit donc une architecture pipe-line comme une architecture dans laquelle chaque variable n'a qu'une variable en lecture : pour tout $z \in Y \cup T$, $|R(z)| = 1$. Intuitivement, dans un pipe-line, l'information est transmise de processus en processus, en droite ligne depuis le canal d'entrée jusqu'au canal de sortie. Soit une formule $\psi$ de CTL* et une architecture distribuée de type pipe-line.

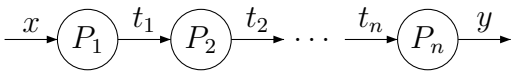



FIG. 2.8 – L'architecture pipe-line

Soit $x$ la variable d'entrée et $D(x)$ l'ensemble de ses valeurs, et soit $\{t_i\}$ l'ensemble des variables dans $T$ telles que $(x, t_1) \in A$, et pour tout $1 < i < n$, $(t_{i-1}, t_i) \in A$, $(t_n, y) \in A$. On note $D(o) = \bigcup_{t \in T} D(t) \cup D(y)$.

Un arbre de run (ou de comportement) $t$ est donc un $D(x)$-arbre $(D(x) \times D(o))$-étiqueté $t : D(x)^* \to (D(x) \times D(o))$ associant à chaque entrée du système la valeur sur toutes les variables à cet instant. On rappelle que, dans une architecture avec délai, l'arbre de run $t : D(x)^* \to D(x) \times D(y) \cup \bot$ induit par un arbre de stratégie $f : D(x)^* \to D(y)$ est :

$$\begin{aligned} t(\epsilon) &= \bot \\ t(s) &= s, f(\epsilon) \text{ pour tout } s \in D(x) \\ t(s_1 \dots s_n) &= s_n, f(s_1 \cdots s_{n-1}) \text{ pour tous } (s_1, \dots, s_n) \in D(x)^n \end{aligned}$$

On définit la composition de deux stratégies dans le cas du pipe-line avec une sémantique 1-délai. Il s'agit de calculer par une seule fonction (la fonction composée) les valeurs en sortie de deux processus en fonction des valeurs en entrée du premier, valeurs tenant compte du délai dans la transmission des informations (voir figure 2.9). C'est-à-dire que la stratégie composée va calculer la valeur de la variable $y$ en fonction de l'entrée sur $x$ ainsi que la valeur de la variable $z$ en fonction des entrées sur la variable $y$ *jusqu'à l'instant précédent*. Ainsi les comportements induits par les stratégies distinctes ou par la stratégie composée sont identiques.

On note formellement en se basant sur l'abstraction des arbres :

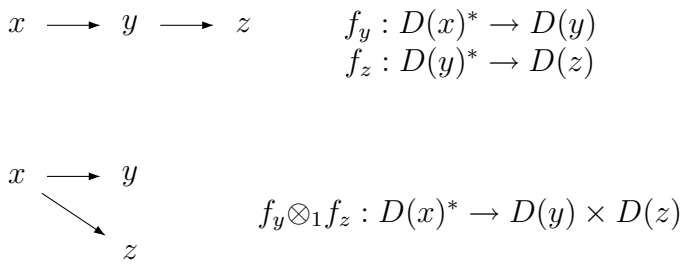



FIG. 2.9 – Vision de l'architecture pour une stratégie composée

**Définition 2.3.1 (*Composition de stratégies avec délai*)**

*Soit $f : X^* \to Y$ et $f' : Y^* \to Z$ deux arbres de stratégie. On définit leur composition $f \otimes_1 f' : X^* \to Y \times Z$ comme suit :*

$$\begin{aligned} f \otimes_1 f'(\epsilon) &= \big(f(\epsilon), f'(\epsilon)\big) \\ f \otimes_1 f'(x_1 \cdots x_n) &= \big(f(x_1 \cdots x_n), f'(f(\epsilon) f(x_1) \cdots f(x_1 \cdots x_{n-1}))\big) \end{aligned}$$

On définit également un autre type de composition pour le pipe-line : étant donné un arbre de run $t : D(y)^* \to D(y) \times D(z) \cup \bot$ et un arbre de stratégie $f : D(x)^* \to D(y)$, on définit le comportement composé de $t'$ tel que $xray(t') = t$ et de $f$ sur $y$ et $z$ comme le comportement sur $y$ et $z$ en fonction de $x$, basé sur le comportement sur $y$ induit par la stratégie $f$ et le comportement sur $z$ qui en découle. Formellement on note :

**Définition 2.3.2 (*Composition de stratégie et de runs*)**

*Soit $f : X^* \to Y$ un programme et $t : Y^* \to Z$ un comportement. On définit leur composition $f \oplus_1 t : X^* \to Y \times Z$ ainsi :*

$$\begin{aligned} f \oplus_1 t(\epsilon) &= (\bot, \bot) \\ f \oplus_1 t(x_1 \cdots x_n) &= \big(f(x_1 \cdots x_{n-1}), t(f(\epsilon) \cdots f(x_1 \cdots x_{n-1}))\big) \end{aligned}$$

On va maintenant exposer des théorèmes généraux sur différentes constructions et transformations d'automates d'arbres, théorèmes qui vont nous être utiles pour décider la réalisabilité.

**Théorème 2.3.3 (*[13]*)**

*Pour toute $\psi$ formule de CTL*, on peut construire un automate d'arbres $\mathcal{A}_\psi$ tel que $\mathcal{A}_\psi$ reconnaisse exactement les arbres satisfaisant $\psi$.*

L'automate ainsi construit reconnaît des arbres dans lesquels la structure est apparente : les nœuds sont également étiquetés par leur direction : ce sont exactement les arbres de runs satisfaisant $\psi$.

**Théorème 2.3.4 (*[12]*)**

*Pour tout $\mathcal{A}$ automate alternant sur des $X$-arbres $(X \times Y)$-étiquetés, on peut construire $\mathcal{A}'$ automate alternant sur des $X$-arbres $Y$-étiquetés tel que $t \in \mathcal{A}'$ si et seulement si $xray(t) \in \mathcal{A}$.*

Le problème de synthèse de programmes dans le cas du pipe-line est résolu en se basant sur l'idée suivante : il s'agit de considérer le pipe-line au départ comme une architecture singleton, avec plusieurs canaux de sortie, puis de décider si la stratégie gagnante pour l'architecture singleton peut être distribuée de façon à correspondre à la structure du pipe-line. Pour cela, deux approches similaires peuvent être adoptées. La première, présentée par [11], est ce que nous avons appelé la vision « stratégies ». Elle se base sur la définition de la composition de stratégies 2.3.1. On calcule une stratégie globale pour toutes les variables du pipe-line, puis, inductivement, on décompose cette stratégie initiale en stratégies spécifiques à chaque variable de l'architecture, par des constructions d'automates d'arbres. La deuxième approche a été développée au cours du stage et est très proche de la précédente. La différence réside dans le fait qu'au lieu de calculer une stratégie globale pour les variables du pipe-line, on calcule les *runs* du pipe-line satisfaisant la spécification, puis, en utilisant la définition 2.3.2, on extrait inductivement les différentes stratégies pour les variables qui, composées avec le comportement du reste du pipe-line, vont induire un comportement accepté. Nous présentons dans les parties suivantes ces deux approches successivement.

### 2.3.2 Vision « stratégies »

La construction pour décider de l'existence d'une stratégie distribuée satisfaisant $\psi$ se base principalement sur le théorème suivant :

**Théorème 2.3.5 (*[11]*)**

*Soit un automate d'arbres non déterministe $\mathcal{A}$ acceptant des $X$-arbres $Y \times Z$-étiquetés. On peut construire un automate $\mathcal{A}'$ alternant reconnaissant des $Y$-arbres $Z$-étiquetés $T$ tels qu'il existe un arbre de stratégie $f$ tel que la composition $f \otimes_1 T$ soit acceptée par $\mathcal{A}$. On note $\mathcal{L}(\mathcal{A}') = shape_{\otimes_Y}(\mathcal{L}(\mathcal{A}))$.*

**Démonstration :** (**Idée de preuve**) Soit $\mathcal{A} = (Q, Y \times Z, q_0, \delta, \alpha)$ automate non déterministe avec Q l'ensemble d'états, $Y \times Z$ l'ensemble d'étiquetage,

$q_0$ l'état initial, $\delta$ la fonction de transition et $\alpha$ la condition d'acceptation. On construit l'automate du théorème ainsi : $\mathcal{A}' = (Q, Z, q_0, \delta', \alpha)$ avec

$$\delta'(q,z) = \bigvee_{\substack{y \in Y \\ ((s_1,x_1),\ldots,(s_{|X|},x_{|X|})) \models \delta(q,(y,z))}} (s_1,y) \wedge \ldots \wedge (s_{|X|},y).$$

L'idée est que l'automate $\mathcal{A}'$ mime sur un arbre $t$ le comportement qu'aurait $\mathcal{A}$ sur la stratégie composée d'un certain programme $f$ qu'on devine et $t$. Il accepte donc un arbre $t$ si et seulement si on a pu deviner un arbre $f$ tel que la composée selon 2.3.1 de $f$ et de $t$ est acceptée par $\mathcal{A}$.

- Soit $t$ un $Y$-arbre $Z$-étiqueté accepté par $\mathcal{A}'$. On a donc un run acceptant de $\mathcal{A}'$ sur $t$. La racine du run est étiquetée par $(q_o, \epsilon)$ et l'automate lit la lettre $t(\epsilon)$. On devine $y_0 = f(\epsilon)$ et on se comporte selon $\mathcal{A}$ sur $(q_0, (f(\epsilon), t(\epsilon))$ c'est-à-dire selon $\delta(q_0, f \otimes_1 t(\epsilon))$. On va donc dans les états $q_1, \ldots, q_{|X|}$ tels que $((q_1,x_1),\ldots,(q_{|X|},x_{|X|})) \models \delta(q_0, f(\epsilon), t(\epsilon))$ et dans la direction $f(\epsilon)$. Toutes les copies de l'automate vont donc lire $t(f(\epsilon))$. Soit une copie de l'automate se trouvant dans l'état $q_i$. Dans l'automate $\mathcal{A}$, cela signifie que l'on est allé dans la direction $x_i$ dans l'arbre composé. La lettre $y$ que $\mathcal{A}'$ va donc deviner correspondra à $f(x_i)$. L'automate $\mathcal{A}'$ se comporte suivant $\delta(q_i, (f(x_i), t(f(\epsilon))) = \delta(q_i, f \otimes_1 t(x_i))$. Il choisit l'ensemble d'états $(s_1, \ldots, s_{|X|})$ tel que $((s_1,x_1),\ldots,(s_{|X|},x_{|X|})) \models \delta(q_i, (f(x_i), t(f(\epsilon)))$ et envoie toutes les copies dans la direction $f(x_i)$. Par suite, une copie de $\mathcal{A}'$ dans un état $s$ (qui était associé à une direction $x$) et dans un nœud $y$ correspondant à $f(\sigma \cdot x_1)$ va lire $t(f(\sigma \cdot x_1))$, deviner $f(\sigma \cdot x_1 \cdot x)$ et se comporter en suivant $\delta(s, (f(\sigma \cdot x_1 \cdot x), t(f(\sigma \cdot x_1))))$, qui est bien égal à $f \otimes_1 t(\sigma \cdot x_1 \cdot x)$. Le run de $\mathcal{A}'$ correspond bien à un run de $\mathcal{A}$ sur la composée d'une stratégie $f$ et de $t$. De plus, la suite d'états dans l'arbre de run étant la même, le run de $\mathcal{A}$ est acceptant.
- Réciproquement, soit $t$ un $Y$-arbre $Z$-étiqueté et $f$ un $X$-arbre $Y$-étiqueté tel que $f_1 = f \otimes_1 t \in \mathcal{L}(\mathcal{A})$. Soit un run acceptant de $\mathcal{A}$ sur $f_1$. Alors on construit un run acceptant de $\mathcal{A}'$ sur $t$ en devinant les bonnes valeurs de $f$ et en choisissant les ensembles d'états qui étiquettent le run de $\mathcal{A}$.

Intuitivement, l'automate $\mathcal{A}'$ fonctionne ainsi : lorsqu'il lit une valeur $z$ sur l'arbre, il devine la valeur que le processus précédent est en train d'écrire (programme $f$ qu'on devine) et va dans cette direction car c'est la valeur qu'on lira à l'instant suivant (prise en compte du délai). Les états dans lesquels se trouvent les différentes copies de l'automate à l'instant suivant mémorisent la valeur qu'a lue le processus dont on devine la stratégie et influent sur la valeur de $f$ qu'on va deviner ($f(\sigma \cdot x_i)$ si on est dans l'état associé à la direction $x_i$). ■

L'algorithme de décidabilité fonctionne comme suit :

- On construit un automate $\mathcal{A}_\psi$ qui reconnaît l'ensemble des arbres de comportement satisfaisant la formule $\psi$ (en se servant du théorème 2.3.3). Cet automate contient les valeurs des inputs sur ses étiquettes.
- On construit l'automate $\mathcal{A}_0$ qui accepte les arbres $t$ tels que $xray(t) \in \mathcal{L}(\mathcal{A}_\psi)$. C'est-à-dire qu'il accepte uniquement les arbres acceptés par $\mathcal{A}_\psi$ pour lesquels la partie de l'étiquette correspondant à l'input est la même que le noeud étiqueté, en supprimant cette information devenue redondante. L'automate $\mathcal{A}_0$ accepte donc des arbres $t : D(x)^* \to D(o)$ (en se servant du théorème 2.3.4).
- On construit l'automate $\mathcal{A}'_0$ qui accepte les arbres $t$ tels que $delay(t) \in \mathcal{L}(\mathcal{A}_0)$. $\mathcal{A}'_0$ accepte donc les stratégies globales qui induisent un comportement satisfaisant $\psi$. En effet $t \in \mathcal{L}(\mathcal{A}'_0)$ si et seulement si $xray(delay(t)) \in \mathcal{L}(\mathcal{A}_\psi)$. Si il existe une stratégie distribuée acceptée par $\mathcal{A}'_0$, alors $\psi$ est réalisable. On procède par induction.
- Pour $1 \leq i \leq n-1$
  - On construit $\mathcal{A}_i$ automate non déterministe équivalent à $\mathcal{A}'_{i-1}$.
  - On construit $\mathcal{A}'_i = shape_{\otimes D(t_i)}$ automate alternant acceptant des $D(t_i)$-arbres $D(t_{i+1}) \times \cdots \times D(t_n)$-étiquetés $T$, programmes pour les variables $t_{i+1}, \cdots t_n$ tels qu'il existe un programme $f$ pour $t_i$ (un $D(t_{i-1})$-arbre $D(t_i)$-étiqueté) tel que $f \otimes_1 T \in \mathcal{L}(\mathcal{A}_i)$ (en utilisant le théorème 2.3.5).
- $\mathcal{L}(\mathcal{A}_n)$ est non vide si et seulement si il existe une stratégie $f_y$ pour $y$ telle qu'il existe une stratégie $f_n$ pour $t_n$ telle qu'il existe ... une stratégie $f_1$ pour $t_1$ telle que $f_1 \otimes_1 \cdots \otimes_1 f_n \otimes_1 f_y$ soit une stratégie gagnante pour $\psi$, c'est-à-dire que tout $f_1 \otimes_1 \cdots \otimes_1 f_n \otimes_1 f_y$-run satisfasse $\psi$.

On a donc $\mathcal{L}(\mathcal{A}_{n-1})$ est non vide si et seulement si $\psi$ est réalisable.

### 2.3.3 Vision « comportements »

Nous exposons à présent l'approche considérant les comportements du système. Pour cela, nous avons besoin d'un théorème similaire au théorème 2.3.5 mais permettant d'extraire des stratégies d'arbres de comportements globaux. Nous avons développés cette approche car elle nous semblait au premier abord plus naturelle. L'étape initiale de [11] qui consiste à sélectionner les stratégies globales gagnantes conduit en fait à considérer le pipe-line comme une architecture singleton pour lequel on calcule une stratégie. Intuitivement, il nous semblait plus naturel de considérer quels étaient les comportements acceptants (puisque $\psi$ contraint les comportements), puis d'en déduire les stratégies pour les processus.

