THÈSE

présentée à l'École Normale Supérieure de Cachan

pour obtenir le grade de

Docteur de l'École Normale Supérieure de Cachan

par : Nathalie Sznajder

Spécialité : Informatique

Synthèse de systèmes distribués ouverts

Soutenue le 12 novembre 2009, devant un jury composé de :

– Ahmed Bouajjani	président du jury
– Paul Gastin	directeur de thèse
– Claude Jard	examinateur
– Jean-François Raskin	rapporteur
– Igor Walukiewicz	rapporteur
– Marc Zeitoun	examinateur

Résumé

Cette thèse est un travail sur la synthèse de systèmes distribués ouverts. Dans le problème de synthèse, on considère un système en interaction avec un environnement incontrôlable, et on se donne une description formelle de ses comportements corrects (la *spécification*). On cherche alors à dériver automatiquement un programme pour le système dont toutes les exécutions satisfont la spécification et ce, quelle que soit la façon dont l'environnement se comporte. Pour le problème de synthèse de systèmes distribués, on se donne en plus de la spécification une description des communications possibles entre les différents processus du système, ainsi que des communications entre l'environnement et les processus (l'*architecture*). Le but est alors de synthétiser automatiquement un programme pour chaque processus du système. Ce dernier problème est indécidable en général, et le travail effectué au cours de cette thèse vise à comprendre les principales causes d'indécidabilité afin de proposer les restrictions les plus naturelles possibles sous lesquelles le problème serait décidable. On choisit en particulier de se restreindre aux spécifications de type externe, c'est-à-dire ne restreignant que les interactions entre le système et l'environnement, en laissant non spécifiées les interactions entre les processus eux-mêmes. Dans un premier chapitre, on définit un modèle général présentant de manière uniforme les différents résultats de la littérature sur le sujet, qui étaient exprimés dans des formalismes divers. Ensuite on expose les résultats obtenus dans le cas de systèmes distribués synchrones. On propose une condition nécessaire sur l'architecture des systèmes pour la décidabilité de ce problème, puis on définit une classe d'architectures (les architectures uniformément bien connectées), pour laquelle cette condition devient un critère nécessaire et suffisant. Finalement, on essaie d'étendre cette classe en définissant la classe des architectures bien connectées, et on montre que le critère établi n'est plus une condition suffisante pour la décidabilité du problème sur cette classe plus large. La preuve d'indécidabilité apporte des éléments nouveaux sur les causes d'indécidabilité, permettant d'améliorer la compréhension du problème. Le dernier chapitre présente les travaux effectués dans le cas de systèmes ayant un comportement asynchrone. On introduit un nouveau modèle de communication : la communication par signaux ; un signal est une action commune à deux processus, mais qui n'est contrôlée que par un seul. On s'intéresse à des spécifications externes portant sur des ordres partiels, et on définit des propriétés de clôture que les spécifications doivent respecter. On montre que dans ce cadre le problème est décidable pour tous les systèmes dont le graphe de communication est fortement connexe.

Abstract

We study the synthesis problem for distributed open systems. Synthesis problems consists in, given a formal description of correct behaviors of a system interacting with an uncontrollable environment (the *specification*), automatically deriving a program for this system such that all its behaviors meet the specification, regardless how the environment behaves. In synthesis of distributed systems, one is given along with the specification, a description of the communications between the different processes of the system, and of the communications between the processes and the environment (the *architecture*). The goal is in that case to produce a program for each process. In general this last problem is undecidable. The work presented here aims at clarifying the causes of undecidability, in order to define the most natural restrictions leading to decidable subclasses of the problem. We consider in particular external specifications, i.e., that constrain only communications between the system and the environment, while communications between the different processes of the architecture are left unrestricted. In a first chapter, we define a model for the synthesis problem for distributed systems allowing to present in a uniform setting the different results of the litterature. Then we give the results obtained for synchronous distribued systems. We determine a necessary condition on the shape of the architectures for decidability, and we introduce a class of systems (uniformly well connected architectures), for which this condition becomes a necessary and sufficient criterion. Finally, we introduce the larger class of well connected architectures, and we show that our criterion is not anymore a sufficient condition for this class. The undecidability proof gives new insights on the causes of undecidability, letting us to better understand the problem. The last chapter presents the work done on asynchronous systems. We define a new model of communication : communications by signals ; a signal is a common action between two processes, but initiated by only one of them. We consider external specifications over partial orders, and also ask for some closure properties for the specifications. We then show that this setting gives decidability results for the class of systems where communication happens through a strongly connected graph.

Remerciements

Je tiens à remercier Ahmed Bouajjani, Claude Jard et Marc Zeitoun d'avoir accepté de faire partie de mon jury de thèse, ainsi que Jean-François Raskin et Igor Walukiewicz de m'avoir faire l'honneur d'être rapporteurs. Marc a par ailleurs accepté de co-encadrer mon stage de Master 2, et j'ai ainsi pu bénéficier de sa gentillesse et de ses explications éclairantes durant sa dernière année à Paris. Par dessus tout je voudrais remercier très sincèrement Paul Gastin pour son encadrement au cours de ces premières années de recherche. Sa rigueur et sa pédagogie (ainsi que sa louable patience !) ont été pour moi des guides précieux. J'espère avoir un peu appris ces qualités-là. Je le remercie également d'avoir toujours su dégager du temps lorsque j'en ai eu besoin.

J'ai eu la chance de passer ces années de thèse au LSV. L'atmosphère familiale qui règne dans ce laboratoire associée à un réel souci d'excellence ont été un moteur pour moi. J'ai ainsi pu côtoyer un grand nombre de personnalités d'exception, que je ne pourrais pas toutes citer – à croire que le recrutement se fait autant sur les qualités humaines que sur les performances scientifiques. Je vais tout de même remercier Nico pour ses aides répétées en LaTeX. Ce document lui doit beaucoup ! Je remercie également Thomas Brihaye qui lors de son très sympathique passage à Cachan m'a initiée un peu aux habitudes belges. Aujourd'hui je ne suis pas déroutée lorsqu'on me parle de Louis, ou qu'un chauffeur me dit qu'il ne sait pas entrer dans ce garage ; je pense aussi à Louis lorsque j'hésite devant mes cinq poubelles de tri : où jeter ce plastique qui ne porte pas de numéro ? ? Je remercie également Benedikt pour les instants Kinder (avec la participation de Peter, qui poussait la tentation jusqu'à me les déposer sur mon bureau). Merci à Nathalie et Marie pour avoir partagé mes obsessions culinaires, et patiemment écouté mes angoisses et déboires de thésarde (Arnaud en a aussi pris sa part). Je remercie également ceux qui ont partagé mon bureau pendant ces années : Stéphanie (arriver avant elle le matin a été un challenge que j'ai rarement relevé), Vincent Bernat, Régis, Antoine, Arnaud et Najla. Un merci particulier à Vincent pour son aide technique, et à Antoine pour avoir écouté avec autant d'enthousiasme le récit quotidien des péripéties de Betsalel.

Je remercie enfin, et comme c'est la tradition, ma famille pour son soutien. Ma mère et ma belle-mère en particulier (dites « les grands-mères »), ont fait preuve d'un dévouement sans faille pendant la rédaction de ce manuscrit, et ont donné beaucoup de leur temps, de leur amour et de leur énergie à Betsalel, puis à Noa. Pour cela, je les remercie du fond du cœur.

Mes derniers remerciements vont à Vincent, pour sa vaillance, et pour m'avoir tant apporté ces dernières années.

Table des matières

Chapitre 1

Introduction

Vérification de programmes

Alors que les systèmes informatiques ont pris une place aussi prépondérante dans la vie moderne, les conséquences tant économiques qu'humaines de défaillances de ces applications sont devenues de plus en plus importantes. Il est donc crucial d'être capable de s'assurer de leur bon fonctionnement. La méthode de vérification la plus immédiate consiste en fournir un jeu de tests au programme à vérifier, et à confronter les résultats obtenus à ceux attendus. Cette technique permet de mettre en évidence certaines erreurs, mais présente de sérieux défauts ; tout d'abord, l'ensemble de tests effectués peut ne pas être exhaustif, c'est-à-dire qu'il peut rester des exécutions erronées qui n'ont pas été détectées. De plus, les objectifs à vérifier peuvent être donnés de façon imprécise et donner lieu à des erreurs d'interprétation. Il convient donc de développer des méthodes plus fiables et plus rigoureuses permettant de s'assurer du bon fonctionnement d'un programme. Démontrer des propriétés sur des programmes est un objectif essentiel de l'informatique moderne, mais n'est pourtant pas récent, et n'est pas uniquement lié aux enjeux actuels. Dès 1953, [Ric53] a montré qu'il est impossible de prouver automatiquement toute propriété non triviale sur un programme. Dès lors, la quête de programmes automatiquement certifiés comme étant corrects semble vouée à l'échec. Cependant, il est possible d'espérer trouver des solutions partielles permettant de vérifier les programmes. Les méthodes formelles font partie des approches proposées dans ce sens durant les dernières décennies. Parmi elles, on distingue notamment

- *la génération automatique de tests*, qui, se basant sur une description formelle des comportements souhaités du système, génère un ensemble exhaustif de tests à effectuer. Cependant il n'est pas toujours possible de fournir un tel ensemble de tests.
- *la démonstration automatique*, qui laisse l'ordinateur prouver automatiquement des théorèmes décrivant les propriétés du système, en se basant sur la description du programme, et sur un ensemble de règles de déduction et d'axiomes. Cette méthode n'est cependant pas totalement automatisable, et l'assistant de preuves a besoin d'être plus ou moins guidé par une aide humaine pendant le processus.
- *le Model-checking*, méthode qui part, elle, d'une représentation abstraite du programme, et d'une spécification formelle des comportements souhaités, et qui vérifie de façon exhaustive que tous les comportements du modèle satisfont cette spécification. Bien sûr, la recherche n'est exhaustive que sur l'abstraction du système considérée, et cette méthode peut nécessiter d'être utilisée itérativement, après des raffinements successifs.

Si la notion classique de programme est capturée par le modèle des machines de Turing (une fonction qui prend une donnée en entrée, effectue un calcul dessus, et produit une sortie), il existe de nombreux programmes qui n'ont pas ce type de comportements. Il s'agit de systèmes qui maintiennent une interaction permanente avec un environnement, en réagissant aux actions de ce dernier (l'environnement peut modéliser aussi bien un autre programme, qu'un composant physique en interaction avec le système, ou encore un utilisateur humain). On appellera de tels programmes (modélisant par exemple des systèmes d'exploitation) des *systèmes réactifs*, par opposition aux premiers que l'on appellera *programmes transformationnels*. Les spécifications sur de tels systèmes décrivent des séquences éventuellement infinies d'événements. Le formalisme logique le plus largement utilisé par la communauté scientifique pour la description des comportements de ce type de systèmes est celui des *logiques temporelles*, introduites dans le monde de la vérification par [Pnu77]. Les logiques temporelles se divisent en deux « familles », correspondant à deux représentations du temps. Dans la première on trouve les logiques de temps linéaire (dont la plus célèbre est LTL), qui supposent l'exécution du système fixée dès le départ. Les modalités utilisées concernent alors l'unique futur possible dans l'exécution sélectionnée. Dans la seconde sont regroupées les logiques de temps arborescent (CTL, CTL*,...). Dans ce cas, on estime qu'à chaque instant de l'exécution plusieurs futurs sont envisageables, et on permet donc d'exprimer des contraintes sur l'existence d'un prolongement possible de l'exécution courante. Le Model-checking décrit ci-dessus, avec des spécifications décrites par des formules de logique temporelle, s'est révélé particulièrement populaire pour la vérification de tels systèmes.