**Théorème 2.3.6**

*Soit un automate d'arbres non déterministe $\mathcal{A}$ acceptant des $X$-arbres $Y \times Z$-étiquetés. On peut construire un automate $\mathcal{A}'$ alternant reconnaissant des $Y$-arbres $Z$-étiquetés $t$ tels qu'il existe un arbre de stratégie $f$ tel que la composition $f \oplus_1 t$ soit acceptée par $\mathcal{A}$. On note $\mathcal{L}(\mathcal{A}') = shape_{\oplus_Y}(\mathcal{L}(\mathcal{A}))$.*

**Démonstration : (Idée de preuve)** Soit $\mathcal{A} = (Q, Y \times Z, q_0, \delta, \alpha)$ automate non déterministe avec Q l'ensemble d'états, $Y \times Z$ l'ensemble d'étiquetage, $q_0$ l'état initial, $\delta$ la fonction de transition et $\alpha$ la condition d'acceptation. On construit l'automate du théorème ainsi : $\mathcal{A}' = (Q \times Y, Z, (q_0, \bot), \delta', \alpha \times Y)$ avec

$$\delta'((q,y),z) = \bigvee_{\substack{y_1 \in Y \\ ((s_1,x_1),\ldots,(s_{|X|},x_{|X|})) \models \delta(q,(y,z))}} ((s_1,y_1),y_1) \wedge \ldots \wedge ((s_{|X|},y_1),y_1).$$

Soit $t_1 \in \mathcal{L}(\mathcal{A}')$. On a donc un run acceptant de $\mathcal{A}'$ sur $t_1$. La racine du run est étiquetée par $(q_0, \bot)$ et lit $t_1(\epsilon)$. L'automate $\mathcal{A}'$ devine une valeur $y_1$ (qui va être égale à $f(\epsilon)$) et choisit un ensemble d'états $(q_1, \ldots q_{|X|})$ tels que $(q_1, x_1), \ldots, (q_{|X|}, x_{|X|}) \models \delta((q_0, \bot), t_1(\epsilon))$. L'automate se trouve maintenant dans un nœud étiqueté par $(q_i, y_1)$ et lit $t_1(y_1) = t_1(f(\epsilon))$. On se comporte donc comme $\mathcal{A}$ sur $(f(\epsilon), t_1(f(\epsilon))) = t(x)$, pour $x \in X$. Les fils de ce nœud dans l'arbre de run sont étiquetés par les états $(s_j, y_2)$ avec $y_2 = f(x_i)$ et les $s_j$ tels que $(s_1, x_1) \wedge \ldots \wedge (s_{|X|}, x_{|X|}) \models \delta(q_i, y_1, t_1(y_1)) = \delta(q_i, t(x))$. Par suite, lorsque l'arbre du run de $\mathcal{A}'$ est dans un nœud étiqueté par $(q, y = f(\sigma_X))$, il lit $t_1(\sigma_Y \cdot y)$ (pour $\sigma_Y \in Y^*$). Ses fils dans l'arbre de run seront étiquetés par les différents $q_x$ et $y'(= f(\sigma_X \cdot x))$ tels que $\bigwedge_{x \in X}(q_x, x) \vDash \delta(q, y, t_1(\sigma_Y \cdot y)) = \delta(q, f(\sigma_X), t_1(f(\sigma_X))$. Ainsi, à ce run de $\mathcal{A}'$ correspond un run de $\mathcal{A}$ sur la composition $\oplus_1$ d'un certain arbre (de stratégie) $f$ qu'on devine et de $t_1$. Les deux runs étant étiquetés par les mêmes états, le run de $\mathcal{A}$ est acceptant, et donc il existe un arbre $f$ tel que $f \oplus_1 t_1$ soit accepté par $\mathcal{A}$.

Réciproquement, soient $f$ et $t_1$ tels que $f \oplus_1 t_1 \in \mathcal{L}(\mathcal{A})$. On a alors un run acceptant de $\mathcal{A}$ sur $f \oplus_1 t_1$. On construit un run acceptant de $\mathcal{A}'$ sur $t_1$ en devinant la stratégie $f$ et en choisissant les ensembles d'états étiquetant le run acceptant de $\mathcal{A}$.

On a donc $\mathcal{L}(\mathcal{A}') = shape_{\oplus_Y}(\mathcal{L}(\mathcal{A}))$. ■

L'algorithme de décision est le suivant :

– On construit $\mathcal{A}_\psi$ l'automate reconnaissant les arbres de comportement globaux satisfaisant $\psi$.

- On construit $\mathcal{A}_0$ acceptant les arbres $t$ tels que $xray(t) \in \mathcal{L}(\mathcal{A}_\psi)$. On procéde ensuite par induction :
- $\forall 1 \leq i \leq n-1$
  - On construit $\mathcal{A}'_{i-1}$ automate non déterministe équivalent à $\mathcal{A}_{i-1}$.
  - On construit $\mathcal{A}_i = shape_{\oplus D(t_i)}(\mathcal{L}(\mathcal{A}'_{i-1})$. $\mathcal{A}_i$ accepte des arbres de comportement $t : D(t_i)^* \to D(t_{i+1}) \cup \ldots \cup D(t_n)$ tels qu'il existe un arbre de stratégie $f : D(t_{i-1})^* \to D(t_i)$ tel que $f \oplus_1 t \in \mathcal{L}(\mathcal{A}'_{i-1})$
- $\mathcal{A}_{n-1}$ accepte des arbres de comportement $t$ pour $t_n$ tels qu'il existe une stratégie $f_{n-1}$ pour $t_{n-1}$, ... une stratégie $f_1$ pour $t_1$ telles que $f_1 \oplus_1 (\ldots (f_{n-1} \oplus_1 t)) \in \mathcal{L}(\mathcal{A}_0)$. On construit $\mathcal{A}_n$ tel que $t \in \mathcal{L}(\mathcal{A}_n)$ si et seulement si $delay(t) \in \mathcal{L}(\mathcal{A}_{n-1})(\mathcal{A}_n = wait(\mathcal{A}_{n-1}))$ et on a un automate d'arbres acceptant des arbres de stratégie $f_n$ pour $t_n$ tels qu'il existe une stratégie $f_{n-1}$ pour $t_{n-1}$, ... une stratégie $f_1$ pour $t_1$ telles que $f_1 \oplus_1 (\ldots (f_{n-1} \oplus_1 t)) \in \mathcal{L}(\mathcal{A}_0)$.

$\mathcal{L}(\mathcal{A}_n)$ non vide si et seulement si $\psi$ est réalisable.

### 2.3.4 Remarque sur la complexité

On remarque que dans l'une comme l'autre des constructions, le théorème de base faisant marcher la preuve a besoin d'un automate non déterministe pour construire un automate alternant. Donc, pour passer à l'induction, on doit transformer l'automate alternant nouvellement construit en un automate non déterministe. Cette opération ayant un coût exponentiel et devant être répétée $O(n)$ fois pour un pipe-line de taille $n$, on obtient une complexité non élémentaire a priori pour la décidabilité du pipe-line.

## 2.4 Critère de décidabilité pour spécifications sur toutes les variables

B. Finkbeiner et S. Schewe [6] ont étendu ce résultat de décidabilité du pipe-line et ont exhibé un critère uniforme d'indécidabilité : l'*information fork*. Ce critère est vrai pour des spécifications du $\mu$-calcul qui contraignent toutes les variables du système (internes et externes). Par ailleurs, le modèle utilisé autorise le « multicast » de l'information – c'est-à-dire qu'une même variable peut étiqueter plusieurs arêtes. Ici encore, les preuves reposent sur des techniques d'automates d'arbres. Nous ne les développerons pas, mais présenterons plutôt les idées intuitives sur lesquelles repose ce critère. Ici, nous allons utiliser comme définition d'information fork la caractérisation présentée dans [6], utilisant le formalisme des graphes de processus.

### 2.4.1 Définition

Soit $(P, V, r, w)$ une architecture distribuée classique. On définit un ordre $\preceq$ sur les processus tel que $p \preceq q$ si et seulement si « $p$ a au moins autant d'information que $q$ ». Formellement, si on pose

$$E_p = \{z \in V / z \notin r^{-1}(p)\}$$

l'ensemble des variables invisibles pour $p$, et

$$U_p = \{q \in P / \text{ il n'y a pas de chemin de l'environnement à } p \text{ dans } E_p\}$$

on dit que $p \preceq q$ si et seulement si $q \in U_p$.

**Définition 2.4.1 (*Information fork*)**

*On dit qu'une architecture a un* information fork *si et seulement si on ne peut pas totalement pré-ordonner tous ses processus.*

La figure 2.10 donne des graphes dans lesquels le préordre est total. La figure 2.11 met en évidence la fourche d'information dans des architectures avec information fork.

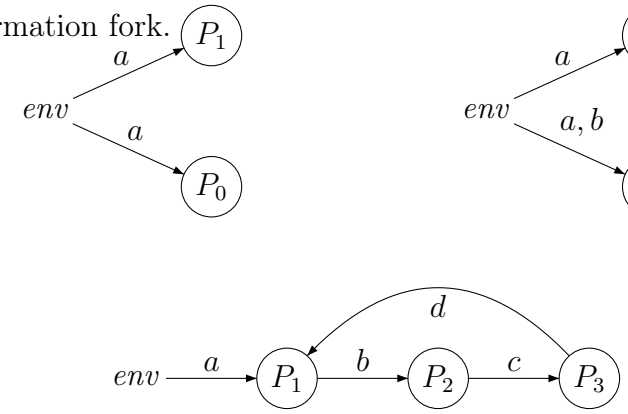



FIG. 2.10 – Exemples de graphes de processus sans information fork

### 2.4.2 Procédure de décision

Le résultat principal de [6] est le suivant.

**Théorème 2.4.2**

*Le problème de synthèse sur une architecture $\mathfrak{A}$ est décidable pour des spécifications $\mu$-calcul portant sur toutes les variables si et seulement si*

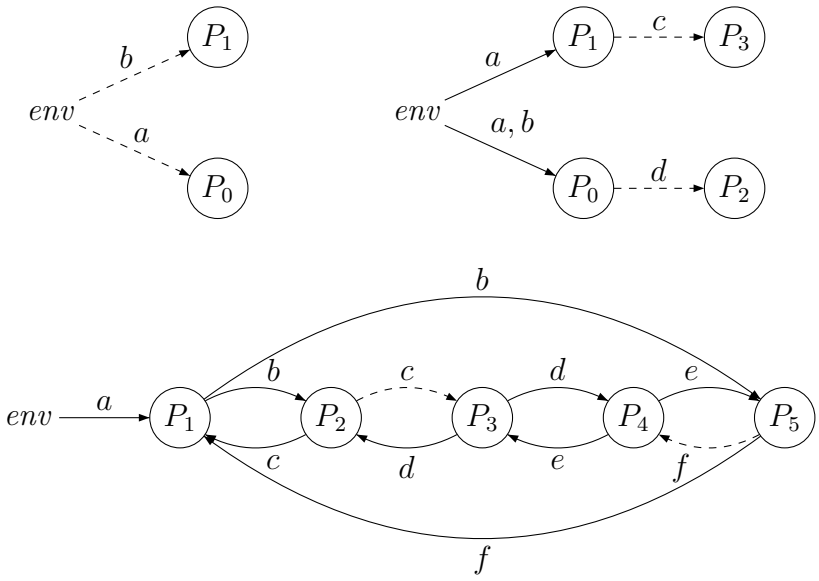



FIG. 2.11 – Exemples de graphes de processus avec information fork

$\mathfrak{A}$ *est sans information fork.*

Lorsqu'une architecture ne contient pas d'information fork, l'algorithme de synthèse commence par effectuer quelques transformations dessus : essentiellement il s'agit de regrouper les processus ayant la même quantité d'information, et de supprimer les arcs allant d'un processus moins informé vers un processus plus informé.

Les processus ayant accès à la même quantité d'information peuvent simuler les stratégies de leurs variables réciproques : on peut donc les fusionner en un seul processus et obtenir ainsi une architecture totalement ordonnée équivalente à l'architecture initiale. Dans le même ordre idée, les arcs allant d'un processus moins bien informé à un processus mieux informés sont redondants car ces arcs « retour » peuvent être simulés par le processus le mieux informé. On peut donc construire une architecture équivalente à l'architecture totalement ordonnée déjà obtenue en supprimant tous les arcs « retour ».

Une fois ces transformations effectuées, on obtient une architecture de la forme présentée à la figure 2.12.

C'est en fait un pipe-line dans lequel chaque processus peut recevoir des entrées de l'environement. Cette architecture reste décidable, et avec des techniques d'automates d'arbres, car les processus sont totalement ordonnés : le premier processus a accès à toute l'information. La synthèse de programme est donc une généralisation de la construction par automate d'arbres présen-

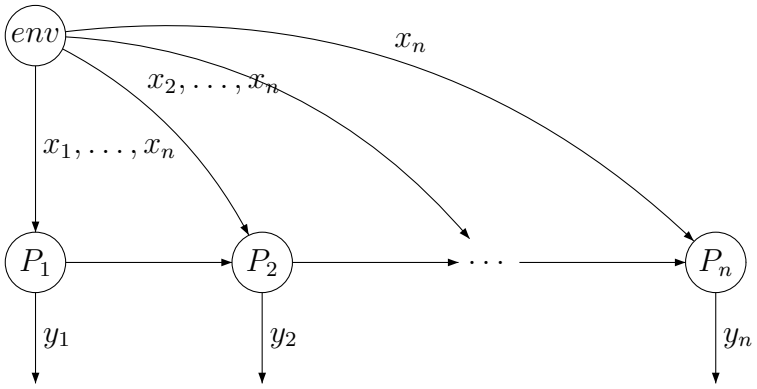



FIG. 2.12 – Forme générique des architectures décidables

tée dans [11] (et que nous avons rappelée à la section 2.3) : il s'agit, à partir de l'automate d'arbres reconnaissant les stratégies globales du système, de construire successivement des automates d'arbres reconnaissant des stratégies globales pour un nombre de plus en plus petits de processus, tels qu'il existe des stratégies pour les processus « retirés » de l'architecture globale, telles que le comportement global induit satisfait la spécification. Comme à la section précédente, la preuve repose sur un théorème assurant que cette opération de quotient sur les automates d'arbres est possible.

### 2.4.3 Indécidabilité des architectures avec information fork

La preuve d'indécidabilité repose sur l'idée que dès que l'environnement envoie de l'information séparément à deux processus distincts, cela suffit pour retrouver la preuve d'indécidabilité donnée dans [20]. En effet, rajouter de la communication entre les processus dans ce cas-là n'apporte rien, car la spécification - portant également sur les canaux internes - peut imposer d'encoder l'information transmise, rendant ainsi la valeur en entrée inutilisable pour le deuxième processus.

# Chapitre 3

# Nouveaux résultats : Un nouveau critère de décidabilité

Dans ce chapitre nous présentons un critère de décidabilité pour des spécifications du $\mu$-calcul ne contraignant que les variables d'entrée et de sortie du système (et non les canaux internes), sur les architectures à *bande passante maximale*, dans une sémantique 0-délai. Nous exposons d'abord ce qui nous a amené à définir un tel critère, puis nous présentons la preuve de décidabilité d'une architecture servant de base à la décidabilité du critère, et enfin le critère proprement dit : la notion d'*information incomparable*.

## 3.1 Motivations

### Type de spécifications

Le critère proposé dans [6] - qui autorisait les spécifications à contraindre *toutes les variables* du système - ne nous semblait pas satisfaisant : contraindre les variables internes force parfois l'indécidabilité ; que se passe-t-il si l'on veut être plus fin ? D'un point de vue intuitif, il semble raisonnable de supposer que les processus à l'intérieur d'une architecture sont libres de communiquer de la façon dont ils le souhaitent pour obtenir le résultat attendu, qui, lui, repose sur les variables d'entrée et de sortie uniquement. C'est d'ailleurs le cadre proposé par Pnueli et Rosner dans [20]. Par la suite O. Kupferman et M.Y. Vardi (dans [11]) ont démontré des résultats de décidabilité sur certaines architectures (en partie exposée au chapitre précédent) pour des spécifications CTL* contraignant toutes les variables du système. Cette extension du pouvoir d'expression des formules nous semble justifiée dans ce cas précis car il s'agissait de *décidabilité* ; le résultat est donc plus fort. Par

contre, ce type de spécification réduit le nombre d'architectures décidables : nous montrons à la section 3.2 une architecture indécidable pour une spécification contraignant toutes les variables, mais qui devient décidable si on lève cette dernière contrainte. Ce type de spécifications nous semble donc trop fort pour un critère (nécessaire et suffisant) de décidabilité.

### Bande passante

Par ailleurs, lorsque les canaux internes ne sont pas contraints par la spécification, un facteur qui influence beaucoup la décidabilité de l'architecture est celui de la bande passante de ces canaux - la quantité d'information qu'ils peuvent transmettre. C'est pourquoi, dans un premier temps, nous nous sommes proposés de démontrer un nouveau critère de décidabilité, sur des architectures à bande passante maximale, c'est-à-dire des architectures pour lesquelles les bandes passantes des canaux internes sont suffisamment grandes pour faire passer toute l'information possible. Cette contrainte sur les bandes passantes est d'ailleurs souvent une condition nécessaire de décidabilité : des architectures décidables si l'on est à bande passante maximale deviennent indécidables si on restreint d'une certaine manière la capacité des canaux.

Par ailleurs, nous considérons pour ce critère une sémantique 0-délai.

## 3.2 Preuve de décidabilité de l'architecture $\mathfrak{A}_1$

Nous allons d'abord prouver la décidabilité de l'architecture de la figure 3.1 (ou 3.2)ci-dessous dans laquelle $|D(x_0)| \leq |D(y)|$. Cette architecture va se révéler être l'architecture sous-jacente de toutes les architectures décidables dans ce contexte, c'est donc sa décidabilité qui induit la décidabilité de toutes les architectures n'ayant pas d'information incomparable. Cette architecture contient un *information fork* au sens de [6] (formé par les variables $x_0$ et $x_1$ et par les processus $P_0$ et $P_1$) et est donc indécidable pour des spécifications du $\mu$-calcul contraignant toutes les variables du système.

Nous allons montrer que dans le cadre que nous nous sommes fixés :

– spécifications ne portant pas sur $Y$,

– bande passante suffisante,

cette architecture devient décidable. Nous exposons tout d'abord une construction générale sur les automates d'arbres, puis nous montrons comment elle s'applique à la décidabilité de l'architecture de la figure 3.2, pour des spécifications du $\mu$-calcul, dans une sémantique 1-délai, puis dans une sémantique 0-délai.

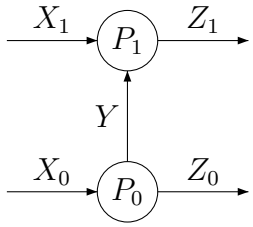



FIG. 3.1 – Architecture $\mathfrak{A}_1$ : graphe de processus

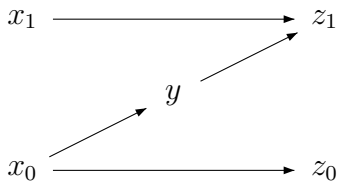



FIG. 3.2 – Architecture $\mathfrak{A}_1$ : graphe de variables

### 3.2.1 Constructions sur les automates d'arbres

#### Rappels sur les arbres

On rappelle les théorèmes suivants :

**Théorème 3.2.1 (*[10]*)**

*Étant donnée une formule $\psi$ du $\mu$-calcul sur un ensemble AP de propositions atomiques et un ensemble $X$ de directions, il existe un automate alternant à parité $\mathcal{A}_{\psi,X}$ reconnaissant exactement les $(X, 2^{AP})$-arbres satisfaisant $\psi$.*

On rappelle que, étant donné un $(X, Y)$-arbre $t$ on note $xray(t)$ le $(X, X \times Y)$-arbre $t$ auquel on a ajouté la direction du noeud dans chaque étiquette. Formellement, pour $\sigma \in X^*$, $x \in X$, $xray(t)(\sigma{\cdot}x) = (x, t(\sigma{\cdot}x))$ et $xray(t)(\epsilon) = \perp$

On définit également, pour tout $(X, Y)$-arbre $t$ un $(X \times Z, Y)$-arbre $wide_Z(t)$, dont les étiquettes ne tiennent pas compte de $Z$. Formellement, pour tout $(\sigma_X, \sigma_Z) \in (X \times Z)^*$, $wide_Z(t)(\sigma_X, \sigma_Z) = t(\sigma_X)$.

Enfin, pour tout $(X, Y)$-arbre $t$, on peut construire un $(X, Y)$-arbre $delay(t)$ tel que pour tout $\sigma \in X^*$, $x \in X$, $delay(t)(\sigma \cdot x) = t(x_0 \cdot \sigma)$, avec $x_0 \in X$ une valeur arbitraire de direction de la racine.

**Théorème 3.2.2 (*[12]*)**

*Soit un automate d'arbres alternant $\mathcal{A}$ sur des $(X, X \times Y)$-arbres. On peut construire un automate d'arbres alternant $\mathcal{A}'$ sur des $(X, Y)$-arbres tel que $\mathcal{A}'$ accepte un arbre $t$ si et seulement si $\mathcal{A}$ accepte $xray(t)$.*

On note $\mathcal{A}' = cover(\mathcal{A})$.

**Théorème 3.2.3 (*[12]*)**

*Soient $X$,$Y$ et $Z$ des ensembles finis. Soit un automate d'arbres alternant $\mathcal{A}$ sur des $(X \times Z, Y)$-arbres. On peut construire un automate d'arbres alternant $\mathcal{A}'$ sur des $(X, Y)$-abres tel que $\mathcal{A}'$ accepte $t$ si et seulement si $\mathcal{A}$ accepte $wide_Z(t)$.*

On note $\mathcal{A}' = narrow_Z(\mathcal{A})$.

**Théorème 3.2.4 (*[12]*)**

*Soit $\mathcal{A}$ un automate d'arbres alternant sur des $(X, Y)$-arbres. On peut construire un automate d'arbres alternant $\mathcal{A}'$ sur des $(X, Y)$-arbres tel que $\mathcal{A}'$ accepte un arbre $t$ si et seulement si $\mathcal{A}$ accepte $delay(t)$.*

On note $\mathcal{A}' = wait(\mathcal{A})$.

**La construction**

Pour un $(X_0 \times X_1, Z_0)$-arbre $f_{Z_0}$ et un $(X_0 \times X_1, Z_1)$-arbre $f_{Z_1}$, on définit $f$ la composée des deux comme étant un $(X_0 \times X_1, Z_0 \times Z_1)$-arbre tel que $\forall(\sigma_1, \sigma_2) \in (X_0 \times X_1)^*$ :

$$f(\sigma_1, \sigma_2) \quad = \quad (f_{Z_0}(\sigma_1, \sigma_2), f_{Z_1}(\sigma_1, \sigma_2))$$

et on note $f = (f_{Z_0}, f_{Z_1})$.

**Définition 3.2.5**

*Soit $f$ un $(X_0 \times X_1, Z)$-arbre. On dit que $f$ est* avec délai sur $X_0$ *si pour tout $(\sigma_0, \sigma_1) \in (X_0 \times X_1)^*$ et pour tout $x_0$, $x'_0 \in X_0$, $x_1 \in X_1$, $f(\sigma_0 \cdot x_0, \sigma_1 \cdot x_1) = f(\sigma_0 \cdot x'_0, \sigma_1 \cdot x_1)$*

**Théorème 3.2.6**

*Soient $X_0, X_1, Z_0, Z_1$ des ensembles finis. Si on a un automate d'arbres non déterministe $\mathcal{A}$ sur des $(X_0 \times X_1, Z_0 \times Z_1)$-arbres, on peut construire $\mathcal{A}'$ automate d'arbres non déterministe sur des $(X_0 \times X_1, Z_0)$-arbres tel que $f_{Z_0} \in \mathcal{L}(\mathcal{A}')$ si et seulement si il existe $f_{Z_1}$ $(X_0 \times X_1, Z_1)$-arbre avec délai sur $X_0$ tel que $(f_{Z_0}, f_{Z_1}) \in \mathcal{L}(\mathcal{A})$.*

On note $\mathcal{A}' = div_{Z_1}(\mathcal{A})$.

Intuitivement, l'automate d'arbres $\mathcal{A}'$ mime le comportement qu'aurait l'automate $\mathcal{A}$ sur la composée du $(X_0 \times X_1, Z_0)$-arbre $f_{Z_0}$ qu'on lit et d'un $(X_0 \times X_1, Z_1)$-arbre $f_{Z_1}$ que l'on devine. L'arbre $f_{Z_1}$ devant être avec délai sur $X_0$, l'automate $\mathcal{A}'$ devine les valeurs $Z(x_1) = f_{Z_1}(\sigma_0 \cdot x_0, \sigma_1 \cdot x_1)$ lorsqu'il est dans l'état associé au noeud $(\sigma_0, \sigma_1)$ et mémorise sa valeur dans l'état associé à chaque direction $(\sigma_0 \cdot x_0, \sigma_1 \cdot x_1)$. Ainsi, on s'assure que la valeur devinée est indépendante de la direction sur $X_0$ choisie. Dans chaque état $(q, z_1)$, en lisant l'étiquette $z_0$, l'automate $\mathcal{A}'$ se comporte comme $\mathcal{A}$ lorsque ce dernier se trouve dans l'état $q$ et lit l'étiquette $z_0, z_1$. Le langage ainsi reconnu correspond bien à celui décrit par le théorème.

**Démonstration :** Soit $\mathcal{A} = (Q, Z_0 \times Z_1, q_0, \delta, \alpha)$. On définit l'automate $\mathcal{A}'$ sur les $(X_0 \times X_1, Z_0)$-arbres ainsi :

$$\mathcal{A}' = (Q \times Z_1, Z_0, \{q_0\} \times Z_1, \delta', \alpha \times Z_1\}$$

avec pour tout $(q, z_1) \in Q \times Z_1$, et pour tout $z_0 \in Z_0$,

$$\delta'((q, z_1), z_0) = \bigvee_{\substack{A \vDash \delta(q, (z_0, z_1)), \\ Z \in Z_1^{X_1}}} \bigwedge_{\substack{x_0 \in X_0, \\ x_1 \in X_1}} (A(x_0, x_1), Z(x_1)), (x_0, x_1).$$

– Soit $f_{Z_0} \in \mathcal{L}(\mathcal{A}')$. $f_{Z_0} : (X_0 \times X_1)^* \to Z_0$. Soit $\rho' : (X_0 \times X_1)^* \to Q \times Z_1$ un run acceptant de $\mathcal{A}'$ sur $f_{Z_0}$. Ce qui signifie que pour tout $\sigma \in (X_0 \times X_1)^*$, $\bigwedge_{x \in X_0 \times X_1} \rho'(\sigma \cdot x) \vDash \delta'(\rho'(\sigma), f_{Z_0}(\sigma))$.
On pose

$$\begin{aligned} \rho : (X_0 \times X_1)^* &\to Q \\ f_{Z_1} : (X_0 \times X_1)^* &\to Z_1 \end{aligned}$$

tels que $\rho' = (\rho, f_{Z_1})$.
On veut montrer que $\rho$ est un run acceptant de $\mathcal{A}$ sur $(f_{Z_0}, f_{Z_1})$. Soit $(\sigma_0, \sigma_1) \in (X_0 \times X_1)^*$. Pour simplifier les notations, on pose $z_0 = f_{Z_0}(\sigma_0.\sigma_1)$, $z_1 = f_{Z_1}(\sigma_0, \sigma_1)$ et $q = \rho(\sigma_0, \sigma_1)$. $\rho'$ étant un run de $\mathcal{A}'$ il existe $A \in Q^{X_0 \times X_1}$ et il existe $Z \in Z_1^{X_1}$ tels que pour tout $(x_0, x_1) \in X_0 \times X_1$, $\rho(\sigma_0 \cdot x_0, \sigma_1 \cdot x_1) = A(x_0, x_1)$ et $f_{Z_1}(\sigma_0 \cdot x_0, \sigma_1 \cdot x_1) = Z(x_1)$ par définition de $\delta'$. La fonction $Z$ ne dépend que de $x_1$ donc on a bien pour tout $x_0, x_0' \in X_0$, pour tout $x_1 \in X_1$, pour tout $(\sigma_0, \sigma_1) \in (X_0, X_1)^*$, $f_{Z_1}(\sigma_0 \cdot x_0, \sigma_1 \cdot x_1) = f_{Z_1}(\sigma_0 \cdot x_0', \sigma_1 \cdot x_1)$, ce qui montre que $f_{Z_1}$ est avec délai sur $X_0$. Par ailleurs, $\bigwedge_{(x_0,x_1) \in X_0 \times X_1} A(x_0, x_1) \vDash \delta(q, z_0, z_1)$ par définition de $\delta'$, donc $\bigwedge_{(x_0,x_1) \in X_0 \times X_1} A(x_0, x_1) = \bigwedge_{(x_0,x_1) \in X_0 \times X_1} \rho(\sigma_0 \cdot x_0, \sigma_1 \cdot x_1) \vDash \delta(\rho(\sigma_0, \sigma_1), (f_{Z_0}, f_{Z_1})(\sigma_0, \sigma_1))$. Donc $\rho$ est bien un run (acceptant) de

$\mathcal{A}$ sur $(f_{Z_0}, f_{Z_1})$ et il existe bien un arbre $f_{Z_1} : (X_0 \times X_1)^* \to Z_1$ avec délai sur $X_0$ tel que $(f_{Z_0}, f_{Z_1}) \in \mathcal{L}(\mathcal{A})$.

– L'autre sens de l'implication est similaire. Soit $f_{Z_0} : (X_0 \times X_1)^* \to Z_0$ et $f_{Z_1} : (X_0 \times X_1)^* \to Z_1$ arbre avec délai sur $X_0$ tels que $(f_{Z_0}, f_{Z_1}) \in \mathcal{L}(\mathcal{A})$. On a donc $\rho : (X_0 \times X_1)^* \to Q$ un run acceptant de $\mathcal{A}$ sur $f_{Z_0} + f_{Z_1}$. Posons $\rho' : (X_0 \times X_1)^* \to Q \times Z_1$ tel que pour tout $(\sigma_0, \sigma_1) \in (X_0 \times X_1)^*$, $\rho'(\sigma_0, \sigma_1) = (\rho(\sigma_0, \sigma_1), f_{Z_1}(\sigma_0, \sigma_1))$. Alors $\rho'$ est un run acceptant de $\mathcal{A}'$ sur $f_{Z_0}$. En effet, soit $(\sigma_0, \sigma_1) \in (X_0 \times X_1)^*$. Pour tout $(x_0, x_1) \in X_0 \times X_1$, $x'_0 \in X_0$, $\rho'(\sigma_0 \cdot x_0, \sigma_1 \cdot x_1) = (\rho(\sigma_0 \cdot x_0, \sigma_1 \cdot x_1), f_{Z_1}(\sigma_0 \cdot x_0, \sigma_1 \cdot x_1)) = (\rho(\sigma_0 \cdot x_0, \sigma_1 \cdot x_1), f_{Z_1}(\sigma_0 \cdot x'_0, \sigma_1 \cdot x_1)$ avec $\bigwedge_{(x_0,x_1) \in X_0 \times X_1} \rho(\sigma_0 \cdot x_0, \sigma_1 \cdot x_1) \vDash \delta(\rho(\sigma_0, \sigma_1)$, par définition de $\rho$ run de $\mathcal{A}$ et de $f_{Z_1}$, arbre avec délai sur $X_0$. On peut donc définir $Z \in Z_1^{X_1}$ par $Z(x_1) = f_{Z_1}(\sigma_0 \cdot x'_0, \sigma_1 \cdot x_1)$ et $A \in Q^{X_0 \times X_1}$ tel que pour tout $(x_0, x_1) \in X_0 \times X_1$, $A(x_0, x_1) = \rho(\sigma_0 \cdot x_0, \sigma_1 \cdot x_1)$. On a donc, par définition de $\delta'$, $\bigwedge_{(x_0,x_1) \in (X_0 \times X_1)} (\rho'(\sigma_0 \cdot x_0, \sigma_1 \cdot x_1) \vDash \delta'(\rho'(\sigma_0, \sigma_1), f_{Z_0}(\sigma_0, \sigma_1))$ et $\rho'$ est un run acceptant de $\mathcal{A}'$ sur $f_{Z_0}$.

On a ainsi montré que le langage accepté par l'automate $\mathcal{A}'$ était égal à l'ensemble des arbres qui, composés avec un $(X_0 \times X_1, Z_1)$-arbre avec délai sur $X_0$ sont acceptés par l'automate de départ $\mathcal{A}$. ■

### 3.2.2 Décidabilité de l'architecture $\mathfrak{A}_1$

On montre à présent quelle est la procédure de synthèse de programmes pour l'architecture $\mathfrak{A}_1$, étant donnée une formule quelconque du $\mu$-calcul.

#### Notations

On rappelle qu'on définit une fonction d'interprétation des variables $s$ qui associe à chaque variable une valeur dans son domaine de définition. $s_0$ donne les valeurs des variables à l'instant initial, puis pour tout $i \in \mathbb{N}$, $s_i[X]$ donne les valeurs jouées par l'environnement à l'instant $i$, $s_i[Y \cup Z]$ donne les valeurs jouées sur les fils internes et de sortie. On note $\overline{s}_i$ pour $s_0 \cdot s_1 \cdots s_i$, l'historique des valeurs jouées jusqu'à l'instant $i$. Pour $\sigma \in X^*$ et une stratégie $f$ ayant pour domaine de définition $X^*$, on note $\overline{f}(\sigma) = f(\epsilon) \cdots f(\sigma)$. On rappelle qu'on identifie les stratégies à des arbres dont les directions représentent les valeurs jouées par l'environnement et les étiquettes les valeurs à écrire sur les fils internes et de sortie. Un *arbre de run* suivant une stratégie $f$ est un arbre dont les directions sont également les valeurs jouées par l'environnement, et les étiquettes donnent les valeurs qu'on peut lire, *au même instant* sur les différents fils d'entrée et de sortie. Les différents runs possibles se lisent donc sur les différentes branches de l'arbre de run. Dans le cas d'une sémantique

1-délai, on remarque que l'arbre des runs associé à $f$ est $xray(delay(f))$. Par la suite, lorsqu'on considérera les stratégies comme des arbres, on désignera les noeuds par des variables du type $\sigma$, lorsqu'on considérera les stratégies par rapport à un run, on utilisera plutôt les notations $\overline{s}_i$ pour désigner l'historique des valeurs jouées.