Synthèse et contrôle de systèmes

Un des reproches qui a cependant été fait à ces techniques de vérification est qu'elles interviennent uniquement à la fin du processus de développement, une fois qu'une quantité importante de ressources a été investie dans le programme. Il serait intéressant de pouvoir dériver automatiquement un programme, directement à partir de sa spécification. Lorsque le système interagit avec un environnement, le problème est essentiellement de savoir s'il est possible de déterminer, étant donné une description des comportements que l'on souhaite, s'il existe un programme dont toutes les exécutions satisfont la spécification et ce, *quel que soit la façon dont se comporte l'environnement*. Il s'agit de synthèse de systèmes *ouverts*. La question a été posée pour la première fois par Church [Chu63], et concernait la synthèse de circuits, et des spécifications données dans la logique monadique du second ordre sur les mots MSO. La réponse a été apportée par Büchi et Landweber [BL69], qui ont montré que le problème était décidable, en utilisant des techniques issues de la théorie des jeux. En effet, la synthèse de systèmes réactifs peut être vue comme un jeu entre l'environnement et le système, dans lequel la condition de gain est la spécification. Lorsqu'une partie satisfait la spécification, le système gagne, sinon c'est l'environnement. Ainsi, fournir une *stratégie gagnante* pour le système revient à donner une description du programme capable de satisfaire la spécification quel que soit le comportement de l'environnement au cours de l'exécution. De plus, si on peut construire une stratégie à mémoire finie, elle correspond à un programme à états finis pour le système. Quelques années plus tard, une autre démonstration, plus simple, a été apportée par [Rab72], basée sur la remarque que même si la spécification porte sur des exécutions vues comme des *mots*, la nécessité de considérer tous les comportements possibles de l'environnement induit un branchement qui impose d'utiliser des automates sur les *arbres* afin de résoudre le problème.

Le problème de l'existence d'un programme satisfaisant une spécification donnée sur toutes ses exécutions est appelé le problème de réalisabilité. Si la réponse est positive, produire un tel programme est le problème de synthèse.

Dans leurs formes modernes, les problèmes de réalisabilité et de synthèse ont été étudiés dans les années suivantes, sous diverses variantes. La synthèse de programmes transformationnels a été traitée par [MW80], puis celle de systèmes réactifs *clos* (i.e., qui n'interagissent pas avec un environnement incontrôlable, comme dans le cas de systèmes ouverts) devant satisfaire des spécifications données en logique temporelle a été résolue par [EC82, MW84]. La synthèse de systèmes réactifs ouverts (comme celle considérée par [Chu63, BL69]) a été à nouveau considérée par [ALW89, PR89a]. En particulier, les travaux de [PR89a] résolvent le problème de synthèse pour des spécifications de la logique temporelle LTL, et donnent un algorithme bien plus simple que dans le cas général de la logique monadique du second ordre. Le travail de cette thèse se place dans le cadre des systèmes réactifs ouverts.

À la fin des années 1980 a été également développée la théorie du contrôle des systèmes à événements discrets [RW89]. Les systèmes à événements discrets (DES) sont des systèmes de transition dans lesquels le changement d'état est déclenché par l'occurence d'un événement. Dans le problème de contrôle, on se donne un programme (modélisé par un DES) en interaction avec l'environnement, et une spécification, mais, contrairement au problème de la vérification que l'on a évoqué plus haut, le but n'est pas de s'assurer que le programme satisfait la spécification, mais de le forcer à la satisfaire, i.e., de restreindre ses comportements possibles. Ceci se fait par l'intermédiaire d'un contrôleur, modélisé également par un DES, qui va évoluer en parallèle avec le système à contrôler (on dit aussi à superviser), en le contraignant à rester dans l'ensemble des comportements autorisés par la spécification. Dans ce modèle, les événements du système à contrôler sont partitionnés entre les événements contrôlables, et les événements incontrôlables. Le contrôleur à synthétiser n'est bien sûr autorisé à restreindre que les actions contrôlables du système, et ne doit pas interdire toutes les actions contrôlables (dans ce dernier cas, le problème deviendrait trivial : l'ensemble des comportements possibles du système étant vide, il est toujours inclus dans l'ensemble des comportements acceptés par la spécification !). Le cadre de la théorie des jeux reste très utile pour ce problème (voir [AVW03], dans lequel ont été considérées des spécifications du μ-calcul). Des contrôleurs pour des spécifications données dans une logique de temps arborescent ont également été étudiés dans [MT02a], où les contrôleurs assurent la bisimilarité du système supervisé et de la spécification.

En fait, le problème de synthèse de Church et le problème de contrôle de Ramadge-Wonham sont extrêmement proches conceptuellement, et l'on peut voir le problème de synthèse comme un cas particulier du problème de contrôle : celui dans lequel le système à superviser est constitué d'un unique état, à partir duquel toutes les actions sont possibles.

Les travaux cités jusqu'à présent se concentraient sur des systèmes centralisés. Or, les applications réelles impliquent de plus en plus des systèmes *distribués*, c'est-à-dire constitués de plusieurs processus, pouvant communiquer entre eux de manière plus ou moins restreinte (cet ensemble de processus et leurs possibilités de communication étant décrits par une *architecture*). Étendre les résultats sur la synthèse et le contrôle de systèmes réactifs ouverts aux systèmes distribués constitue donc un objectif important. Ce problème est plus difficile, puisque dans ce cas, les différents processus formant le système ont chacun une vue partielle de l'état global, obtenue en rassemblant les informations reçues localement de l'environnement, et celles obtenues par communication avec les autres processus. Une première étape dans la résolution de ce problème est de considérer des systèmes centralisés, mais à information

partielle. Dans de tels systèmes, on considère que certaines informations de l'environnement sont cachées au contrôleur (ou simplement au processus du système dans le cas de la synthèse), qui doit cependant réussir à satisfaire la spécification. Ce problème a été abordé dans [KV97, KV99, AVW03], pour des spécifications en logique de temps arborescent. Pour les systèmes distribués se pose par ailleurs la question de la sémantique d'exécution : synchrone ou asynchrone. Dans les systèmes à exécution synchrone, une horloge globale règle l'avancement de tous les processus, et chaque processus, ainsi que l'environnement, effectue une action à chaque instant donné par l'horloge. Le problème a été abordé pour la première fois dans le cadre de la synthèse, dans [PR90] avec des spécifications LTL, puis étendu aux spécifications CTL* dans [KV01]. Contrairement au cas centralisé, le problème est indécidable en général dans les systèmes distribués synchrones, et très peu de sous-classes décidables ont pu être identifiées. Les systèmes à comportement asynchrone sont des systèmes dans lesquels chaque processus a sa vitesse d'exécution propre. Le problème de synthèse de systèmes asynchrones a tout d'abord été abordé dans le cas centralisé avec des spécifications de temps linéaire, où le processus et son environnement évoluent de façon asynchrone [PR89b, Var95]. Puis, le problème de contrôle dans le cas distribué asynchrone a été étudié dans [MT02b]. Ils obtiennent la décidabilité du problème sous des restrictions très fortes. Plus tard, les travaux de [GLZ04, MTY05] ont également abordé ce problème, en autorisant une mémoire plus puissante aux contrôleurs, ce qui permet d'élargir un peu ces résultats. Enfin, [FS06] ont traité le problème de synthèse de systèmes distribués pour des spécifications en logique de temps arborescent, avec un modèle de communication par variables partagées. Encore une fois, les résultats provenant de la théorie des jeux restent utiles pour résoudre les problèmes de synthèse et de contrôle dans le cas distribué. Les résultats de [PR90] tant de décidabilité que d'indécidabilité se basent sur des travaux sur les jeux multi-joueurs de [PR79]. Un formalisme de jeux multi-joueurs adapté à la synthèse et au contrôle de systèmes distribués a été proposé [MW03, BJ05]. Ces travaux proposent un cadre abstrait dans lequel s'encodent les problèmes de synthèse et de contrôle des systèmes distribués, tant synchrones qu'asynchrones, et donnent des conditions sur la structure des jeux permettant d'obtenir la décidabilité.

On peut enfin relever que le problème de synthèse de systèmes ouverts a également été étudié pour des spécifications dans des logiques *épistémiques* de temps linéaire, dans [vdMV98] pour les systèmes centralisés, et étendus aux systèmes distribués dans [vdMW05].

Contributions de la thèse

Dans cette thèse est abordé le problème de synthèse de systèmes distribués ouverts, dans les cas à la fois de comportement synchrone et asynchrone. Dans le cas de systèmes distribués, l'indécidabilité du problème de synthèse est très vite atteinte. On peut espérer obtenir des solutions positives, en restreignant la généralité du problème, et en particulier en limitant le pouvoir d'expression des spécifications. On défend dans ce travail l'idée que des spécifications raisonnables doivent être *externes*. Sont externes des spécifications ne décrivant que les interactions entre le système vu dans son ensemble et l'environnement, à l'exclusion des communications entre les processus du système eux-mêmes (ceci, en opposition avec des spécifications dites *totales*). En effet, d'un point de vue applicatif, la description des bons comportements d'un système distribué ne devrait concerner que les actions visibles, c'est-à-dire les sorties du système en fonction des entrées faites par l'environnement, le système en lui-même se comportant comme une boîte noire.

Le chapitre 2 commence par décrire les différents outils mathématiques utilisés dans ce document. Parmi les modèles utilisés, qui vont permettre essentiellement de décrire les comportements de nos systèmes, on trouve les mots, finis ou infinis, qui formalisent les exécutions des systèmes réactifs, vus comme des séquences d'actions du système ou de l'environnement, mais également les arbres. Comme on l'a mentionné, même lorsque l'on considère les comportements du systèmes comme des mots, les *programmes* que l'on cherche à considérer sont en fait des *arbres*, car ils doivent prendre en compte à chaque instant les différents comportements possibles de l'environnement. Par ailleurs, on peut être amené à considérer les exécutions comme des arbres également, lorsque le formalisme de spécification utilisé est une logique de temps arborescent. Cette modélisation des exécutions par des mots, ou des arbres convient bien aux systèmes centralisés, ou aux systèmes distribués synchrones. En effet, lors d'une exécution synchrone, chaque composant du système évolue aux mêmes instants, et il suffit de considérer un alphabet constitué de n-uplets d'actions pour décrire par un mot sur cet alphabet le comportement de ce système distribué. Lorsque les composants du système évoluent de manière asynchrone par contre, les actions effectuées par les différents sites sont concurrentes. Considérer des actions globales comme on le fait dans le cas de systèmes distribués synchrones induirait une synchronisation des événements inadaptée. Une option est alors de voir les exécutions d'un système distribué comme un mot sur l'alphabet des actions locales, dans lequel les actions intervenant de façon concurrente sont entrelacées. Cela revient à remplacer la notion de *concurrence* par celle de *non-déterminisme* : l'ordre d'exécution entre deux actions concurrentes est résolu par un choix non-déterministe. Un formalisme plus satisfaisant a été développé dans les années 70 : celui des traces de Mazurkiewicz, qui se propose de conserver la notion de concurrence de façon explicite, en représentant une exécution par une *classe d'équivalence de mots*. Dans cette thèse, les spécifications seront généralement exprimées par des formules de logique temporelle. On donne donc la syntaxe et la sémantique de ces dernières, en particulier de LTL, CTL et CTL* qui sont celles qui seront le plus utilisées dans ce document. On présente enfin la notion d'automate fini, qui sera également largement utilisée : un automate fini (de mots ou d'arbres) permet de définir un ensemble (de mots ou d'arbres), appelé langage. En particulier, on s'intéressera à des automates pour lesquels la question de savoir s'ils acceptent un langage non-vide est décidable. Ils nous permettront d'une part d'exprimer des spécifications, en décrivant de façon finie l'ensemble des exécutions souhaitées du système, et d'autre part de résoudre le problème de synthèse : en testant si un automate d'arbres est vide ou non, on pourra décider de l'existence d'un programme satisfaisant la spécification.

Dans la seconde partie du chapitre 2, on présente un cadre formel dans lequel on exprime le problème de synthèse et de contrôle de systèmes distribués. En effet, de nombreux travaux ont été effectués sur le sujet, utilisant chacun des hypothèses particulières pour le problème. On traduit un certain nombre de ces modèles dans le formalisme que l'on a défini, ceci afin d'avoir une vision cohérente (et plus complète) de l'état de l'art présenté succinctement dans cette introduction, et de permettre de mieux comprendre les différences et les progrès apportés par les différentes publications sur le sujet. Ce modèle se veut suffisamment concret pour rester proche des intuitions que l'on peut avoir sur la définition du problème. On se base sur le modèle des automates asynchrones de Zielonka [Zie87], que l'on a un peu étendu afin d'exprimer à la fois le problème de contrôle des systèmes distribués synchrones et asynchrones. En particulier, on autorise nos automates à exécuter des n-uplets d'actions (ce qui n'est pas prévu dans le modèle originel), afin de représenter les comportements des systèmes distribués synchrones. On définit ensuite dans ce modèle les notions de spécifications, de programmes (ou stratégies),

et enfin le problème de synthèse de contrôleurs (le vocabulaire utilisé emprunte souvent au vocabulaire issu de la théorie des jeux, en raison des similitudes que l'on a déjà relevées entre les deux domaines). Puis on montre comment décrire dans ce modèle les travaux publiés sur le sujet dans le cas des systèmes à exécution synchrone, et celui des systèmes à exécution asynchrone.

Le chapitre 3 présente les résultats obtenus dans le cadre de systèmes synchrones. Pour ce problème particulier, [PR90] ont exhibé une architecture pour laquelle le problème est indécidable pour des spécifications LTL. Depuis, les travaux sur le sujet ont principalement eu pour objet de définir des architectures particulières pour lesquelles le problème est décidable. Un résultat plus satisfaisant serait d'obtenir un *critère de décidabilité* portant en particulier sur l'architecture de communication. En effet, la façon dont les informations de l'environnement sont distribuées parmi les processus semble cruciale. Un tel critère de décidabilité a été défini pour des spécifications *totales* dans [FS05] : ils établissent que le problème est décidable pour les architectures dans lesquelles les processus ont une connaissance « hiérarchique » de l'état global du système, i.e., une architecture dans laquelle on peut ordonner les processus en fonction de leur connaissance de l'état global du système. Au contraire, pour toute architecture dans laquelle il y a deux processus ayant une connaissance incomparable de l'état global, le problème est indécidable. Dans ce chapitre, on se place par contre dans le cas particulier des spécifications de type *externe*, qui ne contraignent pas les communications entre processus. En effet, en plus d'être plus naturelles d'un point de vue pratique, comme on l'a mentionné, les spécifications externes permettent d'exclure des spécifications trop puissantes, rendant le problème rapidement indécidable. En effet, une spécification contraignant les interactions entre les processus du système entre eux peut facilement interdire toute communication entre deux processus particuliers, ce qui revient à se détacher de l'architecture même du système, en les empêchant par exemple d'avoir toute la connaissance possible de l'état global. Par contre, avec des spécifications externes, l'obtention d'un critère de décidabilité est plus difficile, puisque, contrairement au cas de spécifications totales, la preuve d'indécidabilité de [PR90] ne peut plus s'appliquer dans un aussi grand nombre de cas.

On commence par définir un modèle plus agréable pour le problème de synthèse de systèmes distribués synchrones, rendant les démonstrations plus légères, et dans lequel on autorise les processus à fonctionner avec des délais arbitraires, ce qui constitue une généralisation des modèles précédemment étudiés, dans lesquels tous les processus fonctionnaient sans délai, ou bien avaient tous un délai d'une unité de temps. On montre que ce modèle est équivalent au modèle du chapitre 2 pour le problème de synthèse. Puis, on montre que, dans le cas de spécifications externes, l'indécidabilité mise en évidence dans [PR90] se généralise aux *architectures à information incomparable* : une architecture est à information incomparable s'il existe deux processus *avec sortie* recevant des ensembles d'informations incomparables de l'environnement (éventuellement de façon indirecte). Dans certains cas, cette condition peut être vue comme une condition d'accessibilité dans le graphe de communications du système : existe-t-il deux processus avec sortie, et deux entrées distinctes de l'environnement, tels qu'il existe un chemin de communication d'une entrée vers un processus, mais pas vers l'autre, et de la seconde entrée vers le second processus mais pas vers le premier ? La différence avec le critère établi pour le cas de spécifications totales est qu'on restreint cette condition sur la connaissance de l'état global aux processus en sortie, tandis que dans le premier cas, tous les processus sont concernés. C'est donc un nombre bien plus restreint d'architectures qui sont concernées par cette condition d'indécidabilité. On définit ensuite une sous-classe des

architectures, les architectures uniformément bien connectées, pour lesquelles on montre que cette condition devient un critère nécessaire et suffisant d'indécidabilité. Une architecture est uniformément bien connectée si chaque processus en sortie peut connaître à chaque instant toutes les informations en provenance de l'environnement qui lui sont potentiellement accessibles : en effet un phénomène de goulot d'étranglement dans l'architecture peut empêcher un processus de connaître toute l'information disponible à chaque instant, car le processus devant lui transmettre les valeurs n'a pas suffisamment de bande passante et doit échantillonner l'information à transmettre par exemple. Tester si un système est uniformément bien connecté revient à vérifier l'existence d'un routage possible de l'information au sein de l'architecture. On démontre dans ce chapitre la décidabilité de ce problème, ainsi que sa complexité, puis on montre que le problème de synthèse pour les systèmes distribués est décidable pour une architecture uniformément bien connectée si et seulement si elle n'est pas à information incomparable. Ceci permet de mettre en évidence des architectures pour lesquelles il devient possible de décider le problème de synthèse, alors que cela ne l'était pas dans le cas de spécifications totales. Enfin, on montre que ce critère ne s'applique plus pour les architectures dans lesquelles les processus en sortie n'ont pas accès à toute l'information possible *à chaque instant*. On montre qu'il suffit à un processus de perdre *un seul bit d'information* pour retrouver l'indécidabilité du problème.

Une partie des résultats de ce chapitre a été publiée dans [GSZ06] (une version longue des résultats précédents a été publiée dans [GSZ09]).

On étudie dans le chapitre 4 le problème de synthèse de systèmes distribués en sémantique asynchrone. Dans la littérature, ce problème a été traité sous divers angles : comme dans le cas synchrone, le type de spécifications autorisées est un paramètre du problème, mais aussi le type de communications autorisées entre les processus, et le type de mémoire laissée au contrôleur (ces différents aspects seront abordés plus en détail dans le chapitre 2). On introduit ici un nouveau paradigme de communication, que l'on appelle communication par *signaux*. Ce type de communication est une synchronisation d'actions entre deux processus, mais initiée par un seul, le processus émetteur. Le processus récepteur ne peut refuser de recevoir un signal. Ce type de communications est plus puissant que les variables partagées (mécanisme dans lequel un processus peut lire une variable modifiée par un autre processus, mais dans le cadre asynchrone, il n'a aucune garantie de lire la « bonne » valeur de la variable), et plus réaliste que la synchronisation par rendez-vous, dans laquelle les deux processus doivent se mettre d'accord pour effectuer une action commune. Comme dans le chapitre 3, on ne considère que des spécifications externes, dont les modèles seront des ordres partiels représentant la partie *visible* des exécutions du système (les communications internes entre les processus étant considérées invisibles par la spécification). Le choix de spécifications externes étant motivé par la volonté de ne pas restreindre les communications entre les processus, on se restreint à des spécifications ayant de bonnes propriétés de clôture, assurant qu'elles autorisent bien les processus à communiquer sans restriction, dans la mesure où l'exécution visible reste correcte. Enfin, dans les sémantiques d'exécution asynchrone, il existe des exécutions que l'on peut considérer comme dégénérées. En effet, contrairement au cas synchrone où chaque processus peut et doit jouer à chaque instant de l'exécution, dans les exécutions asynchrones, on suppose généralement l'existence d'un ordonnanceur activant à chaque instant certains des processus qui le souhaitent. Si on ne pose aucune condition sur cet ordonnanceur, on obtient parmi les exécutions du système des cas où un processus qui le souhaitait n'a jamais pu effectuer d'action, car saturé d'événements de la part de l'environnement par exemple. Une

façon d'éliminer ces cas pathologiques et de considérer que l'on a un ordonnanceur équitable. On ne confronte alors à la spécification que les exécutions dites équitables du système. La synthèse équitable avait été abordée dans le cas centralisé [Var95, AM94]. On introduit dans ce chapitre une notion d'équité pour la synthèse de systèmes distribués asynchrones. Sous ces hypothèses, on montre que le problème est décidable pour toute la classe d'architectures fortement connexes (i.e., dans lesquelles chaque processus peut envoyer un signal à chacun des autres, éventuellement en le faisant transmettre par des processus intermédiaires). Ce résultat constitue un progrès par rapport au cas synchrone en particulier, dans lequel les architectures fortement connexes sont en général indécidables.

Des résultats préliminaires à ceux décrits dans ce chapitre ont été publiés dans [CGS09].

Chapitre 2

Formalismes

Sommaire

Ce chapitre fixe les formalismes qui vont être utilisés dans ce manuscrit. La première section reprend des définitions classiques qui seront utiles pour exposer les résultats de cette thèse. La seconde section est une présentation personnelle du problème de synthèse de contrôleurs distribués : on y introduit un cadre uniforme dans lequel intégrer les différents travaux effectués sur le sujet. Cette tentative peut se rapprocher du travail de [MW03], mais l'objectif est différent : on cherche ici non pas un formalisme abstrait permettant de mettre en valeur ou démontrer des résultats généraux, mais plutôt à définir un modèle plus concret rendant compte des différents choix de modélisation faits dans les travaux de la littérature (même si cela rend certains résultats plus difficiles à démontrer). On espère ainsi donner une vision uniforme des différents formalismes choisis, et on se donne les outils pour comparer les résultats déjà obtenus à ceux que l'on développera dans les chapitres suivants.

1 Définitions utiles

1.1 Modèles

1.1.1 Mots

Un *alphabet* Σ est un ensemble fini de symboles. Une séquence d'éléments de Σ est appelée un *mot*. Si $\sigma = s_1 s_2 \cdots s_n$ est un mot fini, n est la *longueur de* σ noté $|\sigma| = n$. Si $\sigma = s_1 s_2 \cdots$ est un mot infini, $|\sigma| = \omega$. Le mot de longueur 0 est appelé *mot vide* et est dénoté ε. On note Σ^* l'ensemble des mots finis de Σ, Σ^+ l'ensemble des mots finis non vides et Σ^ω l'ensemble des mots infinis. On note $\Sigma^\infty = \Sigma^* \uplus \Sigma^\omega$ l'ensemble des mots finis ou infinis de Σ.