**Exemple**

Pour l'architecture $\mathfrak{A}_1$, si on note $f_{z_0}, f_{z_1}, f_y$ respectivement les stratégies des variables $z_0$, $z_1$ et $y$ et $f$ la stratégie globale du système, $s_0 \cdot s_1 \cdots$ est un $f$-run si pour tout $i \in \mathbb{N}$ :
$\overline{s}_{i+1}[z_0, z_1] = f(\overline{s}_i[x_0, x_1]) = f_{z_0}(\overline{s}_i(x_0)), f_{z_1}(\overline{f}_y(\overline{s}_{i-1}(x_0)), \overline{s}_i(x_1))$. On définit un arbre de $f$-run $s : (D(x_0) \times D(x_1))^* \to (D(x_0) \times D(x_1) \times D(z_0) \times D(z_1))$ tel que pour tout $(\sigma_0, \sigma_1) \in (D(x_0) \times D(x_1))^*$, $(x_0, x_1) \in D(x_0) \times D(x_1)$, $s(\sigma_0 \cdot x_0, \sigma_1 \cdot x_1) = x_0, x_1, f(\sigma_0, \sigma_1)$, et $s(\epsilon, \epsilon) = \bot$. Pour le run

$$\begin{array}{ccc} s_0(x_0) & s_1(x_0) & \dots \\ s_0(x_1) & s_1(x_1) & \dots \\ s_0(y) & s_1(y) & \dots \\ s_0(z_0) & s_1(z_0) & \dots \\ s_0(z_1) & s_1(z_1) & \dots \end{array}$$

selon la stratégie $f$, l'arbre de stratégie est étiqueté sur les noeuds de ce run par $f(\epsilon, \epsilon) = s_0(z_0), s_0(z_1)$ les valeurs initiales pour les variables $z_0$ et $z_1$. Le noeud $(s_0(x_0), s_0(x_1))$ est étiqueté par $f(s_0(x_0), s_0(x_1)) = (s_1(z_0), s_1(z_1))$ et ainsi de suite. La branche correspondante dans l'arbre de run $s$ est de la forme :

$$\begin{aligned} s(\epsilon, \epsilon) &= \bot \\ s(s_0(x_0), s_0(x_1)) &= (s_0(x_0), s_0(x_1), s_0(z_0), s_0(z_1)) \\ &= (s_0(x_0), s_0(x_1), f(\epsilon, \epsilon)) \end{aligned}$$

(car nous sommes dans une sémantique 1-délai)

$$\begin{aligned} s((s_0(x_0), s_0(x_1)) \cdot (s_1(x_0), s_1(x_1))) &= (s_1(x_0), s_1(x_1), s_1(z_0), s_1(z_1)) \\ &= (s_1(x_0), s_1(x_1), f(s_0(x_0), s_0(x_1))) \end{aligned}$$

etc.

**Procédure de décision**

Soit $\psi$ une formule de $CTL^*$ sur $D(x_0) \times D(x_1) \times D(z_0) \times D(z_1)$. Pour résoudre le problème de synthèse sur cette architecture, on construit les automates suivants :

- $\mathcal{A}_\psi$ automate de Rabin alternant sur des $(D(x_0) \times D(x_1), D(x_0) \times D(x_1) \times D(z_0) \times D(z_1))$-arbres. Cet automate accepte un arbre $t$ si et seulement si il satisfait $\psi$.
- un automate de Rabin alternant $\mathcal{A} = cover(\mathcal{A}_\psi)$,
- un automate de Rabin alternant $\mathcal{A}' = wait(\mathcal{A})$ sur des $(D(x_0) \times D(x_1), D(z_0) \times D(z_1))$-arbres. L'automate $\mathcal{A}'$ reconnaît en fait les arbres de stratégies globales gagnantes.
- $\mathcal{A}''$ automate de Rabin non déterministe équivalent à $\mathcal{A}'$.
- $\mathcal{A}_1$ automate de Rabin non déterministe tel que $\mathcal{A}_1 = div_{D(z_1)}(\mathcal{A}'')$. $\mathcal{A}_1$ accepte des $(D(x_0) \times D(x_1), D(z_0))$-arbres, c'est-à-dire des arbres de stratégies pour $z_0$ ayant connaissance de la valeur de $x_1$ tels qu'il existe des arbres de stratégie pour $z_1$ tels que la stratégie composée des deux soit acceptée par $\mathcal{A}''$.
- $\mathcal{A}'_1 = narrow_{D(x_1)}(\mathcal{A}_1)$ automate sur des $(D(x_0), D(z_0))$-arbres qui accepte les arbres acceptés par $\mathcal{A}_1$ qui ne dépendent pas de $x_1$.

On a donc le résultat suivant :

**Théorème 3.2.7**

*La synthèse dans le cas 1-délai pour l'architecture ci-dessus est réalisable si et seulement si le langage de l'automate $\mathcal{A}'_1$ est non vide.*

**Démonstration :** Supposons qu'il existe $f_{z_0} \in \mathcal{L}(\mathcal{A}'_1)$. Alors $f_0 = wide_{D(x_1)}(f_{z_0}) \in \mathcal{L}(\mathcal{A}_1)$ par définition de la construction *narrow*. Donc, d'après le théorème 3.2.6, il existe $f_1$ un $(D(x_0) \times D(x_1), D(z_1))$-arbre avec délai sur $D(x_0)$ tel que $(f_0, f_1)$ appartient à $\mathcal{L}(\mathcal{A}'')$. La fonction $f_{z_0}$ est la stratégie pour la variable $z_0$. On définit la stratégie sans mémoire $f_y$ pour la variable $y$ ainsi :

$$\begin{aligned} f_y : D(x_0)^* &\to D(y) \\ f_y(\epsilon) &= s_0(y) \\ f_y(\overline{s}_i(x_0)) &= s_i(x_0) \end{aligned}$$

On définit aussi la stratégie pour la variable $z_1$ :

$$\begin{aligned} f_{z_1} &: (D(y) \times D(x_1))^* \to D(z_1) \\ f_{z_1}(\epsilon, \epsilon) &= f_1(\epsilon, \epsilon) \\ f_{z_1}(\overline{s}_i[y, x_1]) &= \begin{cases} f_1(s_1(y) \cdots s_i(y) \cdot x, \overline{s}_i(x_1)) & \text{pour } x \in D(x_0) \text{ quelconque} \\ & \text{si } s_j(y) \in D(x_0) \text{ pour } j > 1 \\ 0 & \text{sinon} \end{cases} \end{aligned}$$

La fonction $f_{z_1}$ est bien définie car $f_1$ est avec délai sur $D(x_0)$, donc quel que soient $x, x' \in D(x_0)$, $f_1(s_1(y) \cdots s_i(y) \cdot x, \overline{s}_i(x_1)) = f_1(s_1(y) \cdots s_i(y) \cdot$

$x', \overline{s}_i(x_1))$. On définit la stratégie globale du système comme étant la fonction

$$\begin{aligned} f : (D(x_0) \times D(x_1))^* &\rightarrow D(z_0) \times D(z_1) \\ f(\epsilon, \epsilon) &= f_{z_0}(\epsilon), f_{z_1}(\epsilon, \epsilon) \\ f(s_0[x_0, x_1]) &= f_{z_0}(s_0[x_0]), f_{z_1}(f_y(\epsilon), s_0[x_1]) \\ f(\overline{s}_{i+1}[x_0, x_1]) &= f_{z_0}(\overline{s}_{i+1}[x_0]), f_{z_1}(\overline{f}_y(\overline{s}_i[x_0]), \overline{s}_{i+1}[x_1]) \end{aligned}$$

On montre que l'arbre de stratégie $f$ est égal à l'arbre $(f_0, f_1)$. En effet, pour tout $(\sigma_0, \sigma_1) \in (D(x_0) \times D(x_1))^*$, on a $f_{z_0}(\sigma_0) = f_0(\sigma_0, \sigma_1)$, par construction. Par ailleurs, pour tout $(\sigma_0, \sigma_1) \in (D(x_0) \times D(x_1))^*$, et pour tout $x_1 \in D(x_1)$, $f_{z_1}(\overline{f}_y(\sigma_0), \sigma_1 \cdot x_1) = f_{z_1}(s_0(y) \cdot \sigma_0, \sigma_1 \cdot x_1) = f_1(\sigma_0 \cdot x, \sigma_1 \cdot x_1)$ pour $x$ quelconque appartenant à $D(x_0)$. On a donc, pour tout $(\sigma_0, \sigma_1) \in (D(x_0) \times D(x_1))^*$, $f(\sigma_0, \sigma_1) = (f_0, f_1)(\sigma_0, \sigma_1)$, et l'arbre de stratégie globale $f$ est reconnu par $\mathcal{A}''$, ce qui implique que l'arbre de run induit par cette stratégie distribuée $s = xray(delay(f))$ appartient, par construction des différents automates, à $\mathcal{L}(\mathcal{A}_\psi)$ et la stratégie est gagnante. Donc, si $\mathcal{L}(\mathcal{A}'_1)$ est non vide, alors il existe une stratégie distribuée gagnante pour la spécification $\psi$.

Inversement, supposons la spécification réalisable. Alors il existe

$$\begin{aligned} f_{z_0} : D(x_0)^* &\rightarrow D(z_0) \\ f_y : D(x_0)^* &\rightarrow D(y) \\ f_{z_1} : (D(y) \times D(x_1))^* &\rightarrow D(z_1) \end{aligned}$$

stratégies telles que la stratégie composée $f$ soit gagnante. Le run induit $s = xray(delay(f)) \in \mathcal{L}(\mathcal{A}_\psi)$ (car $f$ est gagnante), donc $f \in \mathcal{L}(\mathcal{A}'')$. On pose

$$\begin{aligned} f_0 &= wide_{D(x_0)}(f_{z_0}) \\ f_1 : (D(x_0) \times D(x_1))^* &\rightarrow D(z_1) \\ (\epsilon, \epsilon) &\rightarrow f_{z_1}(\epsilon) \\ \overline{s}_i[x_0, x_1] &\rightarrow f_{z_1}(\overline{f}_y(\overline{s}_{i-1}(x_0)), \overline{s}_i(x_1)) \end{aligned}$$

On remarque que pour tout $(\sigma_0, \sigma_1) \in (D(x_0) \times D(x_1))^*$, pour tout $x_0, x'_0 \in D(x_0)$, pour tout $x_1 \in D(x_1)$,

$$\begin{aligned} f_1(\sigma_0 \cdot x_0, \sigma_1 \cdot x_1) &= f_{z_1}(\overline{f}_y(\sigma_0), \sigma_1 \cdot x_1) \\ &= f_1(\sigma_0 \cdot x'_0, \sigma_1 \cdot x_1) \end{aligned}$$

donc $f_1$ est avec délai sur $D(x_0)$. Par ailleurs, pour tout $(\sigma_0, \sigma_1) \in (D(x_0) \times$

$D(x_1))^*$, $x_0 \in D(x_0)$, $x_1 \in D(x_1)$,

$$\begin{aligned}(f_0, f_1)(\sigma_0 \cdot x_0, \sigma_1 \cdot x_1) &= f_0(\sigma_0 \cdot x_0, \sigma_1 \cdot x_1), f_1(\sigma_0 \cdot x_0, \sigma_1 \cdot x_1) \\ &= wide_{D(x_1)}(f_{z_0})(\sigma_0 \cdot x_0, \sigma_1 \cdot x_1), f_{z_1}(\overline{f}_y(\sigma_0), \sigma_1 \cdot x_1)) \\ &= f_{z_0}(\sigma_0 \cdot x_0), f_{z_1}(\overline{f}_y(\sigma_0), \sigma_1 \cdot x_1) \\ &= f\end{aligned}$$

Donc $(f_0, f_1) = f$ et $(f_0, f_1) \in \mathcal{L}(\mathcal{A}'')$ et, d'après le théorème 3.2.6, $f_0 \in \mathcal{L}(\mathcal{A}_1)$ et $f_{z_0} \in \mathcal{L}(\mathcal{A}'_1)$. Donc, s'il existe une stratégie distribuée gagnante pour la spécification $\psi$ sur l'architecture $\mathfrak{A}_1$, alors le langage de l'automate $\mathcal{A}'_1$ est non vide. ■

### 3.2.3 Décidabilité de l'architecture $\mathfrak{A}_1$ dans une sémantique 0-délai

La preuve de décidabilité est assez similaire à celle présentée dans le cadre 1-délai.

#### Construction sur les automates d'arbres

On reprend la notation pour la composition d'arbres de la section 3.2.1 : la composée d'un $(X, Z_0)$-arbre et d'un $(X, Z_1)$-arbre est un $(X, Z_0 \times Z_1)$-arbre, se note $(f_{Z_0}, f_{Z_1})$ et est définie, pour tout $\sigma \in X^*$ par

$$(f_{Z_0}, f_{Z_1})(\sigma) = f_{Z_0}(\sigma), f_{Z_1}(\sigma).$$

Le théorème sur lequel se base la preuve de décidabilité dans le cas 0-délai est le suivant :

**Théorème 3.2.8**

*Soient $X$, $Z_0$, $Z_1$ des ensembles finis. Si on a un automate d'arbres $\mathcal{A}$ non déterministe reconnaissant des $(X, (Z_0 \times Z_1)$-arbres, on peut construire $\mathcal{A}'$ automate d'arbres non déterministe sur des $(X, Z_0)$-arbres tels que $f_{Z_0} \in \mathcal{L}(\mathcal{A}')$ si et seulement si il existe $f_{Z_1}$ un $(X_0 \times X_1, Z_1)$-arbre tel que $(f_{Z_0}, f_{Z_1}) \in \mathcal{L}(\mathcal{A})$.*

La différence avec le théorème 3.2.6 est qu'ici l'arbre $f_{Z_1}$ n'est pas avec délai sur $X_0$.

On note $\mathcal{A}' = div^0_{Z_1}(\mathcal{A})$.

Intuitivement, comme pour le théorème 3.2.6, l'automate $\mathcal{A}'$ va imiter sur un arbre $f_{Z_0}$ le comportement qu'aurait l'automate $\mathcal{A}$ sur l'arbre composé par $f_{Z_0}$ et un arbre $f_{Z_1}$ qu'on devine. La contrainte de délai sur $X_0$ étant ici

levée, il suffit à l'automate $\mathcal{A}'$ de deviner, dans chaque noeud $(\sigma)$ de l'arbre $f_{Z_0}$ la valeur $z_1 = f_{Z_1}(\sigma)$ de l'arbre qu'on compose avec $f_{Z_0}$. Formellement on écrit :

**Démonstration :** Soit $\mathcal{A} = (Q, Z_0 \times Z_1, q_0, \delta, \alpha)$. On définit l'automate $\mathcal{A}'$ sur les $(X, Z_0)$-arbres ainsi :

$$\mathcal{A}' = (Q, Z_0, q_0, \delta', \alpha)$$

avec, pour tout $q \in Q$, pour tout $z_0 \in Z_0$,

$$\delta'(q, z_0) = \bigvee_{\substack{z_1 \in Z_1 \\ A \vDash \delta(q,(z_0,z_1))}} \bigwedge_{x \in X} (A(x), x)$$

Soit $f_{Z_0} : X^* \to Z_0$ arbre accepté par $\mathcal{A}'$. Soit $\rho : X^* \to Q$ un run acceptant de $\mathcal{A}'$ sur $f_{Z_0}$, ce qui signifie que, pour tout $\sigma \in X^*$, $\bigwedge_{x \in X} \rho(\sigma \cdot x) \vDash \delta'(\rho(\sigma), f_{Z_0}(\sigma))$. Soit $(\sigma) \in X^*$. Alors, il existe $z_1 \in Z_1$, $A \in Q^X$ tels que $\bigwedge_{x \in X}(A(x) \vDash \delta(\rho(\sigma), (f_{Z_0}(\sigma), z_1))$, par définition de $\delta'$. On pose alors $f_{Z_1}(\sigma) = z_1$, et on a $\bigwedge_{x \in X}(A(x) \vDash (\delta(\rho(\sigma)), (f_{Z_0}(\sigma), f_{Z_1}(\sigma)))$. Or, par définition de $\delta'$, on a, pour tout $x \in X$, $\rho(\sigma \cdot x) = A(x)$, ce qui permet d'écrire que $\bigwedge_{x \in X} \rho(\sigma \cdot x) \vDash \delta(\rho(\sigma), (f_{z_0}(\sigma), f_{Z_1}(\sigma))) = \delta(\rho(\sigma), (f_{Z_0}, f_{Z_1})(\sigma))$ et $\rho$ est également un run acceptant de $\mathcal{A}$ sur $(f_{Z_0}, f_{Z_1})$.