Pour tous mots $\sigma = s_1 \cdots s_n$ et $\sigma' = s'_1 \cdots s'_m$, on définit la *concaténation* $\sigma \cdot \sigma' = s_1 \cdots s_n \cdot s'_1 \cdots s'_m$. L'ensemble $(\Sigma, \cdot, \varepsilon)$ forme un monoïde, ou *monoïde libre généré par* Σ.

Soient Σ et Σ' deux alphabets tels que $\Sigma' \subseteq \Sigma$. Soit $\sigma \in \Sigma^*$ un mot fini de Σ. On définit la projection de σ sur Σ', noté $\pi_{\Sigma'}(\sigma)$ par

$$\pi_{\Sigma'}(\sigma) = \begin{cases} \varepsilon & \text{si } \sigma = \varepsilon \\ \pi_{\Sigma'}(u) & \text{si } \sigma = ua \text{ et } a \notin \Sigma' \\ \pi_{\Sigma'}(u)a & \text{si } \sigma = ua \text{ et } a \in \Sigma' \end{cases}$$

Soit $\sigma \in \Sigma^\infty$. On définit l'ensemble des *préfixes* de σ comme étant $\mathrm{Pref}(\sigma) = \{u \mid \exists v, uv = \sigma\}$. La relation *préfixe* est une relation binaire $\sqsubseteq$ sur Σ^∞ telle que $\sigma' \sqsubseteq \sigma$ si et seulement si $\sigma' \in \mathrm{Pref}(\sigma)$. La relation $\sqsubseteq$ est une relation d'ordre.

Pour un mot $\sigma \in \Sigma^\omega$, on définit sa projection sur Σ' par

$$\pi_{\Sigma'}(\sigma) = \bigsqcup_{\substack{\sigma' \in \Sigma^*, \\ \sigma' \sqsubseteq \sigma}} \pi_{\Sigma'}(\sigma').$$

Le préfixe de longueur $i \leq |\sigma|$ d'un mot σ est noté $\sigma[i]$. Par convention, on considère que si $i \leq 0$, $\sigma[i] = \varepsilon$. Pour un mot $\sigma = s_1 s_2 \cdots \in \Sigma^\infty$, pour tout i, j entiers, on note $\sigma[i \dots j]$ le facteur formé des lettres $s_i \cdots s_j$, qui est vide si $i > j$. On utilise aussi cette notation si $i \leq 0$ ou $j \leq 0$: si $j \leq 0$, alors $\sigma[i \cdots j] = \varepsilon$, et si $i \leq 0 < j$ alors $\sigma[i \cdots j] = \sigma[1 \cdots j]$.

Soit $\sigma \in \Sigma^\omega$. On définit l'ensemble des lettres apparaissant infiniment souvent dans σ, noté $inf(\sigma) = \{a \in \Sigma \mid |\pi_{\{a\}}(\sigma)| = \omega\}$.

Un sous-ensemble $\mathcal{L}$ de Σ^∞ est appelé un *langage*. Si $\mathcal{L}$ et $\mathcal{K}$ sont deux langages de mots finis, la concaténation de $\mathcal{L}$ et $\mathcal{K}$, notée $\mathcal{L} \cdot \mathcal{K}$, est l'ensemble des mots uv formés par concaténation de $u \in \mathcal{L}$ et $v \in \mathcal{K}$. Pour n entier positif, on note $\mathcal{L}^n$ la *puissance n-ième* de $\mathcal{L}$, définie par

$$\mathcal{L}^0 = \{\varepsilon\}$$
$$\mathcal{L}^n = \mathcal{L}^{n-1} \cdot \mathcal{L} \text{ si n>0.}$$

Pour $N > 0$, on note $\mathcal{L}^{<N} = \bigcup_{n<N} \mathcal{L}^n$ le langage formé par union de toutes les puissances de $\mathcal{L}$ plus petites que N.

1.1.2 Arbres

Étant donnés un ensemble fini X et un ensemble Y, un X-arbre Y-étiqueté (appelé aussi (X, Y)-arbre) est une fonction $t : X^* \to Y$ dont le domaine de définition est clos par préfixe, dans laquelle les éléments de X sont appelées *directions* et les éléments de Y sont appelés *étiquettes*. Lorsque la fonction est totale, on dit que l'arbre est *complet*. Un mot $\sigma \in X^*$ définit un *nœud* de t et $t(\sigma)$ est son *étiquette*. Le mot vide ε est la *racine* de l'arbre. Un mot $\sigma \in X^\infty$ est une *branche*. Une branche σ est *maximale* si $\sigma \in \mathrm{dom}(t)$ et, pour tout $s \in X$, $\sigma \cdot s \notin \mathrm{dom}(t)$.

On dit que t' est un *sous-arbre de* t si $dom(t') \subseteq dom(t)$.

1.1.3 Ordres partiels

Soit $t = (V, \leq)$ un ensemble partiellement ordonné, et $H \subseteq V$. On note $\downarrow_t H = \{x \in V \mid \exists h \in H, x \leq h\}$ le passé de H dans t. Lorsqu'il n'y a pas d'ambiguïté sur t on écrira simplement $\downarrow H$. On notera également $\downarrow x = \downarrow\{x\}$ et $\Downarrow x = \downarrow x \setminus \{x\}$.

On notera la relation successeur par $\lessdot = < \setminus <^2$.

De plus, lorsque x et y sont deux événements de t non ordonnés, i.e., tels que $x \not\leq y$ et $y \not\leq x$, on notera $x \parallel_t y$.

Définition 2.1 (Ordre partiel étiqueté)**.** *Un* ordre partiel étiqueté par Σ *est un triplet* $t = (V, \leq, \lambda)$ *où* $(V, \leq)$ *est un ordre partiel, et* $\lambda : V \to \Sigma$ *une fonction d'étiquetage des événements par des actions. Deux ordres partiels étiquetés* $t_1 = (V_1, \leq_1, \lambda_1)$ *et* $t_2 = (V_2, \leq_2, \lambda_2)$ *sont* isomorphes *s'il existe une bijection* $\varphi : V_1 \to V_2$

- *préservant l'ordre :* $(x \leq_1 y) \Leftrightarrow (\varphi(x) \leq_2 \varphi(y))$
- *et préservant l'étiquetage :* $\lambda_1(x) = \lambda_2(\varphi(x))$

On dit qu'un ordre partiel étiqueté par Σ, $t' = (V', \leq', \lambda')$, est une *extension* de t s'il existe une bijection $\varphi : V \to V'$ telle que, pour tout $x, y \in V$, $x \leq y$ implique que $\varphi(x) \leq' \varphi(y)$ et $\lambda(x) = \lambda'(\varphi(x))$. Si $\leq'$ est un ordre total, on dit que t' est une *extension linéaire* (ou *linéarisation*) de t.

Pour $t = (V, \leq, \lambda)$, ordre partiel étiqueté par Σ, et pour tout $\Sigma' \subseteq \Sigma$, on note $\pi_{\Sigma'}(t) = (V', \leq', \lambda')$ la *projection de* t *sur* Σ' définie par

$$\begin{aligned} V' &= \lambda^{-1}(\Sigma') \\ \leq' &= \leq \cap (V' \times V') \\ \lambda' &= \lambda_{|\Sigma'} \end{aligned}$$

1.1.4 Traces de Mazurkiewicz

La théorie des traces de Mazurkiewicz [Maz77, Maz86] (voir aussi [RD95]) a été développée dans le but de fournir un cadre formel adapté précisément à la représentation et l'analyse des exécutions de systèmes distribués. En effet, dans un système distribué, où deux actions peuvent avoir lieu de façon indépendante sur deux sites distincts, considérer les exécutions comme des mots implique de choisir un ordre entre ces deux actions, ce qui revient à supposer que deux actions *concurrentes* vont être exécutées dans un ordre choisi de façon *non-déterministe*. Au contraire, les traces de Mazurkiewicz ne retiennent un ordre entre deux actions que si celles-ci sont liées par une relation de *causalité*, notion qui est traduite dans le modèle par la notion de *dépendance*.

Définition 2.2 (Alphabet de dépendance)**.** *Un* alphabet de dépendance *est une paire* (Σ, D) *où* Σ *est un ensemble fini d'actions et* $D \subseteq \Sigma \times \Sigma$ *est une relation réflexive et symétrique appelée* relation de dépendance*. Son complémentaire* $I = \Sigma \times \Sigma \setminus D$ *est appelé* relation d'indépendance*.*

Définition 2.3 (Trace de Mazurkiewicz)**.** *Une* trace de Mazurkiewicz sur (Σ, D) *est, à isomorphisme près, un ordre partiel étiqueté par* Σ*,* $t = (V, \leq, \lambda)$*, vérifiant :*

1. *pour tout* $x \in V$*, l'ensemble* $\downarrow x$ *est fini,*
2. *pour tout* $x, y \in V$*,* $x \lessdot y \Rightarrow \lambda(x)\ D\ \lambda(y)$*,*

3. *pour tout* $x, y \in V$, $\lambda(x)\ D\ \lambda(y) \Rightarrow x \leq y$ *ou* $y \leq x$.

Lorsque l'ensemble V est fini, on dit que la trace t est finie, et lorsqu'il est infini, qu'elle est infinie. On note $\mathbb{R}(\Sigma, D)$ l'ensemble des traces finies et infinies sur (Σ, D) et $\mathbb{M}(\Sigma, D)$ l'ensemble des traces finies sur (Σ, D) (on pourra se référer utilement à [GP95] pour une présentation des traces infinies).

Définition 2.4 (Trace préfixe)**.** *Une trace* $t' = (V', \leq', \lambda')$ *est* préfixe *de la trace* $t = (V, \leq, \lambda)$ *si*

- $V' \subseteq V$
- *pour tout* $x, y \in V'$, $x \leq' y \Leftrightarrow x \leq y$
- *pour tout* $x \in V'$, $\lambda'(x) = \lambda(x)$
- V' *est clos par le bas :* $\downarrow_t V' = V'$.

Il existe une application surjective des mots dans les traces $[\ ] : \Sigma^\infty \to \mathbb{R}(\Sigma, D)$ définie comme suit. Soit u un mot de Σ^∞ et soit $V = \{i \in \mathbb{N} \mid 0 \leq i < |u|\}$. Considérons alors l'ordre total $t_u = (V, \leq)$. Le mot u définit naturellement un fonction d'étiquetage $\lambda_u : V \to \Sigma$: si $u = a_0 a_1 \cdots \in \Sigma^\infty$, on pose $\lambda_u(i) = a_i$ et on peut identifier u à l'ordre partiel étiqueté $(V, \leq, \lambda_u)$. Alors $[u] = (V, \sqsubseteq, \lambda_u)$ avec $\sqsubseteq = R^*$ et $R \subseteq V \times V$ défini par $x\ R\ y$ si et seulement si $x \leq y$ et $(\lambda(x)\ D\ \lambda(y))$.

En fait, tout mot u de Σ^∞ est la linéarisation d'une unique trace de $\mathbb{R}(\Sigma, D)$, et toute trace de $\mathbb{R}(\Sigma, D)$ admet au moins une linéarisation dans Σ^∞.