Inversement, soient $f_{Z_0} : X^* \to Z_0$, $f_{Z_1} : X^* \to Z_1$ deux arbres tels que $(f_{Z_0}, f_{Z_1}) \in \mathcal{L}(\mathcal{A})$. Donc il existe $\rho : X^* \to Q$ run acceptant de $\mathcal{A}$ sur $(f_{Z_0}, f_{Z_1})$. Soit $\sigma \in X^*$.On a, par définition d'un run, $\bigwedge_{x \in X} \rho(\sigma \cdot x) \vDash \delta(\rho(\sigma), (f_{Z_0}, f_{Z_1})(\sigma)) = \delta(\rho(\sigma), (f_{Z_0}(\sigma), f_{Z_1}(\sigma)))$. On pose $z_1 = f_{Z_1}(\sigma)$, $A \in Q^X$ tel que, pour tout $x \in X$, $A(x) = \rho(\sigma \cdot x)$ et on a

$$\bigwedge_{x \in X} \rho(\sigma \cdot x) \quad \vDash \quad \bigvee_{\substack{z_1 \in Z_1 \\ A \vDash \delta(\rho(\sigma),(f_{Z_0}(\sigma),z_1))}} \bigwedge_{x \in X} (A(x), x)$$
$$= \delta'(\rho(\sigma), f_{Z_0}(\sigma).$$

Donc $\rho$ est également un run (acceptant) de $\mathcal{A}'$ sur $f_{Z_0}$.

On a ainsi montré que le langage accepté par $\mathcal{A}'$ était égal à l'ensemble des $(X, Z_0)$-arbres tels qu'il existe un $(X, Z_1)$-arbre tel que la composée des deux est acceptée par $\mathcal{A}$. ■

### Décidabilité de l'architecture dans le cadre 0-délai

**Notations** Dans le cadre 0-délai, on rappelle que la stratégie pour une variable $y$ ayant comme variable en entrée la variable $x$ est définie comme

la fonction $f_y : (D(x))^+ \to D(y)$. La correspondance avec un arbre se fait donc en étiquetant la racine par $\perp$. Avec la sémantique 0-délai, les arbres de runs se calculent à partir des arbres de stratégie en rajoutant simplement la direction du noeud dans l'étiquette. Formellement, pour une stratégie $f$, un $f$-run $s$ s'écrit $s = xray(f)$. Pour l'architecture $\mathfrak{A}_1$, si on note $f_y$, $f_{z_0}$, et $f_{z_1}$ les stratégies des variables $y$, $z_0$ et $z_1$, et $f$ la stratégie globale du système, on dit donc que $s_0 \cdot s_1 \cdots$ est un $f$-run si pour tout $i \in \mathbb{N}$, $s_i[z_0, z_1] = f(\overline{s}_i[x_0, x_1]) = (f_{z_0}(\overline{s}_i[x_0]), f_{z_1}(\overline{f}_y(\overline{s}_i[x_0]), \overline{s}_i(x_1)))$. Un arbre de $f$-run $s$ ici est donc tel que

$$\begin{array}{rcl} s : (D(x_0) \times D(x_1))^* & \to & (D(x_0) \times D(x_1) \times D(z_0) \times D(z_1)) \\ s(\epsilon, \epsilon) & = & (\perp, \perp, f(\epsilon, \epsilon)) = (\perp, \perp, \perp) \\ s(\sigma_0 \cdot x_0, \sigma_1 \cdot x_1) & = & (x_0, x_1, f(\sigma_0 \cdot x_0, \sigma_1 \cdot x_1)) \end{array}$$

**Procédure de décision** La procédure de décision dans le cas 0-délai est la même que dans le cas 1-délai, sauf pour l'emploi de la construction $div^0_{D(z_1)}$ au lieu de $div_{D(z_1)}$.

Précisément, on construit les automates suivants :

- $\mathcal{A}_\psi$ automate de Rabin alternant sur des $(D(x_0) \times D(x_1), D(x_0) \times D(x_1) \times D(z_0) \cup \{\perp\} \times D(z_1) \cup \{\perp\})$-arbres. Cet automate accepte un arbre $t$ si et seulement si il satisfait $\psi$.
- un automate de Rabin alternant $\mathcal{A} = cover(\mathcal{A}_\psi)$.
- $\mathcal{A}'$ automate de Rabin non déterministe équivalent à $\mathcal{A}$.
- $\mathcal{A}_1$ automate de Rabin non déterministe tel que $\mathcal{A}_1 = div^0_{D(z_1)}(\mathcal{A}')$. $\mathcal{A}_1$ accepte des $(D(x_0) \times D(x_1), D(z_0) \cup \{\perp\})$-arbres, c'est-à-dire des arbres de stratégies pour $z_0$ ayant connaissance de la valeur de $x_1$ tels qu'il existe des arbres de stratégie pour $z_1$ tels que la stratégie composée des deux soit acceptée par $\mathcal{A}'$.
- $\mathcal{A}'_1 = narrow_{D(x_1)}(\mathcal{A}_1)$ automate sur des $(D(x_0), D(z_0) \cup \{\perp\})$-arbres qui accepte les arbres acceptés par $\mathcal{A}_1$ qui ne dépendent pas de $x_1$.

Et, comme à la section 3.2.2, on a le résultat suivant :

**Théorème 3.2.9**

*Le problème de synthèse pour l'architecture $\mathfrak{A}_1$ dans le cas 0-délai est réalisable si et seulement si le langage de l'automate $\mathcal{A}'_1$ est non vide.*

**Démonstration :** Supposons qu'il existe $f_{z_0} \in \mathcal{L}(\mathcal{A}'_1)$. Alors, l'arbre $f_0 : (D(x_0) \times D(x_1))^* \to D(z_0)$ tel que $f_0 = wide_{D(x_1)}(f_{z_0})$ est accepté par l'automate $\mathcal{A}_1$ par définition de la construction *narrow*. D'aprés le théorème 3.2.8, il existe $f_1 : (D(x_0) \times D(x_1))^* \to D(z_1)$ tel que $(f_0, f_1)$ soit accepté par $\mathcal{A}$. On définit pour les variables $z_0$, $y$ et $z_1$ les stratégies $f_{z_0}$, arbre appartenant à $\mathcal{L}(\mathcal{A}'_1)$, $f_y : D(x_0)^+ \to D(y)$ stratégie sans mémoire

telle que $f_y(\overline{s}_i[x_0]) = s_i[x_0]$, et

$$\begin{array}{rcl} f_{z_1} & : & (D(y) \times D(x_1))^+ \to D(z_1) \\ f_{z_1}(\overline{s}_i[y, x_1]) & = & \begin{cases} f_1(\overline{s}_i(y), \overline{s}_i(x_1)) & \text{si } s_j(y) \in D(x_0) \text{ pour } j \geq 0 \\ 0 & \text{sinon} \end{cases} \end{array}$$

La stratégie globale du système est définie comme suit :

$$\begin{array}{rcl} f : (D(x_0) \times D(x_1))^+ & \to & (D(z_0) \times D(z_1)) \\ f(\overline{s}_i[x_0, x_1]) & = & (f_{z_0}(\overline{s}_i(x_0)), f_{z_1}(\overline{f}_y(\overline{s}_i(x_0)), \overline{s}_i(x_1))) \\ & = & (f_{z_0}(\overline{s}_i(x_0)), f_{z_1}(\overline{s}_i[x_0, x_1])) \end{array}$$

On remarque que pour tout $i \in \mathbb{N}$, $f_{z_0}(\overline{s}_i(x_0)) = f_0(\overline{s}_i[x_0, x_1])$ par construction. Par ailleurs, $f_{z_1}(\overline{s}_i(x_0), \overline{s}_i(x_1)) = f_1(\overline{s}_i[x_0, x_1])$. Donc, $f = (f_0, f_1)$ et par suite, $f \in \mathcal{L}(\mathcal{A})$. L'arbre de $f$-run $s = xray(f)$ appartient donc, par définition de *cover*, à $\mathcal{L}(\mathcal{A}_\psi)$, et la stratégie distribuée construite est gagnante.

Inversement, supposons la spécification réalisable. Alors il existe

$$\begin{array}{rl} f_{z_0} : D(x_0)^+ & \to D(z_0) \\ f_y : (D(x_0))^+ & \to D(y) \\ f_{z_1} : (D(y) \times D(x_1))^+ & \to D(z_1) \end{array}$$

stratégies telles que la stratégie composée $f$ soit gagnante. Le run induit $s = xray(f) \in \mathcal{L}(\mathcal{A}_\psi)$, donc $f \in \mathcal{L}(\mathcal{A})$. On pose $f_0 = wide_{D(x_0)}(f_{z_0})$, et $f_1 : (D(x_0) \times D(x_1))^* \to D(z_1)$ tel que $f_1(\epsilon, \epsilon) = \perp$ et $f_1(\overline{s}_i[x_0, x_1]) = f_{z_1}(\overline{f}_y(\overline{s}_i(x_0)), \overline{s}_i(x_1))$. On remarque que, pour tout $(\sigma_0, \sigma_1) \in (D(x_0) \times D(x_1))^*$,

$$\begin{array}{rcl} (f_0, f_1)(\sigma_0, \sigma_1) & = & (f_0(\sigma_0, \sigma_1), f_1(\sigma_0, \sigma_1)) \\ & = & (f_{z_0}(\sigma_0), f_{z_1}(\overline{f}_y(\sigma_0), \sigma_1)) \\ & = & f(\sigma_0, \sigma_1) \end{array}$$

Donc $(f_0, f_1) = f$ et par suite, $(f_0, f_1) \in \mathcal{L}(\mathcal{A}) = \mathcal{L}(\mathcal{A}')$. Par le théorème 3.2.8, $f_0 \in \mathcal{L}(\mathcal{A}_1)$, et, par définition de $f_0$ et de *narrow*, $f_{z_0} \in \mathcal{L}(\mathcal{A}'_1)$. Ainsi, $\mathcal{L}(\mathcal{A}'_1$ est non vide si et seulement si la spécification est réalisable sur l'architecture $\mathfrak{A}_1$. ■

## 3.3 Critère de décidabilité pour les architectures à bande passante maximale

On définit un critère de décidabilité pour les architectures dans lesquelles la contrainte sur la bande passante est levée et pour des spécifications ne

parlant que des valeurs d'entrée et de sortie du système. Par ailleurs on s'intéresse à une sémantique 0-délai.

### 3.3.1 Critère de décidabilité

On note pour une architecture $(V, A)$ : $Acc = A^*$. On va définir un pré-ordre sur $Y$ tel que pour tout $y_0, y_1 \in Y$, $y_0 \leq y_1$ si et seulement si $y_1$ a « plus d'information » que $y_0$. Formellement, on note $y_0 \leq y_1$ si et seulement si $Acc^{-1}(y_0) \cap X \subseteq Acc^{-1}(y_1) \cap X$.

**Définition 3.3.1**

*Une architecture* $(V = X \uplus Y \uplus T, A)$ *est à* information incomparable *si il existe* $x_0$ *et* $x_1 \in X$ *tels que* $Acc(x_i) \cap Y \backslash Acc(x_{1-i}) \neq \emptyset$, $i = 0, 1$.

On rappelle que pour toute variable $z \in V$, on note $R(z) = A^{-1}(z)$.

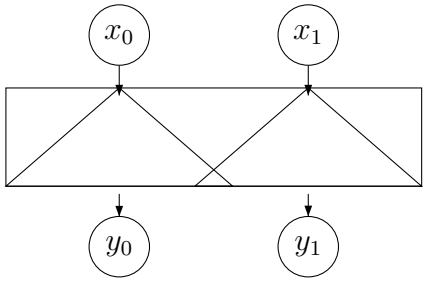



FIG. 3.3 – Schéma d'architecture à information incomparable

**Théorème 3.3.2**

*Une architecture est décidable pour des spécifications du* $\mu$*-calcul si et seulement si elle n'est pas* à information incomparable.

Nous allons le démontrer dans les sections suivantes.

On obtient facilement le lemme suivant :

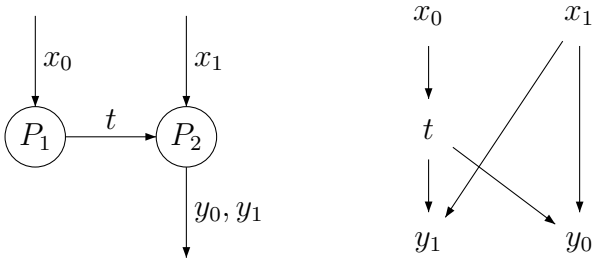
**Lemme 3.3.3**

*Une architecture est à* information incomparable *si et seulement si on ne peut pas totalement pré-ordonner les processus de* $Y$ *selon le pré-ordre* a plus d'information que.

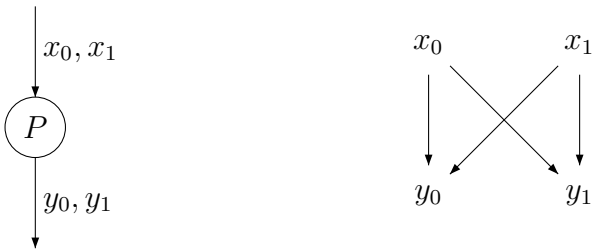
**Démonstration :** Si il existe $y_0$ et $y_1 \in Y$ tels que $y_0 \not\leq y_1$ et $y_1 \not\leq y_0$ alors il existe $x_0 \in X$ tel que $y_0 \in A^*(x_0)$ et $y_1 \notin A^*(x_0)$ (par définition de $y_0 \not\leq y_1$), et il existe $x_1 \in X$ tel que $y_1 \in A^*(x_1)$ et $y_0 \notin A^*(x_1)$ (par définition de $y_1 \not\leq y_0$). Alors $y_0 \in Acc(x_0) \cap Y \backslash Acc(x_1)$ et $y_1 \in Acc(x_1) \cap Y \backslash Acc(x_0)$ et l'architecture est à information incomparable.

Inversement, soient $y_0, y_1 \in Y$ et $x_0, x_1 \in X$ tels que $y_0 \in Acc(x_0) \cap Y \backslash Acc(x_1)$ et $y_1 \in Acc(x_1) \cap Y \backslash Acc(x_0)$. Alors $x_0 \in Acc^{-1}(y_0) \cap X$ mais $x_0 \notin Acc^{-1}(y_1) \cap X$ donc $y_0 \not\leq y_1$ et $x_1 \in Acc^{-1}(y_1) \cap X$ mais $x_1 \notin Acc^{-1}(y_0) \cap X$ donc $y_1 \not\leq y_0$. ■

Ce critère se base sur une approche différente de celle faite dans [6]. Le critère d'« information fork » défini dans ce papier est plus précis que la notion d'accessibilité utilisée ici : l'indécidabilité est obtenue quand il existe deux processus dont les étiquettes des chemins menant de l'environnement à ces processus sont incomparables au sens ensembliste. Cette définition n'est pas adaptée à notre cas. En effet considérons l'architecture $\mathfrak{A}$ suivante dans laquelle toutes les variables ont même domaine (la figure de gauche représente le graphe des processus, celle de droite, le graphe des variables) :




Elle contient clairement une *information fork* selon [6] : les variables $x_0$ et $x_1$ et les processus $P_1$ et $P_2$. Cependant on remarque que, dans notre cadre, il existe une réduction vers l'architecture $\mathfrak{A}'$ :




En effet, si on a une stratégie gagnante pour l'architecture $\mathfrak{A}$, alors la stratégie pour l'architecture $\mathfrak{A}'$ va simplement simuler la stratégie sur la variable $t$, et inversement, si il existe une stratégie gagnante pour l'architecture $\mathfrak{A}'$, on peut avoir la même stratégie sur les variables $y_0$ et $y_1$ dans l'architecture $\mathfrak{A}$ en faisant simplement passer la valeur de la variable $x_0$ sur $t$.

Supposons qu'il existe une stratégie distribuée $f_t, f_{y_0}, f_{y_1}$ sur $\mathfrak{A}$. Alors on pose : $g_{y_0}(\overline{s}_i(\{x_0, x_1\})) = f_{y_0}(\overline{f}_t(\overline{s}_i(x_0)), \overline{s}_i(x_1))$ et $g_{y_1}(\overline{s}_i(\{x_0, x_1\})) =$

$f_{y_1}(\overline{f}_t(\overline{s}_i(x_0)), \overline{s}_i(x_1))$, et si $f_t, f_{y_0}, f_{y_1}$ est gagnante, alors $g_{y_0}, g_{y_1}$ aussi. Inversement, s'il existe $g_{y_0}, g_{y_1}$ stratégie distribuée gagnante pour $\mathfrak{A}'$, alors on pose $f_t(\overline{s}_i(x_0)) = s_i(x_0)$, $f_{y_0} = g_{y_0}$ et $f_{y_1} = g_{y_1}$, qui sont correctement définies car pour tout $i$, $s_i(t) = f_t(\overline{s}_i(x_0)) = s_i(x_0)$.