On remarque par ailleurs que $[\Sigma^*] = \mathbb{M}(\Sigma, D)$.

Définition 2.5 (Concaténation de traces)**.** *Soient* $t_1 = (V_1, \leq_1, \lambda_1) \in \mathbb{M}(\Sigma, D)$ *et* $t_2 = (V_2, \leq_2, \lambda_2) \in \mathbb{M}(\Sigma, D)$ *deux traces finies. On note* $t_1 \cdot t_2 = (V, \leq, \lambda) \in \mathbb{M}(\Sigma, D)$ *l'unique trace de Mazurkiewicz telle que*

- $V = V_1 \uplus V_2$,
- $\lambda_{|V_1} = \lambda_1$,
- $\lambda_{|V_2} = \lambda_2$,
- $\leq \cap (V_1 \times V_1) = \leq_1$,
- $\leq \cap (V_2 \times V_2) = \leq_2$,
- *pour tout* $x_1 \in V_1$, *pour tout* $x_2 \in V_2$, $x_1 \leq x_2$ *si et seulement si* $\lambda(x_1)\ D\ \lambda(x_2)$,
- *pour tout* $x_1 \in V_1$, *pour tout* $x_2 \in V_2$, $x_2 \not\leq x_1$.

Pour $t \in \mathbb{M}(\Sigma, D)$ et $a \in \Sigma$, on note $t \cdot a \in \mathbb{M}(\Sigma, D)$ la trace $t \cdot t_a$ avec $t_a = (\{x\}, =, \lambda)$ où $\lambda(x) = a$.

1.2 Logiques temporelles

Les logiques temporelles sont des logiques modales permettant de spécifier des relations temporelles entre des événements (au cours d'une exécution d'un système typiquement). Leur utilisation pour raisonner sur le comportement des programmes est due à [Pnu77]. On distingue les logiques temporelles *linéraires* des logiques temporelles *arborescentes*.

1.2.1 Logique temporelle linéaire : LTL

La logique temporelle LTL (*Linear Temporal Logic*) a été introduite [Pnu77, Kam68, GPSS80] pour raisonner sur *une exécution particulière* du système. Sa syntaxe est la sui-

vante :

$$\varphi ::= \bot \mid \top \mid p \mid \neg\varphi \mid \varphi \vee \varphi \mid \mathsf{X}\,\varphi \mid \varphi\,\mathsf{U}\,\varphi$$

où p est une proposition atomique d'un ensemble dénombrable AP.

Les opérateurs temporels, X (*next*) et U (*until*), permettent respectivement d'exprimer qu'une formule est vraie à l'instant suivant de l'exécution, et qu'une certaine formule est vérifiée à tout instant jusqu'à ce qu'une autre formule soit vraie.

Formellement, les modèles de LTL sont des exécutions vues comme des mots finis ou infinis de 2^{AP}. Un modèle de LTL est donc une séquence de taille $|u|$ assimilée à l'application $u : \{i \in \mathbb{N} \mid 0 \leq i < |u|\} \to 2^{AP}$. De façon équivalente, un modèle peut également être vu comme un mot $u \in (2^{AP})^\omega$, avec $u = u_0 u_1 \cdots$, l'application associant alors à $u(i)$ la lettre u_i, ou comme un ordre total étiqueté par 2^{AP}. Étant donnés AP un ensemble de propositions atomiques, u un modèle, $0 \leq i < |u|$ une position et φ une formule de LTL, on définit par récurrence la relation de satisfaction $\models$ de la façon suivante :

- $u, i \models \top$
- $u, i \not\models \bot$
- $u, i \models p$ si et seulement si $p \in u(i)$
- $u, i \models \neg\varphi$ si et seulement si $u, i \not\models \varphi$
- $u, i \models \varphi \vee \psi$ si et seulement si $u, i \models \varphi$ ou $u, i \models \psi$
- $u, i \models \mathsf{X}\,\varphi$ si et seulement si $i + 1 < |u|$ et $u, i + 1 \models \varphi$
- $u, i \models \varphi\,\mathsf{U}\,\psi$ si et seulement si il existe $j < |u|$ tel que $i \leq j$ et $u, j \models \psi$ et, pour tout $i \leq k < j$, $u, k \models \varphi$.

On peut définir les connecteurs supplémentaires classiques $\varphi \wedge \psi \stackrel{\text{def}}{=} \neg(\neg\varphi \vee \neg\psi)$ et $\varphi \to \psi \stackrel{\text{def}}{=} \neg\varphi \vee \psi$, ainsi que les modalités temporelles $\mathsf{F}\,\varphi \stackrel{\text{def}}{=} (\top\,\mathsf{U}\,\varphi)$ signifiant que la propriété φ est vérifiée à un moment ultérieur de l'exécution, $\mathsf{G}\,\varphi \stackrel{\text{def}}{=} \neg(\mathsf{F}\,\neg\varphi)$ signifiant que la propriété φ est vraie à tous les moments ultérieurs de l'exécution, et $\varphi\,\mathsf{R}\,\psi \stackrel{\text{def}}{=} \neg(\neg\varphi\,\mathsf{U}\,\neg\psi)$ signifiant que l'obligation de vérifier ψ cesse lorsque φ est vérifiée : ψ est vraie jusqu'à ce que φ soit vraie (si φ est vérifiée un jour). Un autre opérateur utile est le *weak until* $\varphi\,\mathsf{W}\,\psi \stackrel{\text{def}}{=} \mathsf{G}\,\psi \vee (\varphi\,\mathsf{U}\,\psi)$, signifiant que la propriété φ est vérifiée tant que la propriété ψ n'est pas vérifiée, tout en autorisant la possibilité que ψ ne soit jamais vérifiée.

On dit qu'une séquence u satisfait la formule φ si $u, 0 \models \varphi$ et on notera généralement $u \models \varphi$.

On peut également introduire des opérateurs faisant référence au passé [Pri67]. L'opérateur $\mathsf{Y}\,\varphi$ se lit « la formule φ est vraie à l'instant précédent » et l'opérateur $\varphi\,\mathsf{S}\,\psi$ se lit « ψ a été vraie et depuis, φ est vraie ». Formellement leur sémantique est donnée par

- $u, i \models \mathsf{Y}\,\varphi$ si et seulement si $i > 0$ et $u, i - 1 \models \varphi$,
- $u, i \models \varphi\,\mathsf{S}\,\psi$ si et seulement si il existe $0 \leq j \leq i$ tel que $u, j \models \psi$ et pour tout $j < k \leq i$, $u, k \models \varphi$.

Comme dans le cas « pur futur », on peut définir la modalité $\mathsf{H}\,\varphi \stackrel{\text{def}}{=} (\top\,\mathsf{S}\,\varphi)$. Il a été montré dans [GPSS80, Gab87] que les modalités passées n'ajoutent pas d'expressivité lorsqu'on interprète les formules sur les modèles depuis l'origine. Par la suite, sauf mention contraire, on ne considérera la logique LTL que dans sa version ne contenant pas les modalités passées.

1.2.2 Logiques temporelles arborescentes : CTL et CTL*

Les logiques temporelles arborescentes supposent une vision *branchante* du temps. À un instant donné, plusieurs futurs sont possibles, tandis que dans les logiques temporelles linéaires, on considère l'exécution fixée dès le départ. Ceci implique que des propriétés comme « Depuis tout état où p est vérifiée, il est possible d'atteindre un état où q est vérifiée » ne peuvent être exprimées en LTL. Les logiques arborescentes ont donc été introduites pour pallier ce manque. Une logique temporelle arborescente est une logique capable de quantifier sur les *chemins*. Des exemples standards de telles logiques sont CTL (*Computation Tree Logic*) [CE81], CTL* [EH83] et le μ-calcul modal [Koz83].

La syntaxe de CTL* est définie par la grammaire suivante :

$$\Phi ::= \top \mid \bot \mid p \mid \neg\Phi \mid \Phi \vee \Phi \mid \mathsf{E}\,\psi$$
$$\psi ::= \Phi \mid \mathsf{X}\,\psi \mid \psi\,\mathsf{U}\,\psi$$

où p est une proposition atomique d'un ensemble dénombrable AP. Les formules Φ sont des formules d'*état*, et les formules ψ sont des formules de *chemin*. L'opérateur E (à rapprocher de l'opérateur existentiel $\exists$ de la logique du premier ordre) permet d'exprimer qu'à partir de l'état courant, il existe une exécution vérifiant une formule.

Les modèles de CTL* sont des arbres pour lesquels on associe à chaque nœud un ensemble de propositions atomiques de AP.

On note pour $n \in \mathbb{N}$, $\mathbb{N}_n = \{0, \ldots, n\}$ et $\mathbb{N}_\omega = \mathbb{N}$. Formellement, étant donnés $t : X^* \to 2^{AP}$ un arbre, $\sigma \in X^\infty$ une branche maximale de t, $i \in \mathbb{N}_{|\sigma|}$ une position, et Φ une formule de CTL* on définit par récurrence la relation de satisfaction $\models$ de la façon suivante :

- $t, \sigma, i \models \top$
- $t, \sigma, i \not\models \bot$
- $t, \sigma, i \models p$ si et seulement si $p \in t(\sigma[i]))$
- $t, \sigma, i \models \neg\Phi$ si et seulement si $t, \sigma, i \not\models \Phi$
- $t, \sigma, i \models \Phi_1 \vee \Phi_2$ si et seulement si $t, \sigma, i \models \Phi_1$ ou $t, \sigma, i \models \Phi_2$
- $t, \sigma, i \models \mathsf{E}\,\psi$ si et seulement si il existe $\sigma' \in X^\infty$ telle que $\sigma[i] \cdot \sigma'$ est une branche maximale de t et $t, \sigma[i] \cdot \sigma', i \models \psi$
- $t, \sigma, i \models \mathsf{X}\,\psi$ si et seulement si $i + 1 \in \mathbb{N}_{|\sigma|}$ et $t, \sigma, i + 1 \models \mathsf{X}\,\psi$
- $t, \sigma, i \models \psi_1\,\mathsf{U}\,\psi_2$ si et seulement si il existe $j \in \mathbb{N}_{|\sigma|}$ tel que $i \leq j$ et $t, \sigma, j \models \psi_2$ et pour tout $i \leq k < j$, $t, \sigma, k \models \psi_1$

On peut comme précédemment définir les connecteurs logiques $\wedge$ et $\rightarrow$ ainsi que les modalités temporelles F, G et R. On peut également définir l'opérateur de chemin dual de E par $\mathsf{A}\,\psi \stackrel{\text{def}}{=} \neg\,\mathsf{E}\,\neg\psi$, signifiant intuitivement que la propriété ψ est vérifiée sur tous les chemins partant du nœud courant.

On dit qu'un arbre t satisfait une formule *d'état* Φ de CTL* (noté $t \models \Phi$) si $t, \sigma, 0 \models \Phi$ avec σ branche maximale quelconque de t.

Il est clair que LTL est un fragment de CTL*. Un autre fragment bien connu de CTL* est la logique CTL définie par

$$\Phi ::= \top \mid \bot \mid p \mid \neg\Phi \mid \Phi \vee \Phi \mid \mathsf{EX}\,\Phi \mid \mathsf{E}\,\Phi\,\mathsf{U}\,\Phi \mid \mathsf{EG}\,\Phi$$

Les formules de CTL ne s'interprètent donc que sur les états, et non sur les chemins.

En fait, il se trouve que les logiques LTL et CTL ont des pouvoirs d'expression incomparables, et la logique CTL* et est plus expressive que LTL et CTL. Le μ-calcul que nous ne définirons pas ici est plus expressif que CTL*.

1.3 Automates finis

1.3.1 Automates de mots

Définition 2.6. *Un* automate de mots *est un quintuplet* $\mathfrak{A} = (\Sigma, Q, \delta, q_0, \Omega)$ *où*
- Σ *est un alphabet fini,*
- Q *est un ensemble fini d'*états *(que l'on suppose non vide),*
- q_0 *est l'*état initial*,*
- $\delta : Q \times \Sigma \to 2^Q$ *est la* fonction de transition*, et attribue à un état et une lettre un ensemble d'états successeurs,*
- $\Omega \subseteq Q$ *est la condition d'acceptation.*

La condition d'acceptation Ω peut prendre différentes formes :
- si $\mathfrak{A}$ est un automate sur les mots *finis*, $\Omega \subseteq Q$,
- si $\mathfrak{A}$ est un *automate de Büchi* sur les mots *infinis*, $\Omega \subseteq Q$,
- si $\mathfrak{A}$ est un *automate de Rabin* sur les mots *infinis*, $\Omega \subseteq 2^Q \times 2^Q$.

L'automate $\mathfrak{A}$ est *déterministe* si pour tout $q \in Q$, et $a \in \Sigma$, $|\delta(q, a)| \leq 1$.

Une *exécution* de $\mathfrak{A}$ sur un mot $\sigma = s_1 s_2 \cdots \in \Sigma^\infty$ est une séquence $\rho = r_0 s_1 r_1 \cdots \in Q \cdot (\Sigma \cdot Q)^{|\sigma|}$ telle que $r_0 = q_0$, et pour tout $0 \leq i < |\sigma|$, $r_{i+1} \in \delta(r_i, s_{i+1})$. Une exécution *acceptante* est définie en fonction du type de l'automate considéré :
- Une exécution ρ d'un automate sur les mots finis est acceptante sur le mot $\sigma \in \Sigma^*$ si le dernier état visité est acceptant, i.e., si $r_{|\sigma|} \in \Omega$.
- Une exécution ρ d'un automate de Büchi est acceptante sur $\sigma \in \Sigma^\omega$ si l'automate visite infiniment souvent au cours de l'exécution un état acceptant, i.e. si $inf(\pi_Q(\rho)) \cap \Omega \neq \emptyset$,
- Une exécution ρ d'un automate de Rabin est acceptante sur le mot $\sigma \in \Sigma^\omega$ s'il existe une paire $(L, U) \in \Omega$ telle que $inf(\pi_Q(\rho)) \cap U \neq \emptyset$ et $inf(\pi_Q(\rho)) \cap L = \emptyset$.

On dit qu'un mot $\sigma \in \Sigma^\infty$ est accepté par $\mathfrak{A}$ s'il existe une exécution acceptante de $\mathfrak{A}$ sur ce mot. L'ensemble des mots acceptés par un automate $\mathfrak{A}$ est appelé le *langage* de l'automate $\mathfrak{A}$ et noté $\mathcal{L}(\mathfrak{A})$.

Un langage $\mathcal{L} \subseteq \Sigma^*$ est dit *régulier* s'il existe un automate $\mathfrak{A}$ tel que $\mathcal{L}(\mathfrak{A}) = \mathcal{L}$.
Un langage $\mathcal{L} \subseteq \Sigma^\omega$ est dit *ω-régulier* s'il existe un automate de Büchi $\mathfrak{A}$ tel que $\mathcal{L}(\mathfrak{A}) = \mathcal{L}$.

1.3.2 Automates d'arbres

Automates d'arbres non-déterministes On définit ici les automate sur des arbres *complets.*

Définition 2.7 (Automate d'arbres non-déterministe)**.** *Un* automate d'arbres non-déterministe *est un sextuplet* $\mathfrak{A} = (X, Y, Q, Q_0, \delta, \alpha)$ *avec*
- X *l'ensemble des directions,*
- Y *l'ensemble (fini) des étiquettes,*
- Q *l'ensemble des* états*,*
- $Q_0 \subseteq Q$ *les* états initiaux*,*
- $\delta : Q \times Y \to 2^{Q^X}$ *la fonction de transition qui associe à un état et une étiquette un ensemble de fonctions attribuant à chaque direction de l'arbre un état successeur,*
- $\alpha \subseteq Q^\omega$ *la condition d'acceptation.*

On assimile le n-uplet $(q_x)_{x\in X}$ à l'application $X \to Q$ définie par $x \mapsto q_x$. Une *exécution* d'un automate non-déterministe $\mathfrak{A}$ sur un (X,Y)-arbre t est un arbre $\rho : X^* \to Q$ telle que $\rho(\varepsilon) \in Q_0$ et pour tout $\sigma \in X^*$, $(\rho(\sigma \cdot x))_{x\in X} \in \delta(\rho(\sigma), t(\sigma))$. Une exécution est *acceptante* si toutes ses branches $x_1x_2\cdots \in X^\omega$ sont telles que $\rho(\varepsilon)\rho(x_1)\rho(x_1x_2)\cdots \in \alpha$. Comme pour les mots, il existe différents types de conditions d'acceptation (Büchi, Rabin, parité,...). Par contre, contrairement au cas des mots, [Rab70] a montré que les automates d'arbres de Büchi étaient strictement moins expressifs que les automates d'arbres de Rabin. Pour plus de précisions, on pourra se référer par exemple à [Tho90]. Un (X,Y)-arbre t est accepté par $\mathfrak{A}$ s'il existe une exécution acceptante de $\mathfrak{A}$ sur t.

Comme pour les mots, l'ensemble des (X,Y)-arbres acceptés par $\mathfrak{A}$ est noté $\mathcal{L}(\mathfrak{A})$.

Automates d'arbres alternants Un automate d'arbres alternant (notion introduite par [MS87]) se définit comme un automate non-déterministe, à la différence que lors d'une transition, on peut choisir d'envoyer *plusieurs copies* de l'automate dans la même direction de l'arbre ou d'ignorer une direction en ne la visitant pas du tout. Ces automates généralisent donc les automates non-déterministes en définissant une transition à l'aide d'une formule logique. Formellement, soit $\mathcal{B}^+(X)$ l'ensemble des formules booléennes positives sur les éléments de X défini par

$$\Phi ::= \top \mid \bot \mid p \mid \alpha \vee \alpha \mid \alpha \wedge \alpha$$

avec $p \in X$. Un ensemble $Y \subseteq X$ satisfait une formule $\Phi \in \mathcal{B}^+(X)$ (on notera $Y \models \Phi$) si la valuation qui affecte $\top$ à tout élément de Y et $\bot$ à tout élément de $X \setminus Y$ vérifie Φ.

Définition 2.8 (Automate d'arbres alternant)**.** *Un* automate d'arbres alternant *est un sextuplet* $\mathfrak{A} = (X, Y, Q, q_0, \delta, \alpha)$ *avec*
- X *l'ensemble des directions,*
- Y *l'ensemble (fini) des étiquettes,*
- Q *l'ensemble des* états*,*
- q_0 *l'*état initial*,*
- $\delta : Q \times Y \to \mathcal{B}^+(X \times Q)$ *la* fonction de transition*,*
- $\alpha \subseteq Q^\omega$ *la condition d'acceptation.*

La fonction de transition associe donc à un état et une étiquette une formule qui suggère une nouvelle configuration de l'automate. Par exemple, si $X = \{0,1\}$, la transition définie par $\delta(q,x) = (0,q_1) \wedge (0,q_2) \vee (0,q_2) \wedge (1,q_2) \wedge (1,q_3)$ signifie que lorsque l'automate est dans l'état q et lit la lettre x, il peut soit envoyer deux copies, une dans l'état q_1, l'autre dans l'état q_2 dans la direction 0, soit envoyer une copie dans la direction 0, dans l'état q_2 et envoyer deux copies, une dans l'état q_2, l'autre dans l'état q_3 dans la direction 1. Il est clair qu'un automate non-déterministe est un cas particulier d'automate alternant, dans lequel la fonction de transition pour chaque état et chaque direction, réécrite sous forme normale disjonctive propose une disjonction de formules de la forme $\bigwedge_{x\in X}\{x\} \times Q$.

Une *exécution* d'un automate alternant $\mathfrak{A}$ sur un (X,Y)-arbre t est un arbre $\rho : X_r^* \to X^* \times Q$, où X_r est un ensemble de directions vérifiant, pour tout $\sigma_r \in \mathrm{dom}(\rho)$, s'il existe $q \in Q$ et $\sigma \in X^*$ tel que $\rho(\sigma_r) = (\sigma, q)$ alors, pour tout $x_r \in X_r$ tel que $\sigma_r \cdot x_r \in \mathrm{dom}(\rho)$, il existe $x \in X$ et $q_x \in Q$ tel que $\rho(\sigma_r \cdot x_r) = (\sigma \cdot x, q_x)$. On définit de plus l'ensemble $S_{\sigma_r} = \{(x, q_x) \in X \times Q \mid \text{ il existe } x_r \in X_r \text{ tel que } \rho(\sigma_r \cdot x_r) = (\sigma \cdot x, q_x)\}$. On demande alors que ρ vérifie :
- $\rho(\varepsilon) = (\varepsilon, q_0)$,

– pour tout $\sigma_r \in X_r^*$ tel qu'il existe $q \in Q$, $\sigma \in X^*$ tels que $\rho(\sigma_r) = (\sigma, q)$, alors $S_{\sigma_r} \models \delta(q, t(\sigma))$.

Une exécution est *acceptante* si toutes ses branches $x_1 x_2 \cdots \in (X_r)^\omega$ sont telles que $\pi_Q(\rho(\varepsilon))\pi_Q(\rho(x_1))\pi_Q(\rho(x_1 x_2)) \cdots \in \alpha$. Un (X, Y)-arbre t est accepté par $\mathfrak{A}$ s'il existe une exécution acceptante de $\mathfrak{A}$ sur t.

Une fois de plus, on note $\mathcal{L}(\mathfrak{A})$ l'ensemble des arbres accepté par l'automate alternant $\mathfrak{A}$.

En fait, les automates d'arbres alternants ne sont pas plus expressifs que les automates d'arbres non-déterministes, mais ils sont exponentiellement plus succints :

Théorème 2.9 ([MS95]). *Un automate d'arbres alternant de Rabin avec m états et k paires peut être transformé en un automate d'arbres non déterministe de Rabin, reconnaissant le même langage, ayant $m^{\mathcal{O}(mk)}$ états, et $\mathcal{O}(mk)$ paires.*

2 Le problème de synthèse et de contrôle distribués

On va présenter dans cette section un formalisme dans lequel exprimer le problème de contrôle de systèmes distribués, et donner un certain nombre de résultats reliés à nos travaux.

2.1 Cadre général

Le problème général peut s'exprimer de la façon suivante : étant donnés un système constitué de plusieurs composants, et une spécification précisant les comportements désirables de ce système, on cherche à produire automatiquement des programmes pour les différents composants assurant que l'ensemble des comportements du système satisfait la spécification. Dans le cas du contrôle, le comportement des processus est déjà en partie déterminé par des programmes, et il s'agit donc de restreindre ces comportements de façon à ne pas violer la spécification. En fait, le problème de synthèse est un cas particulier du problème de contrôle : celui où le programme disponible pour chaque processus autorise à chaque instant toutes les actions possibles.