L'architecture $\mathfrak{A}'$ étant l'architecture singleton, elle est décidable, et donc $\mathfrak{A}$ est décidable. Cette différence vient du type de spécifications considérées ; dans [6], Finkbeiner et Schewe s'intéressent à des spécifications contraignant toutes les variables du système, tandis qu'ici, les variables internes sont laissées libres. C'est pourquoi la stratégie construite dans l'architecture $\mathfrak{A}$ de l'exemple est gagnante : la spécification est la même pour les deux architectures et la variable $t$ peut prendre la valeur que l'on veut. De façon générale, l'information provenant de l'environnement peut donc être transmise par deux canaux étiquetés différemment, si les stratgies l'autorisent, et ce, quelle que soit la spécification. C'est pourquoi le critère d'*information fork* n'est pas pertinent dans ce cadre.

### 3.3.2 Réduction des architectures à bande passante maximale

L'exemple de réduction présenté à la section 3.3.1 se généralise à toutes les architectures à bande passante maximale. Plus précisément, on a la proposition suivante :

**Proposition 3.3.4**

*Soit $\mathcal{V} = (V, A)$ avec $V = X \uplus Y \uplus T$ un graphe de variables. Alors $\mathcal{V}$ se réduit à $\mathcal{V}' = (V', A')$ avec $V' = X \uplus Y$ et $A' = Acc \cap (X \times Y)$.*

**Démonstration :** Soit $(f_z)_{z \in Y \uplus T}$ une stratégie distribuée sur $\mathcal{V}$. Pour unifier les notations on pose artificiellement pour tout $x \in X$, $f'_x(\overline{s}_i(x)) = s_i(x)$. On définit ensuite inductivement sur $t \in T \uplus Y$ sur le graphe (acyclique) $\mathcal{V}$ les fonctions $f'_t$ ainsi :

$$f'_t(\overline{s}_i[Acc^{-1}(t) \cap X]) = f_t(\overline{f'}_v(\overline{s}_i[Acc^{-1}(v) \cap X]))_{v \in R(z)}.$$

Et on pose pour tout $z \in Z$, $g_z(\overline{s}_i[R'(z)]) = f'_z(\overline{s}_i[Acc^{-1}(z) \cap X])$. La fonction est bien définie car $R'(z) = Acc^{-1}(z) \cap X$. On montre maintenant que $(g_z)_{z \in Y \uplus T}$ est équivalente à $(f_z)_{z \in Y \uplus T}$, c'est-à-dire que pour tout $i \in \mathbb{N}$, pour tout $z \in Z$, $g_z(\overline{s}_i[R'(z)]) = f_z(\overline{s}_i[R(z)]$. En effet, soit $t \in T \uplus Y$ tel que $R(t) \subseteq X$. Alors $f'_t(\overline{s}_i[Acc^{-1}(t) \cap X]) = f_t(\overline{f'}_v(\overline{s}_i[R(v)]))_{v \in R(z)})$ par définition et donc $f'_t(\overline{s}_i[Acc^{-1}(t) \cap X]) = f'_t(\overline{s}_i[R(t)]) = f_t(\overline{s}_i[R(t)])$. Soit $t \in T \cup Y$, $f'_t(\overline{s}_i[Acc^{-1}(t) \cap X]) = f_t(\overline{f'}_v(\overline{s}_i[R(v)]))_{v \in R(z)})$, donc, par induction, pour tout $v$ tel que $v \in R(t)$, $f'_v(\overline{s}_i[Acc^{-1}(v) \cap X]) = f_v(\overline{s}_i[R(v)])$. Donc, $f'_t(\overline{s}_i[Acc^{-1}(t) \cap X]) = f_t(\overline{f}_v(\overline{s}_i[R(v)])_{v \in R(z)} = f_t(\overline{s}_i[R(t)])$ par définition

d'un $f$-run. Donc, pour tout $z \in Y \uplus T$, $g_z(\overline{s}_i[R'(z)]) = f'_z(\overline{s}_i[Acc^{-1}(z) \cap X])$. Intuitivement, on a construit des stratégies qui simulent celles qu'avaient les variables internes en les calculant directement sur les fils de sortie.

Inversement, soit $(g_z)_{z \in Y}$ stratégie distribuée sur $\mathcal{V}'$. Alors, on pose pour tout $t \in T$, avec $D(t) = \prod_{x \in X}(D(x) \cup \{\bot\})$, $f_t(\bar{s}_i[R(t)]) = \bigsqcup_{v \in R(t)} s_i(v)$ (et non $f_t(\bar{s}_i[R(t)]) = s_i[R(t)]$). Notons que, dans un $f$-run, pour tout $t \in T \cup Y$, $s_i(t) = f_t(\overline{s}_i[R(t)]) = s_i[R(t)]$. Donc, par induction, pour tout $i \in \mathbb{N}$, pour tout $t \in T \cup Y$, $s_i(t) = s_i[Acc^{-1}(t) \cap X]$. On peut donc définir, pour tout $y \in Y$ $f_y(\overline{s}_i[R(y)]) = f_y(\overline{s}_i[Acc^{-1}(y) \cap X]) = f_y(\overline{s}_i[R'(y)]) = g_y(\overline{s}_i[R'(y]])$, et on a immédiatement que la stratégie distribu'ee sur $\mathcal{V}$ $(f_z)_{z \in Y \cup T}$ est équivalente à la stratégie distribuée $(g_z)_{z \in Y}$ sur $\mathcal{V}'$.

On peut donc conclure que tout graphe de variables $\mathcal{V} = (X \uplus Y \uplus T, A)$ est équivalent à un graphe de variables $\mathcal{V}' = (X \uplus Y, A')$ dans lequel les arêtes de $A'$ relient les variables selon leur accessibilité dans le graphe initial. ■

### 3.3.3 Procédure de synthèse pour les architectures décidables

On remarque que la réduction précédente ne change pas le critère : les deux graphes (avant et après réduction) sont équivalents pour la décidabilité et la réalisabilité – si je sais construire une stratégie distribuée pour l'un, je sais construire une stratégie distribuée pour l'autre qui induit les mêmes runs, et inversement.

On va montrer que les architectures à bande passante maximale ne contenant pas d'*information incomparable* sont décidables.

#### Fusion de variables de sortie ayant accès à la même information

Une architecture ne contenant pas d'information incomparable peut être totalement pré-ordonnée pour les variables de $Y$ (d'après le lemme 3.3.3). Les variables $y_0$ et $y_1$ de $Y$ telles que $y_0 \leq y_1$ et $y_1 \leq y_0$ peuvent en fait être fusionnées. En effet elles ont accès à la même quantité d'information, et les stratégies associées vont avoir le même domaine de définition. Formellement, à partir d'un graphe de variables simplifié $\mathcal{V} = (V = X \uplus Y, A)$, et d'une fonction $\nu : Y \to \mathbb{N}_n$ qui pré-ordonne les variables, on construit un graphe de variables $\mathcal{V}' = (V' = X \uplus Y', A' = \{(x, \nu(y)) \mid (x, y) \in A\})$ avec $Y' = \mathbb{N}_n$ et pour tout $i \in Y'$, $D(i) = \prod_{y \in \nu^{-1}(i)} D(y)$.

**Lemme 3.3.5**

*$\mathcal{V}$ est réalisable pour $\phi$ formule du $\mu$-calcul si et seulement si $\mathcal{V}'$ est réalisable pour $\phi$.*

**Démonstration :** Soit $(f_y)_{y\in Y}$ une stratégie distribuée pour $\mathcal{V}$. On construit une stratégie distribuée $(g_i)_{i\in\mathbb{N}_n}$ pour $\mathcal{V}'$. Sur un graphe de variables de la forme $V = X \uplus Y$, une stratégie distribuée $f$ s'écrit comme $f = (f_y)_{y\in Y}$. En effet, il n'y a pas de variable intermédiaire dont on aurait besoin de calculer la valeur pour appliquer la stratégie sur un fil de sortie. La stratégie globale, qui lie les variables d'entrée aux variables de sortie, est donc dans ce cas précis simplement la juxtaposition des stratégies locales $f_y$, pour $y \in Y$ sur $R(y) \subseteq X$. On commence remarque que, si $y_1$ et $y_2 \in \nu^{-1}(i)$, alors par définition, $Acc^{-1}(y_1) \cap X = Acc^{-1}(y_2) \cap X$ c'est-à-dire $R(y_1) = R(y_2)$. On peut donc poser pour tout $i \in \mathbb{N}_n$, $g_i = (f_y)_{y\in\nu^{-1}(i)}$. On a donc $g = \prod_{i\in\mathbb{N}_n} g_i = \prod_{y\in Y} f_y = f$ et en particulier $f \vDash \phi$ si et seulement si $g \vDash \phi$.

Inversement, soit $(g_i)_{i\in\mathbb{N}_n}$ stratégie distribuée sur $\mathcal{V}'$. Pour tout $y \in Y$, on définit $f_y$ comme la projection sur $D(y)$ de $g_{\nu(y)}$ et on a immédiatement que $g = f$. ■

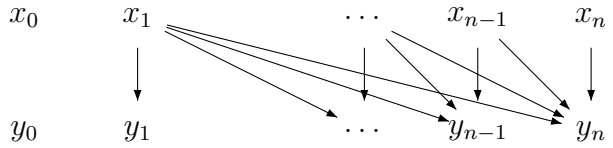
### Fusion des variables d'entrée reliées aux mêmes variables de sortie

En réduisant d'abord l'architecture à une architecture sans variables internes, puis en fusionnant les variables ayant accès à la même quantité d'information, on obtient une architecture dans laquelle on peut totalement ordonner les éléments de $Y$ : $y_1 < y_2 < \ldots < y_n$ donc par définition $R(y_1) \subsetneq R(y_2) \subsetneq \ldots \subsetneq R(y_n)$. On définit un pré-ordre sur les variables de $X$ tel que $x_1 \leq x_2$ si et seulement si $Acc(x_2) \subseteq Acc(x_1)$. L'architecture étant totalement ordonnée pour les variables de sortie, elle est également totalement pré-ordonnée pour les variables d'entrée selon l'ordre ci-dessus. Soit $\mu$ la fonction qui ordonne les variables de $X$. On a $\mu : X \to \mathbb{N}_n$ et à partir d'une architecture $\mathcal{V} = (X \uplus Y, A)$ déjà totalement ordonnée sur $Y$, on construit une architecture $\mathcal{V}' = (\mathbb{N}_n \uplus Y, A')$ avec $(i, y) \in A'$ si et seulement si $\mu(R(y)) = \{i\}$ ($A' = \{(i, y)/\forall x \in \mu^{-1}(i), (x, y) \in A\}$. Pour tout $i \in \mathbb{N}$, $D(i) = \prod_{x\in\mu^{-1}(i)} D(x)$. On montre maintenant que pour tout $y \in Y$, $\prod_{x\in R(y)} D(x) = \prod_{x'\in R'(y)} D(x')$. En effet, $\prod_{x'\in R'(y)} D(x') = \prod_{x'\in A'^{-1}(y)} \prod_{x\in\mu^{-1}(x')} D(x) = \prod_{x\in\{\mu^{-1}(x')|x'\in A'^{-1}(y)\}} D(x)$. Or, par définition de $A'$, $\{\mu^{-1}(x') \mid x' \in A'^{-1}(y)\} = A^{-1}(y)$, donc $\prod_{x'\in R'(y)} D(x') = \prod_{x\in R(y)} D(x)$. Donc, s'il existe une stratégie distribuée $(f_y)_{y\in Y}$ sur $\mathcal{V}$ alors c'est également une stratégie distribuée sur $\mathcal{V}'$ et réciproquement, et les runs induits sont les mêmes. Donc $\mathcal{V}$ est réalisable si et seulement si $\mathcal{V}'$ est réalisable.

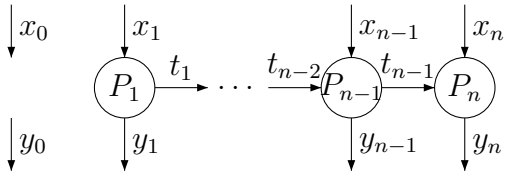
On constate que le critère est invariant pour ces transformations.

**Algorithme de décision**

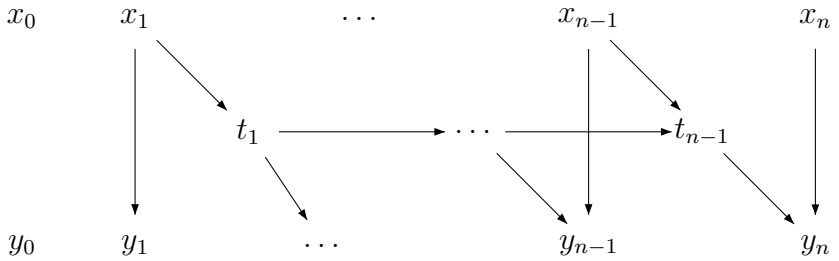
Après avoir effectué ces différentes transformations sur l'architecture, on obtient un graphe de variables de la forme :



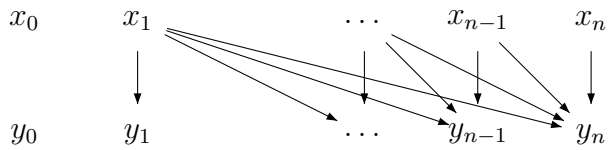

Les architectures de cette forme sont décidables. En effet, un graphe de variables n'est pas directement issu d'un graphe de processus dans notre cadre (c'est une généralisation de la figure 2.5 présentée à la section 2.1.1), mais on montre que c'est équivalent à l'architecture suivante :




En effet, si on transforme le graphe ci-dessus en graphe des variables, on obtient le graphe suivant :




graphe qui, après la réduction de la section 3.3.2, est égal à :




Pour résoudre le problème de synthèse, on peut donc se servir de façon inductive de la procédure décrite à la section 3.2 en sémantique 0-délai. Soit

$\psi$ formule du $\mu$-calcul. Par le théorème 3.2.1, il existe $\mathcal{A}_\psi$ automate d'arbres alternant sur des $(D(x_0) \times D(x_1) \times \cdots \times D(x_n), D(x_0) \times D(x_1) \times \cdots \times D(x_n) \times D(y_0) \cup \{\bot\} \times D(y_1) \cup \{\bot\} \times \cdots \times D(y_n) \cup \{\bot\})$-arbres, arbres acceptés si et seulement si ils vérifient la spécification $\psi$. On construit ensuite $\mathcal{A} = cover(\mathcal{A}_\psi)$, qui accepte les $(D(x_0) \times D(x_1) \times \cdots \times D(x_n), D(y_0) \cup \{\bot\} \times D(y_1) \cup \{\bot\} \times \cdots \times D(y_n) \cup \{\bot\})$-arbres, arbres correspondant aux arbres acceptés par $\mathcal{A}_\psi$ pour lesquels l'étiquetage correspondait bien à la direction du noeud et dans lesquels on a effacé la direction du noeud dans l'étiquette, puis $\mathcal{A}'$ automate d'arbres non déterministe équivalent à $\mathcal{A}$, et $\mathcal{A}_0 = narrow_{D(x_0)}(\mathcal{A}')$. Puis on construit, pour $1 \leq i \leq n$ les automates

- $\mathcal{A}_i = div^0_{D(y_{n-i+1})}(\mathcal{A}'_{i-1})$ (avec $\mathcal{A}'_0 = \mathcal{A}_0$), automate non déterministe acceptant des $(D(x_1)\times\cdots\times D(x_{n-i+1}), D(y_0)\cup\{\bot\}\times\cdots\times D(y_{n-i})\cup\{\bot\})$-arbres de stratégie globale pour les variables $y_0, \ldots, y_{n-i}$ ayant connaissance de la valeur de $x_{n-i+1}$, tels qu'il existe un arbre de stratégie pour la variable $y_{n-i+1}$ tel que la stratégie composée des deux soit acceptée par $\mathcal{A}'_{i-1}$,
- $\mathcal{A}'_i = narrow_{D(x_{n-i+1})}(\mathcal{A}_i)$, automate sur des $(D(x_1)\times\cdots\times D(x_{n-i}), D(y_0)\cup \{\bot\}\cdots \times D(y_{n-i})\cup\{\bot\})$-arbres, acceptant les arbres de stratégie globale pour les variables $y_0, \ldots, y_{n-i}$.