Dans cette thèse on s'est intéressé au problème de synthèse de programmes réactifs et ouverts, i.e., interagissant avec un environnement non contrôlable. Dans ce cadre, on veut que le système satisfasse la spécification quel que soit le comportement de l'environnement. On présente maintenant plus précisément un certain nombre de travaux effectués ces dernières années sur le sujet, en utilisant le formalisme suivant.

Les systèmes sont représentés par des automates asynchrones tels que définis par Zielonka dans [Zie87] (voir aussi [Zie95]). On considère donc qu'un système est constitué d'un ensemble de registres (ou variables), et d'actions agissant sur ces registres : les actions peuvent lire et modifier certains registres. Cependant, pour capturer dans un même formalisme les exécutions de type asynchrone et de type synchrone, nous allons définir une sémantique plus générale que la sémantique originale ; alors que dans [Zie87], on exécute une action à la fois, on va autoriser plusieurs actions à être jouées *simultanément* (et non pas de façon concurrente).

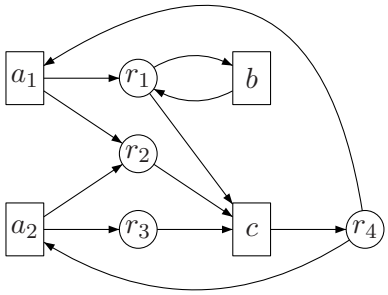


FIG. 2.1 – Une signature

2.1.1 Système distribué.

Une *signature* est un triplet (Σ, V, E) dans lequel

$$\begin{aligned}&\Sigma \text{ est un alphabet d'actions}\\ &V \text{ est un ensemble de variables, ou registres}\\ &E \subseteq (V \times \Sigma) \cup (\Sigma \times V)\end{aligned}$$

La relation E indique, pour chaque action $a \in \Sigma$, quelles sont les variables que a peut lire ($E^{-1}(a)$) et lesquelles a peut modifier ($E(a)$).

Exemple 2.10. Soient

$$\begin{aligned}\Sigma &= \{a_1, a_2, b, c\},\\ V &= \{r_1, r_2, r_3, r_4\},\\ E &= \{(a_1, r_1), (a_1, r_2), (a_2, r_2), (a_2, r_3), (b, r_1), (c, r_4)\}\\ &\quad \cup \{(r_1, b), (r_1, c), (r_2, c), (r_3, c), (r_4, a_1), (r_4, a_2)\}\end{aligned}$$

les éléments d'une signature. Celle-ci est représentée sur la figure 2.1, dans laquelle les actions sont représentées par des rectangles, et les variables par des cercles. Par exemple, l'action c modifie le registre r_4 en fonction des valeurs de r_1, r_2 et r_3, et l'action b lit et modifie le seul registre r_1.

La signature définit donc en quelque sorte le squelette du système. Pour pouvoir décrire son comportement, on dote chaque variable v d'un domaine S^v qu'on supposera fini, et on donne une sémantique à chaque action $a \in \Sigma$: chaque action a est associée à une fonction partielle $\delta_a : S^{E^{-1}(a)} \to S^{E(a)}$, appelée transition locale de l'action a qui définit la façon dont a modifie ses variables en écriture en fonction de la valeur de ses variables en lecture. On remarque, que, contrairement au modèle originel, les transitions δ_a sont des fonctions déterministes. En effet, les contrôleurs que l'on cherche à synthétiser vont autoriser ou non des actions du système, et on considérera toujours que l'effet de cette action sur les registres concernés est déterminé, i.e., en choisissant une action, un contrôleur connaît l'effet de cette action sur les registres.

Un système distribué, (ou *architecture*) est un tuple $\mathcal{A} = (\Sigma, V, E, (S^v)_{v\in V}, s_0, \Delta)$ dans lequel

(Σ, V, E)	est une signature
S^v	pour tout $v \in V$ est le domaine de la variable v
$s_0 \in (S^v)_{v\in V}$	indique la valeur initiale des variables
$\Delta = \{\delta_a : S^{E^{-1}(a)} \to S^{E(a)} \mid a \in \Sigma\}$	définit un ensemble de transitions locales

Dans le cas de systèmes ouverts, les actions de Σ sont partitionnées entre les actions contrôlables par le système (Σ_C) et les actions non contrôlables (et donc contrôlées par l'environnement) (Σ_{NC}) : $\Sigma = \Sigma_C \uplus \Sigma_{NC}$.

Un état global d'une architecture $\mathcal{A}$ est un tuple $(s^v)_{v\in V} \in \Pi_{v\in v} S^v$. Pour $U \subseteq V$, on notera S^U l'ensemble $\Pi_{v\in U} S^v$, et pour tout $s = (s^v)_{v\in V} \in S^V$, pour tout $U \subseteq V$, on notera $s^U = (s^v)_{v\in U}$ la projection de l'état s sur le sous-ensemble U des variables. On étend cette notation aux séquences d'états : pour $\sigma = s_0 s_1 \cdots \in (S^V)^\infty$, pour $U \subseteq V$, on note $\sigma^U = s_0^U s_1^U \cdots$. Dans un état donné, le système peut jouer un certain nombre d'actions. Formellement, on dit qu'une action a est *activable* en un état $s \in S^V$ s'il existe $s' \in S^{E(a)}$ tel que $s' = \delta_a(s^{E^{-1}(a)})$. On note $en(s)$ l'ensemble des actions activables en s.

Exemple 2.11. Reprenons la signature de l'exemple 2.10. Pour obtenir un système distribué, on ajoute les information suivantes. Les actions incontrôlables sont $\Sigma_{NC} = \{a_1, a_2\}$, et on suppose $S^v = \{0, 1\}$ pour tout $v \in V$. On définit par exemple la transition de l'action a_1 par la fonction mettant tous les registres en écriture de a_1 à 1 si $r_4 = 0$ et à 0 si $r_4 = 1$: $\delta_{a_1}(0)^{r_1} = \delta_{a_1}(0)^{r_2} = 1$ et $\delta_{a_1}(1)^{r_1} = \delta_{a_1}(1)^{r_2} = 0$. On définit $\delta_{a_2}(0)^{r_2} = 0$ et $\delta_{a_2}(0)^{r_3} = 1$. Si l'action a_1 est activable dans tous les états du système, ce n'est pas le cas de l'action a_2. L'action b remplace la valeur du registre r_1 par sa négation : $\delta_b(0) = 1$ et $\delta_b(1) = 0$, et l'action c place dans le registre r_4 la valeur de l'addition modulo 2 de tous ses registres en entrée :

δ_c :	r_1	r_2	r_3	r_4
	1	1	1	1
	0	0	1	1
	0	1	0	1
	1	0	0	1
	0	1	1	0
	1	0	1	0
	1	1	0	0
	0	0	0	0

2.1.2 Exécutions.

Pour décrire les exécutions du système, on se dote d'un nouvel ensemble d'actions $\Sigma' \subseteq 2^\Sigma$ indiquant la façon dont les actions peuvent être groupées pour être exécutées en une transition du système. On associe à Σ' la relation $E' \subseteq (V \times \Sigma') \cup (\Sigma' \times V)$ définie, pour tout $A \in \Sigma'$,

par :

$$E'^{-1}(A) = \bigcup_{a \in A} E^{-1}(a) = E^{-1}(A)$$
$$E'(A) = \bigcup_{a \in A} E(a) = E(A)$$

Pour expliciter l'effet d'une action $A \in \Sigma'$ sur les variables qu'elle modifie, on définira la relation $\delta_A \subseteq S^{E^{-1}(A)} \times S^{E(A)}$. Dans le cas où $A = \{a\}$ et $a \in \Sigma$, on supposera que $\delta_A = \delta_a$.

On peut à présent associer à l'architecture $\mathcal{A}$ un système de transition $TS_{\mathcal{A}} = (S^V, \Sigma', \Rightarrow, s_0)$ dans lequel la relation de transition $\Rightarrow \subseteq S^V \times \Sigma' \times S^V$ est définie par $(s, A, s') \in \Rightarrow$ si et seulement si $(s^{E^{-1}(A)}, s'^{E(A)}) \in \delta_A$, et $s'^v = s^v$ pour tout $v \in V \setminus E(A)$. En général, le système de transition $TS_{\mathcal{A}}$ est donc non-déterministe. On dit que le système de transition $TS_{\mathcal{A}}$ a une exécution sur un mot $\alpha = A_1 A_2 \cdots \in \Sigma'^{\infty}$ s'il existe une séquence d'états $\sigma = s_0 s_1 \cdots \in (S^V)^{\infty}$ telle que s_0 est l'état initial de $\mathcal{A}$ et, pour tout $0 \leq i < |\alpha|$, $(s_i, A_{i+1}, s_{i+1}) \in \Rightarrow$. On appelle $\mathcal{L}(\mathcal{A}) \subseteq (\Sigma')^{\infty}$ l'ensemble des mots sur lesquels $TS_{\mathcal{A}}$ a une exécutions et $\mathrm{Runs}(\mathcal{A}) \subseteq S^V \cdot (\Sigma' \cdot S^V)^{\infty}$ l'ensemble des exécutions de $TS_{\mathcal{A}}$. Si δ_A est non-déterministe, on dira que $\mathcal{A}$ est non-déterministe, sinon on dira qu'il est déterministe.

Les actions de Σ sont locales dans le sens où elles ne dépendent pas des valeurs de toutes les variables, et ne modifient pas toutes les variables. En particulier, si deux actions a et $b \in \Sigma$ ont des domaines de lecture et d'écriture disjoints, en partant d'un état global $s \in S^V$ donné, on arrive dans le même état $s' \in S^V$ après avoir joué la séquence ab ou la séquence ba. Plus précisément, on dit que deux actions a et $b \in \Sigma$ sont *causalement liées* si une variable modifiée par b est dans le domaine de lecture de a, ou si une variable modifiée par a est dans le domaine de lecture de b, i.e., si $E^{-1}(a) \cap E(b) \neq \emptyset$ ou si $E(a) \cap E^{-1}(b) \neq \emptyset$. On dit que a et b sont en conflit d'écriture si elles modifient une même variable : $E(a) \cap E(b) \neq \emptyset$. Dans ces deux cas, le fait d'exécuter l'action a a une influence sur le fait d'exécuter l'action b et réciproquement. On étend cette notion aux actions de Σ' et on définit formellement une relation de dépendance $D \subseteq \Sigma' \times \Sigma'$ par

$$ADA' \text{ si et seulement si } (E^{-1}(A) \cap E(A')) \cup (E(A) \cap E^{-1}(A')) \cup (E(A) \cap E(A')) \neq \emptyset \quad (2.1)$$

Remarque 2.12. L'ensemble $\mathcal{L}(\mathcal{A})$ est clos par équivalence de traces de $\mathbb{R}(\Sigma', D)$: si $\alpha \in \mathcal{L}(\mathcal{A})$, alors quel que soit $\beta \in \Sigma'^{\infty}$ tel que $[\alpha] = [\beta]$, $\beta \in \mathcal{L}(\mathcal{A})$.