On remarque que cet algorithme est un passage à l'induction de la procédure présentée à la section 3.2.3. On a encore le résultat

**Théorème 3.3.6**

*Le problème de synthèse est réalisable si et seulement si le langage de l'automate $\mathcal{A}'_n$ est non vide.*

**Démonstration (Idée de preuve) :** Supposons qu'il existe $f_{y_0} : \emptyset \to D(y_0)\cup\{\bot\}$ appartenant à $\mathcal{L}(\mathcal{A}'_n)$. Alors $f_0 = wide_{D(x_1)}(f_{y_0})$ appartient à $\mathcal{L}(\mathcal{A}_n)$ et, d'après le théorème 3.2.8, il existe un arbre $f_{y_1} : D(x_1)^+ \to D(y_1)\cup\{\bot\}$ tel que $(f_0, f_{y_1}) = (f_{y_0}, f_{y_1}) \in \mathcal{L}(\mathcal{A}'_{n-1})$. On note $(f_{y_0}, f_{y_1})$ $\bar{f}_1$. Par induction, il existe $\bar{f}_i : (D(x_1) \times \ldots \times D(x_i))^+ \to (D(y_0) \cup \{\bot\} \times \ldots \times D(y_i) \cup \{\bot\}) \in \mathcal{L}(\mathcal{A}_{n-i})$, d'où $f_i = wide_{D(x_{i+1})}(\bar{f}_i) \in \mathcal{L}(\mathcal{A}_{n-i})$. Par le théorème 3.2.8, on sait qu'il existe $f_{y_i+1} : (D(x_1) \times \ldots \times D(x_{i+1}))^+ \to D(y_{i+1} \cup \{\bot\})$ tel que $(f_i, f_{y_{i+1}}) = (\bar{f}_i, f_{y_{i+1}}) = \bar{f}_{i+1} \in \mathcal{L}(\mathcal{A}'_{n-i-1})$. Finalement, on obtient l'existence de $f_{y_n} : (D(x_1) \times \ldots \times D(x_n))^+ \to D(y_n) \cup \{\bot\}$ tel que $(\bar{f}_{n-1}, f_{y_n}) \in \mathcal{L}(\mathcal{A}_0)$. Pour tout $0 \leq i \leq n$, $R(y_i) = \{x_1, \ldots, x_i\}$ et $f_{y_i}$ ainsi construit est tel que : $f_{y_i} : (D(x_1) \times \ldots \times D(x_i))^+ \to D(y_i) \cup \{\bot\}$ est bien une stratégie pour la variable $y_i$. La stratégie composée $(\bar{f}_{n-1}, f_{y_n})$ appartenant à $\mathcal{L}(\mathcal{A}_0)$, on a que la stratégie $f = wide_{D(x_0)}((\bar{f}_{n-1}, f_{y_n})$ qui est la stratégie globale du système telle qu'aucune variable ne dépende de la valeur de $x_0$ (qui ne communique avec personne) appartient à $\mathcal{A}$. Donc le run induit appartient à $\mathcal{A}_\psi$ et la stratégie $f$ est gagnante.

Inversement, supposons qu'il existe un ensemble de stratégies $f_{y_i}$ : $(D(x_1) \times \ldots \times D(x_i))^+ \rightarrow D(y_i)$ pour $0 \leq i \leq n$ telles que la stratégie composée soit gagnante. C'est-à-dire que $f = (f_{y_0}, f_{y_1}, \ldots, f_{y_n}) \in \mathcal{L}(\mathcal{A}_0)$. Alors, par application successive du théorème 3.2.8, on a que, pour $1 \leq i \leq n$, $(f_{y_0}, \ldots, f_{y_i}) \in \mathcal{L}(\mathcal{A}'_{n-i})$. Donc finalement, $f_{y_0} \in \mathcal{L}(\mathcal{A}'_n)$ et le langage de $\mathcal{A}'_n$ est donc non vide. ■

### 3.3.4 Architectures indécidables

Cette section montre que l'information incomparable est bien un critère d'indécidabilité. Nous le prouvons d'abord dans le cadre de la bande passante maximale, mais on verra en section 3.3.5 que le résultat est valide sur architectures quelconques, donnant ainsi une condition suffisante d'indécidabilité.

**Théorème 3.3.7**

*Les architectures à* information incomparable *sont indécidables pour les spécifications LTL.*

**Démonstration :** On note $\mathsf{X}$ et $\mathsf{U}$ les opérateurs *next* et *until* de LTL. On fait une réduction du problème de synthèse sur l'architecture $\mathfrak{A}_0$ introduite dans [20], rappelée dans les figures 3.4 et 3.5.

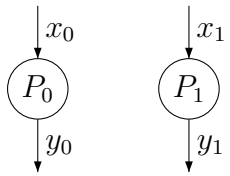



FIG. 3.4 – L'architecture $\mathfrak{A}_0$ - graphe de processus

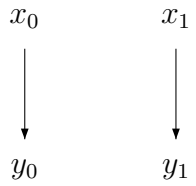



FIG. 3.5 – L'architecture $\mathfrak{A}_0$ - graphe de variables

Soit $\mathfrak{A}$ une architecture à information incomparable. Notons $x^0$ et $x^1$ ses variables d'entrée et $y^0, y^1$ ses variables de sortie définissant l'information incomparable : c'est-à-dire que $y^i \in Acc(x^i) \cap Y \backslash Acc(x^{1-i})$. Soit

$\bar{X} = \prod_{x \in X \setminus \{x^0, x^1\}} D(x)$ et $\bar{Y} = \prod_{y \in Y \setminus \{y^0, y^1\}} D(y)$. On pose $\Sigma = D(x^0) \times \bar{X} \times D(x^1) \times D(y^0) \times \bar{Y} \times D(y^1)$. Soit $\Sigma_0 = D(x^0) \times D(x^1) \times D(y^0) \times D(y^1)$. On définit une réduction *red* qui à une formule $\phi_0$ de LTL($\Sigma_0$) associe $\phi = red(\phi_0)$ de LTL($\Sigma$) définie par induction structurelle sur $\phi_0$ comme suit :

- si $\phi_0 = (x_0, x_1, y_0, y_1)$ pour $x_i \in D(x^i)$, $y_i \in D(y^i)$, alors on pose $red(\phi_0) = \bigvee_{\bar{x} \in \bar{X}, \bar{y} \in \bar{Y}} (x_0, \bar{x}, x_1, y_0, \bar{y}, y_1)$,
- si $\phi_0 = \psi_0 \vee \psi_1$ alors $red(\phi_0) = red(\psi_0) \vee red(\psi_1)$,
- si $\phi_0 = \neg\psi_0$ alors $red(\phi_0) = \neg red(\psi_0)$,
- si $\phi_0 = \mathsf{X}\psi_0$, alors $red(\phi_0) = \mathsf{X}(red(\psi_0))$,
- si $\phi_0 = \psi_0 \,\mathsf{U}\, \psi_1$ alors $red(\phi_0) = red(\psi_0) \,\mathsf{U}\, red(\psi_1)$.

La spécification $\phi$ définit un langage $L \subseteq \Sigma^\infty$. Soit $\pi_{\Sigma_0}$ la projection naturelle de $\Sigma$ sur $\Sigma_0$.

**Remarque 3.3.8**

*Un run $s_o \cdot s_1 \cdots$ satisfait $\phi = red(\phi_0)$ si et seulement si $\pi_{\Sigma_0}(s_0 \cdot s_1 \cdots)$ satisfait $\phi_0$.*

En effet,

- $s_0 \cdot s_1 \cdots \vDash \phi = \bigvee_{\bar{x} \in \bar{X}, \bar{y} \in \bar{Y}} (x_0, \bar{x}, x_1, y_0, \bar{y}, y_1)$ si et seulement si $s_0 \vDash \phi$ si et seulement si $\pi_{\Sigma_0}(s_0) \vDash (x_0, x_1, y_0, y_1)$ si et seulement si $\pi_{\Sigma_0}(s_0 \cdot s_1 \cdots) \vDash (x_0, x_1, y_0, y_1) = \phi_0$.
- $s_0 \cdot s_1 \cdots \vDash red(\neg\psi) = \neg red(\psi)$ si et seulement si $s_0 \cdot s_1 \cdots \nvDash red(\psi)$ si et seulement si (par induction) $\pi_{\Sigma_0}(s_0 \cdot s_1 \cdots) \nvDash \psi$ si et seulement si $\pi_{\Sigma_0}(s_0 \cdot s_1 \cdots) \vDash \neg\psi$.
- $s_0 \cdot s_1 \cdots \vDash red(\psi_0 \vee \psi_1) = red(\psi_0) \vee red(\psi_1)$ si et seulement si $s_0 \cdot s_1 \cdots \vDash red(\psi_0)$ ou $s_0 \cdot s_1 \cdots \vDash red(\psi_1)$, si et seulement si, par induction, $\pi_{\Sigma_0}(s_0 \cdot s_1 \cdots) \vDash \psi_0$ ou $\pi_{\Sigma_0}(s_0 \cdot s_1 \cdots) \vDash \psi_1$ si et seulement si $\pi_{\Sigma_0}(s_0 \cdot s_1 \cdots) \vDash \psi_0 \vee \psi_1$.
- $s_0 \cdot s_1 \cdots \vDash red(\mathsf{X}\psi) = \mathsf{X} red(\psi)$ si et seulement si $s_1 \cdots \vDash red(\psi)$ si et seulement si, par induction, $\pi_{\Sigma_0}(s_1 \cdots) \vDash \psi$ si et seulement si $\pi_{\Sigma_0}(s_0 \cdot s_1 \cdots) \vDash \mathsf{X}\psi$.
- $s_0 \cdot s_1 \cdots \vDash red(\psi_0 \,\mathsf{U}\, \psi_1) = red(\psi_0) \,\mathsf{U}\, red(\psi_1)$ si et seulement s'il existe $j$ tel que $s_j \cdot s_{j+1} \cdots \vDash red(\psi_1)$ et pour tout $k < j$, $s_k \vDash red(\psi_1) \cdots$ si et seulement si, par induction, $\pi_{\Sigma_0}(s_j \cdot s_{j+1} \cdots) \vDash \psi_1$ et pour tout $k < j$, $\pi_{\Sigma_0}(s_k) \cdots \vDash \psi_0$ si et seulement si $\pi_{\Sigma_0}(s_0 \cdot s_1 \cdots) \vDash \psi_0 \,\mathsf{U}\, \psi_1$.

On veut montrer que $\phi_0$ est réalisable sur $\mathfrak{A}_0$ si et seulement si $\phi$ est réalisable sur $\mathfrak{A}$. Grâce à la proposition 3.3.4, on peut supposer que $\mathfrak{A}$ ne possède aucune variable interne.

Soit $(f_{y^0}, f_{y^1})$ une stratégie distribuée sur $\mathfrak{A}_0$ réalisant $\phi_0$. Alors, on construit une stratégie $g$ distribuée sur $\mathfrak{A}$ qui réalise $\phi = red(\phi_0)$ en posant

$$g_{y^k}(\overline{s}_i[R(y^k)]) = f_{y^k}(\overline{s}_i(x^k)) \quad \text{pour } k = 0, 1, \tag{3.1}$$

et on complète $g$ arbitrairement sur les autres sorties de $\mathfrak{A}$. Notons que (3.1) définit bien des stratégies, car $\mathfrak{A}$ n'a pas de variable interne et $x^k \in$

$R(y^k)$. Soit $s_0 \cdot s_1 \cdots$ un $g$-run sur $\mathfrak{A}$. Par définition, pour $k = 0, 1$, on a $s_i(y^k) = g_{y^k}(\overline{s}_i[R(y^k)]) = f_{y^k}(\overline{s}_i(x^k))$, donc $\pi_{\Sigma_0}(s_0 \cdot s_1 \cdots)$ est un $f$-run sur $\mathfrak{A}_0$, qui satisfait donc $\phi_0$. D'après la remarque 3.3.8, $s_0 \cdot s_1 \cdots$ satisfait $\phi$. Finalement, $g$ réalise $\phi$.

Inversement, soit $(g_z)_{z \in Y}$ stratégie distribuée réalisant $\phi$. On va définir des stratégies $f_{y^k} : D(x^k)^+ \to D(y^k)$ $(k = 0, 1)$ sur $\mathfrak{A}_0$, satisfaisant $\phi_0$, qui simulent les stratégies jouées dans $\mathfrak{A}$ dans le cas où l'environnement ne joue que des 0 sur les entrées différentes de $x^0$ et $x^1$. Pour toute fonction $s$ d'interprétation des variables de $\mathfrak{A}_0$, on définit une fonction d'interprétation $s'$ des variables de $\mathfrak{A}$ telle que $s'(x^k) = s(x^k)$ pour $k = 0, 1$ et pour tout $x \in X \backslash \{x^0, x^1\}$, $s'(x) = 0$. On pose

$$f_{y^k}(\overline{s}_i(x^k)) = g_{y^k}(\overline{s}'_i[R(y^k)]) \quad \text{pour } k = 0, 1. \tag{3.2}$$

Clairement, si $s_0 \cdot s_1 \cdots$ est un $f$-run de $\mathfrak{A}_0$, alors $s'_0 \cdot s'_1 \cdots$ est un $g$-run de $\mathfrak{A}$ dont la projection sur $\Sigma_0$ est $s_0 \cdot s_1 \cdots$. Donc $s'_0 \cdot s'_1 \cdots$ satisfait $\phi = red(\phi_0)$, et à nouveau par la remarque 3.3.8, le $f$-run $s_0 \cdot s_1 \cdots = \pi_{\Sigma_0}(s'_0 \cdot s'_1 \cdots)$ satisfait $\phi_0$. Donc $f = (f_{y_0}, f_{y_1})$ réalise $\phi_0$.

La fonction *red* définit donc une réduction du problème de synthèse distribuée sur l'architecture $\mathfrak{A}_0$ vers le problème de synthèse distribuée sur l'architecture $\mathfrak{A}$, architecture donnée à information incomparable. Le problème de synthèse étant indécidable pour $\mathfrak{A}_0$, on obtient l'indécidabilité pour n'importe quelle architecture contenant un *information incomparable*. ■

### 3.3.5 Indécidabilité sur architectures quelconques

Le critère d'*information incomparable* est suffisant pour l'indécidabilité même dans le cas d'une architecture à bande passante non maximale. En effet la réduction d'une architecture à information incomparable vers l'architecture $\mathcal{A}_0$ de Pnueli-Rosner peut se faire dès qu'on peut transmettre séparément un bit d'information vers les deux sorties. La différence dans la preuve tient au fait qu'ici on ne peut pas a priori transformer le graphe de variables de façon à supprimer les variables internes, n'étant pas sûr de pouvoir transmettre toute l'information jusqu'aux variables de sortie. Il s'agit donc ici de ne transmettre le long des chemins que l'information dont on a besoin.

Plus formellement, on peut réécrire la réduction de la façon suivante : On se place dans le cas où, quel que soit $x \in X$, $|D(x)| = 2$. Soient $x^0$ et $x^1$ les variables de $X$ et $y^0, y^1$ les variables de $Y$ constituant l'information incomparable. La spécification définit un langage $L \subseteq \Sigma^\infty$. Soit $\Sigma_0 = D(x^0) \times D(x^1) \times D(y^0) \times D(y^1)$. On définit une réduction qui à une formule $\phi_0 \in \mathrm{LTL}(\Sigma_0)$ associe $\phi = red(\phi_0)$ une formule de $\mathrm{LTL}(\Sigma)$ de la même façon qu'à la section 3.3.4. La remarque 3.3.8 reste vraie.

On remarque tout d'abord que $\forall x \in X, \forall t \in T, |D(t)| \geq |D(x)|$, et on peut donc définir des injections $f_{x,t} : D(x) \to D(t)$. Pour plus de simplicité, on supposera par la suite que $\forall x \in X, \forall t \in T, D(x) \subseteq D(t)$.

Supposons qu'il existe une stratégie distribuée $\{f_{y^0}, f_{y^1}\}$ sur $\mathfrak{A}_0$ satisfaisant $\phi_0$, alors il existe une stratégie distribuée sur $\mathfrak{A}$ qui satisfait $\phi = red(\phi_0)$. En effet, soit $t_1^0, \ldots, t_n^0$ et $t_1^1, \ldots, t_m^1$ deux suites *disjointes* d'étiquettes d'arêtes telles que $(x^0, t_1^0), (x^1, t_1^1), (t_n^0, y^0), (t_m^1, y^1) \in A$, et, pour tout $1 < i < n$, $1 < j < m$, $(t_i^0, t_{i+1}^0) \in A$, $(t_j^1, t_{j+1}^1) \in A$. Deux telles suites existent car $y^i \in Acc(x^i) \backslash Acc(x^{1-i}, i = 0, 1$. Alors on définit des stratégies locales sur les chemins ainsi distingués qui vont transmettre l'information donnée par l'environnement sur l'entrée $x^j$, $j = 0, 1$ jusqu'à la stratégie écrivant sur le canal $y^j$, qui va elle se comporter comme $f_{y^j}$.
Formellement, pour $1 \leq k \leq n$, on définit $g_{t_k^0}$ que l'on note $g_k$ pour simplifier les notations. $g_k$ est une stratégie sans mémoire telle que pour $k < n$,

$$\begin{aligned} g_k(s) &= s(t_{k-1}^0) \\ g_{y^0}(\overline{s}_i[\{t \in R(y^0)]) &= \begin{cases} f_{y^0}(\overline{s}_i[\{t \in R(y^0)](t_n^0) & \text{si } \overline{s}_i[\{t \in R(y^0)](t_n^0) \in D(x^0)^+ \\ 0 & \text{sinon} \end{cases} \end{aligned}$$

On remarque que, si les processus se comportent suivant leurs stratégies, on a, pour tout $1 \leq k \leq n$, $g_k(s_i[\{R(t_k^0)\}]) = s_i(t_{k-1}^0) = g_{k-1}(s_i[\{R(t_{k-2}^0]) = \ldots = s_i(x^0)$. Donc, $g_{y^0}(\bar{s}_i[\{R(y^0)\}]) = f_{y^0}(\bar{s}_i(t_n^0)) = f_{y^0}(\bar{s}_i(x_0))$. On fait de même pour les $g_{t_j^1}$ et $g_{y^1}$. Pour toutes les autres variables $z \in T \cup Y$, on définit par exemple $g_z = 0$. Soit $\tilde{s}_0 \cdot \tilde{s}_1 \cdots$ un $g$-run sur $\mathfrak{A}$. Alors, pour tout $i$, pour tout $z \in X \cup Y \cup T$, $\tilde{s}_i(z) = g_z(\overline{\tilde{s}}_i(R(z)))$. Donc $\pi_{\{x^0, z^1, y^0, y^1\}}(\tilde{s}_i) = s_i$ avec

$$\begin{aligned} s : \{x^0, x^1, y^0, y^1\} &\to D(x^0) \times D(x^1) \times D(y^0) \times D(y^1) \\ s(z) &= \tilde{s}(z) \text{ pour } z \in \{x^0, x^1, y^0, y^1\} \end{aligned}$$

Donc $s_i(y^j) = g_{y^j}(\overline{\tilde{s}}_i(R(y^j))) = f_{y^j}(\overline{\tilde{s}}_i(x^j)) = f_{y^j}(\overline{s}_i(x^j))$ pour $j = 0, 1$, et $s_0 \cdot s_1 \cdots$ est bien un $f$-run sur $\mathfrak{A}_0$. Comme $\{f_{y^0}, f_{y^1}\}$ est gagnante pour $\phi_0$, alors $\{g_z, z \in V\}$ est gagnante pour $red(\phi_0)$ (d'après la remarque) et il existe une stratégie distribuée gagnante pour $\phi$.

Inversement, supposons qu'il existe $\{g_z | z \in Y \cup T\}$ stratégie distribuée satisfaisant $\phi$. On va définir $f_{y^0} : D(x^0)^+ \to D(y^0)$ et $f_{y^1} : D(x^1)^+ \to D(y^1)$. Les $f_{y^i}$ sont les stratégies pour l'architecture $\mathfrak{A}_0$ satisfaisant $\phi_0$. L'idée est que les processus de $\mathfrak{A}_0$ vont simuler les stratégies jouées sur toutes les variables appartenant respectivement à $Acc^{-1}(y^i), i = 0, 1$ dans le cas où l'environnement ne joue que des 0 sur les entrées différentes de $x^0$ et $x^1$. On définit inductivement sur $z \in Acc^{-1}(Y)$ (induction possible car le graphe est

acyclique) des fonctions $f_z$ :

$$\begin{array}{rcl} f_z : (D(x^0) \times D(x^1))^+ & \rightarrow & D(z) \\ f_{x^0}(\overline{s}_i[\{x^0, x^1\}]) & = & s_i(x^0) \\ f_{x^1}(\overline{s}_i[\{x^0, x^1\}]) & = & s_i(x^1) \\ f_x((\overline{s}_i[\{x^0, x^1\}]) & = & 0 \in D(x) \text{ pour } x \in X \backslash \{x^0, x^1\} \\ f_z((\overline{s}_i[\{x^0, x^1\}]) & = & g_z(\overline{f}_v(\overline{s}_i[\{x^0, x^1\}])_{v \in R(z)}) \end{array}$$

**Remarque :** $f_{y^j}$ ne dépend que de $x^j$, $j = 0, 1$.

En effet, on montre par induction que si $z$ n'est pas dans $Acc(x^j)$ alors $f_z$ ne dépend pas de $x^j$, $j = 0, 1$. Pour $x \in X \setminus \{x_j\}$, c'est évident. Si $z \in T \cup Y \setminus Acc(x^j)$, alors pour tout $v \in R(z)$ on a $v \notin Acc(x^j)$. La stratégie pour la variable $z$ est définie $f_z((\overline{s}_i[\{x^0, x^1\}]) = g_z(\overline{f}_v(\overline{s}_i[\{x^0, x^1\}])_{v \in R(z)})$. Par hypothèse d'induction, $f_v(\overline{s}_i[\{x^0, x^1\}])$ ne dépend pas de $x^j$, pour tout $v \in R(z)$, donc $f_z$ ne dépend pas non plus de $x^j$. Finalement, $y^{1-j}$ n'appartient pas à $Acc(x^j)$, et donc $f_{y^{1-j}}$ ne dépend pas de $x^j$. Donc les fonctions $f_{y^0}$ et $f_{y^1}$ sont bien des stratégies pour $\mathfrak{A}_0$. Soit $s_0 \cdot s_1 \cdots$ un $f$-run de $\mathfrak{A}_0$. On définit $s'_0 \cdot s'_1 \cdots$ $g$-run de $\mathfrak{A}$ tel que

$$\begin{array}{rcl} s'_i : V & \rightarrow & \cup_{z \in V} D(z) \\ s'_i(x^j) & = & s_i(x^j) \text{ pour } j = 0, 1 \\ s'_i(x) & = & 0 \text{ pour } x \in X \backslash \{x^0, x^1\} \end{array}$$

Soit $z \in T \cup Y$, $s'_i(z) = g_z(\overline{s'}_i[R(z)])$ par définition d'un $g$-run. Par induction, pour tout $v \in R(z)$, pour tout $i$, $s'_i(v) = f_v(\overline{s}_i[\{x^0, x^1\}])$, donc $s'_i(z) = g_z(\overline{f}_v(\overline{s}_i[\{x^0, x^1\}])_{v \in R(z)}) = f_z((\overline{s}_i[\{x^0, x^1\}])$. Donc $s'_i(y^j) = f_{y^j}((\overline{s}_i[\{x^0, x^1\}]) = s_i(y^j)$ pour $j = 0, 1$. Donc pour tout $f$-run $s_0 \cdot s_1 \cdots$ sur $\mathfrak{A}_0$, il existe un $g$-run $s'_0 \cdot s'_1 \cdots$ sur $\mathfrak{A}$ tel que $\pi_{\{x^0, x^1, y^0, y^1\}}(s'_0 \cdot s'_1 \cdots) = s_0 \cdot s_1 \cdots$. Comme tout $g$-run est gagnant, alors $s' \vDash \phi = red(\phi_0)$ et $s = \pi_{\{x^0, x^1, y^0, y^1\}}(s') \vDash \phi_0$ et $\{f_{y^0}, f_{y^1}\}$ est une stratégie distribuée gagnante.

On a donc une réduction du problème de synthèse distribuée sur l'architecture $\mathfrak{A}_0$ vers le problème de synthèse distribuée sur n'importe quelle architecture à information incomparable. Le problème de synthèse étant indécidable pour $\mathfrak{A}_0$, on obtient l'indécidabilité pour n'importe quelle architecture à information incomparable. En effet, on constate que les seules informations transmises sont les variables $x^0$ et $x^1$, que l'indécidabilité de $\mathfrak{A}_0$ est conservée si $|D(x^0)| = |D(x^1)| = 2$, c'est-à-dire quand l'information transmise est la plus petite possible, et on obtient donc l'indécidabilité également pour les architectures à bande passante fixée à information incomparable.

# Chapitre 4

# Conclusion et perspectives

Nous avons donc au cours du stage étudié un certain nombre de techniques utilisées pour résoudre les problèmes de synthèse de contrôleur dans le cas distribué. L'indécidabilité arrive très vite lorsque l'information transmise par l'environnement se divise en plusieurs branches distinctes, mais nous avons démontré un critère nécessaire et suffisant de décidabilité plus fin que celui de [6] pour des spécifications ne contraignant que les variables externes du système, dans le cas 0-délai, et pour des architectures à bande passante maximale. La difficulté *technique* principale pour prouver la décidabilité d'architectures dans lesquelles l'environnement communique avec plusieurs processus de façon séparée vient du fait que les informations de l'environnement sont stockées dans les noeuds des arbres de stratégie, et les techniques d'automates d'arbres habituellement utilisées sont inadaptées pour suivre ces « fourches » de l'information.

Nous aimerions à présent étendre ce résultat aux architectures quelconques - i.e. à bande passante fixée. Une étape dans la réalisation de cet objectif est de résoudre l'architecture $\mathfrak{A}_2$ de la figure 4.1.

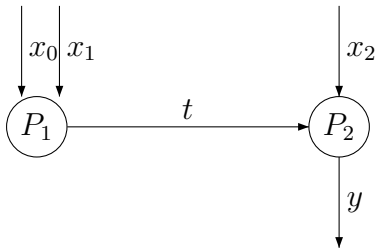



FIG. 4.1 – L'architecture $\mathfrak{A}_2$

En effet, nous savons depuis [20] que l'architecture de la figure 4.2 est in-

décidable pour des spécifications LTL (la preuve repose sur la même idée que la preuve d'indécidabilité de $\mathfrak{A}_0$ présentée à la section 2.2 avec la variation que la spécification cette fois impose que, lorsque $x_0$ transmet la valeur 1, sur le canal de sortie soit écrite une configuration de la machine de Turing *encodée par* $x_1$, ce qui fait que le processus $P_0$ doit transmettre la configuration déjà encodée sur $t$ et $P_1$ ne connaît pas plus la valeur de $x_0$ que dans l'architecture $\mathfrak{A}_0$).

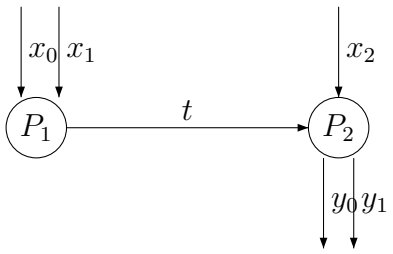



FIG. 4.2 – Architecture indécidable

Cette preuve d'indécidabilité ne peut pas être appliquée telle quelle pour l'architecture $\mathfrak{A}_2$, car cette architecture ne possède qu'un fil de sortie et on ne peut donc forcer avec des spécifications en logique temporelle la simulation d'une machine de Turing. Les différentes contraintes que devraient respecter des stratégies locales synthétisées sur les variables de $\mathfrak{A}_2$ nous poussent à regarder les preuves d'indécidabilité pour les arbres à contraintes d'égalité.

Une autre direction de recherche que nous aimerions explorer serait de trouver un critère de décidabilité pour les architectures à bande passante maximale, mais avec une sémantique 1-délai. En effet, l'introduction du délai dans une architecture comme celle présentée à la figure 4.3 si elle est à bande passante maximale, peut transmettre l'information jusqu'à la variable $y$, mais que se passe-t-il si les informations sont transmises de façon croisée ?

Enfin, ainsi que nous l'avons fait remarquer en section 2.2, la principale source d'indécidabilité provient du fait que les spécifications LTL sont *globales*, alors que les stratégies recherchées sont *locales*. Nous aimerions donc restreindre le langage de spécification de telle sorte que dans ce nouveau modèle, toutes les architectures soient décidables. Une piste serait d'adapter le critère de spécification causale de [7] en tenant compte des bandes passantes.

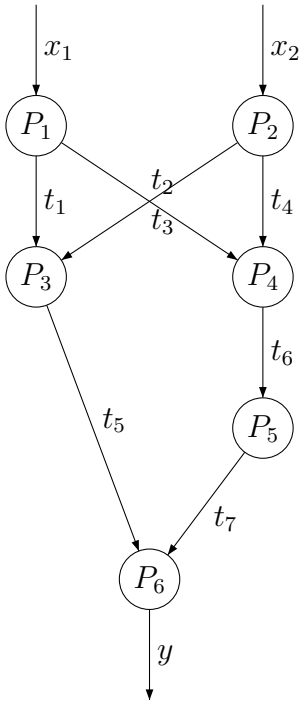



FIG. 4.3 –

# Bibliographie


[1] A. Arnold, A. Vincent, and I. Walukiewicz. Games for synthesis of controllers with partial observation. *Theoret. Comput. Sci.*, 303 :7–34, 2003.

[2] E. Asarin, O. Maler, and A. Pnueli. Symbolic controller synthesis for discrete and timed systems. In *Hybrid Systems*, pages 1–20, 1994.

[3] B. Berard, M. Bidoit, A. Finkel, A. Laroussinie, F.and Petit, L. Petrucci, and P. Schnoebelen. *Systems and Software Verification. Model-Checking Techniques and Tools*. Springer, 2001.

[4] E. Clarke, O. Grumberg, and D. Peled. *Model Checking*. MIT Press, 2000.

[5] A. P. E. Asarin, O. Maler and J. Sifakis. Controller synthesis for timed automata. In *Proc. System Structure and Control*. Elsevier, 1998.

[6] B. Finkbeiner and S. Schewe. Uniform distributed synthesis. In *Proc. 20th Annual IEEE Symposium on Logic in Computer Science (LICS 2005)*. IEEE Computer Society, 2005.

[7] P. Gastin, B. Lerman, and M. Zeitoun. Causal memory distributed games are decidable for series-parallel systems. In *FSTTCS'04*, 2004.

[8] P. Gastin, B. Lerman, and M. Zeitoun. Distributed games and distributed control for asynchronous systems. In M. Farach-Colton, editor, *Proceedings of LATIN '04*, volume 2976 of *Lect. Notes Comp. Sci.*, pages 455–465. Springer, 2004.

[9] D. Kirsten. Alternating tree automata and parity games. *Lect. Notes Comp. Sci.*, pages 153–167, 2002.

[10] O. Kupferman and M. Vardi. $\mu$-calculus synthesis. In *Proc. 25th International Symposium on Mathematical Foundations of Computer Science*,

volume 1893 of *Lecture Notes in Computer Science*, pages 497–507. Springer-Verlag, 2000.

[11] O. Kupferman and M. Vardi. Synthesizing distributed systems. In *Proceedings of LICS '01*. Computer Society Press, 2001.

[12] O. Kupferman and M. Vardi. Church's problem revisited. *The Bulletin of Symbolic Logic*, 5(2) :245–263, June 1999.

[13] O. Kupferman, M. Vardi, and P. Wolper. An automata-theoretic approach to branching-time model checking. *J. ACM*, 47 :312–360, March 2000.

[14] P. Madhusudan and P. Thiagarajan. Distributed controller synthesis for local specifications. In *Proceedings of ICALP '01*, volume 2076 of *Lect. Notes Comp. Sci.*, pages 396–407. Springer, 2001.

[15] P. Madhusudan and P. Thiagarajan. A decidable class of asynchronous distributed controllers. In *Proceedings of CONCUR '02*, volume 2421 of *Lect. Notes Comp. Sci.* Springer, 2002.

[16] S. Mohalik and I. Walukiewicz. Distributed games. In *Proceedings of FSTTCS '03*, volume 2914 of *Lect. Notes Comp. Sci.* Springer, 2003.

[17] D. Muller and P. Schupp. Alternating automata on infinite trees. *Theoret. Comput. Sci.*, 54 :267–276, 1987.

[18] D. Muller and P. Schupp. Simulating alternating tree automata by nondeterministic automata : New results and new proofs of theorems of rabin, mcnaughton and safra. *Theoret. Comput. Sci.*, 2(1) :90–121, 1995.

[19] G. Peterson and J. Reif. Multiple-person alternation. In *20th Annual Symposium on Foundations of Computer Science (San Juan, Puerto Rico, 1979)*, pages 348–363. IEEE, New York, 1979.

[20] A. Pnueli and R. Rosner. Distributed reactive systems are hard to synthetize. In *Proceedings of 31th IEEE Symp. FOCS*, pages 746–757, 1990.

[21] P. Ramadge and W. Wonham. The control of discrete event systems. In *Proceedings of IEEE*, volume 77, pages 81–98, 1989.