En effet, soit $\alpha, \beta \in \Sigma'^{\infty}$. Si $[\alpha] = [\beta]$ alors pour tout α' préfixe fini de α il existe $\alpha'' \in \Sigma'^*$ tel que $\alpha'\alpha''$ est un préfixe de α et il existe β' préfixe fini de β tel que $[\alpha'\alpha''] = [\beta']$ (voir par exemple [Gas90] pour une démonstration de cette caractérisation). Supposons que $\alpha \in \mathcal{L}(\mathcal{A})$. Soit $\alpha' \in \Sigma'^*$ préfixe fini de α. Comme $\alpha'\alpha''$ et $\beta' \in \Sigma'^*$, on sait qu'il existe une séquence $\alpha_0, \ldots, \alpha_k$ de mots de Σ'^* vérifiant $\alpha_0 = \alpha'\alpha''$, $\alpha_k = \beta'$ et, pour tout $0 \leq i < k$, il existe $A, B \in \Sigma'$ tels que $A \; I \; B$, $\alpha^1 \in \Sigma'^*$, $\alpha^2 \in \Sigma'^*$ tels que $\alpha_i = \alpha^1 AB \alpha^2$ et $\alpha_{i+1} = \alpha^1 BA \alpha^2$ (caractérisation de l'équivalence de traces sur les mots finis). De plus $\alpha'\alpha'' \in \mathcal{L}(\mathcal{A})$. On montre par récurrence que pour tout i, $\alpha_i \in \mathcal{L}(\mathcal{A})$. Le cas de base $\alpha_0 = \alpha'\alpha'' \in \mathcal{L}(\mathcal{A})$ est trivialement vérifié. Supposons maintenant que pour $0 \leq i < k$, $\alpha_i \in \mathcal{L}(\mathcal{A})$. Alors il existe une exécution $\sigma = s_0 A_1 s_1 A_2 \cdots A_m s_m \in \mathrm{Runs}(\mathcal{A})$ telle que $\pi_{\Sigma'}(\sigma) = \alpha_i$. Donc il existe $n < m$ tel que $s_0 A_1 s_1 \cdots A_n s_n$ est une exécution de $TS_{\mathcal{A}}$ sur α^1, $(s_n, A, s_{n+1}) \in \Rightarrow$, $(s_{n+1}, B, s_{n+2}) \in \Rightarrow$ et, pour tout $n + 2 \leq i < m$, $(s_i, A_{i+1}, s_{i+1}) \in \Rightarrow$. On montre alors qu'il existe $s' \in S^V$ tel que

$(s_n, B, s') \in \Rightarrow$ et $(s', A, s_{n+2}) \in \Rightarrow$. On définit $s' \in S^V$ par $s'^{E(B)} = s_{n+2}^{E(B)}$ et $s'^v = s_n^v$ pour tout $v \in V \setminus E(B)$. Par définition $(s_{n+1}^{E^{-1}(B)}, s_{n+2}^{E(B)}) \in \delta_B$. Comme $A\ I\ B$, on a de plus que $E^{-1}(B) \cap E(A) = \emptyset$ et $(s_n, A, s_{n+1}) \in \Rightarrow$ implique, par définition de δ_A, que $s_{n+1}^{E^{-1}(B)} = s_n^{E^{-1}(B)}$. Donc $(s_n^{E^{-1}(B)}, s'^{E(B)}) \in \delta_B$, et $(s_n, B, s') \in \Rightarrow$. Comme $E^{-1}(A) \cap E(B) = \emptyset$, par définition de s', $s'^{E^{-1}(A)} = s_n^{E^{-1}(A)}$ et comme $E(A) \cap E(B) = \emptyset$, $(s_{n+1}, B, s_{n+2}) \in \Rightarrow$ implique que $s_{n+1}^{E(A)} = s_{n+2}^{E(A)}$. Donc on obtient $(s'^{E^{-1}(A)}, s_{n+2}^{E(A)}) = (s_n^{E^{-1}(A)}, s_{n+1}^{E(A)}) \in \delta_A$. Soit $v \in V \setminus E(A)$. Si $v \in E(B)$, par définition de s', $s'^v = s_{n+2}^v$. Sinon, en utilisant le fait que $(s_n, B, s') \in \Rightarrow$, $(s_n, A, s_{n+1}) \in \Rightarrow$ et $(s_{n+1}, B, s_{n+2}) \in \Rightarrow$, on obtient que $s'^v = s_{n+2}^v$, et donc $(s', A, s_{n+2}) \in \Rightarrow$, et $\alpha^1 B A \alpha^2 = \alpha_{i+1} \in \mathcal{L}(\mathcal{A})$. Donc $\beta' \in \mathcal{L}(\mathcal{A})$. Ainsi on a montré que pour tout α' préfixe fini de longueur n de α, il existe β' préfixe fini de longueur $m \geq n$ de β tel que $\beta' \in \mathcal{L}(\mathcal{A})$. On remarque par ailleurs que s'il existe deux mots $\beta_1, \beta_2 \in \Sigma'^*$ tels que β_1 est un préfixe de β_2, et $\beta_2 \in \mathcal{L}(\mathcal{A})$, alors $\beta_1 \in \mathcal{L}(\mathcal{A})$ et de plus, il existe une exécution de $TS_{\mathcal{A}}$ sur β_1 qui est un préfixe de l'exécution de $TS_{\mathcal{A}}$ sur β_2. Cette remarque nous permet de conclure que, pour tout préfixe fini β' de β, $\beta' \in \mathcal{L}(\mathcal{A})$, et donc, qu'il existe, par passage à la limite, une exécution de $TS_{\mathcal{A}}$ sur β. Donc $\beta \in \mathcal{L}(\mathcal{A})$.

2.1.3 Spécifications.

Une *spécification* du système définit un sous-ensemble des exécutions du système : les exécutions souhaitables. Elle sera souvent donnée sous une forme finie : automate sur les mots représentant les exécutions, formule d'une logique temporelle dont les propositions atomiques portent sur la valeur des états ou sont des égalités entre actions, etc.

Exemple 2.13 (Exemples de spécifications informelles)**.** Pour l'architecture distribuée de l'exemple 2.11, on peut envisager des spécifications demandant que soit respectées les différentes contraintes suivantes :

- « Toute action b mettant r_1 à 1 est suivie par une action c »
- « Toute action a_1 est suivie par une séquence $bbca_2$ »
- « Le registre r_4 vaut alternativement 0 et 1 »
- « À chaque fois que $r_1 = r_2$, on peut atteindre un état dans lequel $r_4 = 0$ ».

On fixe à présent une architecture distribuée $\mathcal{A} = (\Sigma, V, E, (S^v)_{v \in V}, s_0, \Delta)$.

2.1.4 Programmes.

Un *programme* (ou contrôleur, ou stratégie) est une fonction non-déterministe $F : (\Sigma' \cdot S^V)^* \to 2^{\Sigma'}$ qui, en fonction de l'histoire de l'exécution propose le prochain groupe d'actions à jouer. Ces programmes respectent l'architecture $\mathcal{A}$, donc, pour chaque séquence d'actions et d'états correspondant à une exécution possible de $TS_{\mathcal{A}}$, le programme ne propose que des actions activables dans le dernier état visité. On appelle *programme du système* un programme qui ne restreint que les actions contrôlables. Formellement, on définit les programmes du système (ou stratégies du système) de la façon suivante :

Définition 2.14 (Programmes du système)**.** *On dit que $F : (\Sigma' \cdot S^V)^* \to 2^{\Sigma'}$ est un* programme du système *si, pour tout $\alpha = (A_1 \cdot s_1 \cdots A_i \cdot s_i) \in (\Sigma' \cdot S^V)^i$, les conditions suivantes*

sont vérifiées (avec $s_i = s_0$ si $\alpha = \varepsilon$) :

$$F(\alpha) \subseteq en(s_i) \tag{2.2}$$
$$F(\alpha) \cap \Sigma_{NC} = en(s_i) \cap \Sigma_{NC} \tag{2.3}$$

On dit que le programme F est *non-bloquant* quand F ne peut proposer un ensemble vide d'actions que si le dernier état visité est bloquant (si aucune action n'est activable dans cet état).

Définition 2.15 (Programmes non-bloquants)**.** *On dit que $F : (\Sigma' \cdot S^V)^* \to 2^{\Sigma'}$, programme du système est un* programme non-bloquant *si, pour tout $\alpha = (A_1 s_1 \cdots A_i s_i) \in (\Sigma' \cdot S^V)^i$, et $i \geq 0$ (avec $s_i = s_0$ si $\alpha = \varepsilon$),*

$$\text{si } en(s_i) \neq \emptyset \text{ alors } F(\alpha) \neq \emptyset \tag{2.4}$$

Lorsque le système suit l'avis de la stratégie, l'ensemble des exécutions possibles de $\mathcal{A}$ est restreint. On définit les exécutions suivant une stratégie F (ou exécutions F-compatibles) comme étant les mots de $\text{Runs}(\mathcal{A})$ formés des actions conseillées par la stratégie. Formellement,

Définition 2.16 (Exécution F-compatible)**.** *Une* exécution selon la stratégie F *(ou F-*compatible*) est un mot $\alpha = s_0 A_1 s_1 \cdots \in S^V \cdot (\Sigma' \cdot S^V)^\infty$ tel que s_0 est l'état initial de $\mathcal{A}$, $A_1 \in F(\varepsilon)$ et, pour tout $i > 1$, $A_i \in F(A_1 s_1 \cdots A_{i-1} s_{i-1})$.*

On note $\text{Runs}_F(\mathcal{A})$ l'ensemble des exécutions F-compatibles de l'architecture $\mathcal{A}$.

Remarque 2.17. Tels qu'ils sont définis, les programmes prennent leurs décisions basés sur une exécution vue comme un *mot*. Donc, si $\mathcal{L}(\mathcal{A})$ est fermé par équivalence de traces, cela n'est pas forcément le cas de $\text{Runs}_F(\mathcal{A})$.

On remarque également que, de la même façon que le préfixe d'une exécution est toujours une exécution, tout préfixe d'une exécution F-compatible est une exécution F-compatible. On introduit la notion de *maximalité* d'une exécution, qui indique lorsque l'exécution est « terminée » en un certain sens, c'est-à-dire, lorsqu'elle est infinie, ou lorsque la stratégie ne propose plus d'actions contrôlables. En effet, la stratégie proposant toujours toutes les actions incontrôlables possibles, on laisse la possibilité à l'environnement de n'effectuer qu'un nombre fini d'actions et donc une exécution dans laquelle la stratégie *restreinte aux actions du système* n'est plus définie, peut être terminée. Par ailleurs, si après une séquence finie d'actions, la stratégie propose encore des actions contrôlables, on considère que l'exécution n'est pas terminée : si l'environnement ne joue plus, alors une action contrôlable sera effectuée. On définit formellement les exécutions maximales comme suit.

Définition 2.18 (Exécution F-maximale)**.** *Pour toute stratégie du système $F : (\Sigma' \cdot S^V)^* \to 2^{\Sigma'}$, on dit qu'une exécution F-compatible $\alpha = s_0 \alpha' \in S^V \cdot (\Sigma' \cdot S^V)^\infty$ avec $s_0 \in S^V$ état initial de $\mathcal{A}$ et $\alpha' \in (\Sigma' \cdot S^V)^\infty$, est* F-maximale *si $\alpha \in S^V \cdot (\Sigma' \cdot S^V)^\omega$ ou si $F(\alpha') \cap \Sigma_C = \emptyset$.*

On note l'ensemble des exécutions F-maximales de $\mathcal{A}$: $\text{Runs}_F^{\max}(\mathcal{A}) \subseteq \text{Runs}_F(\mathcal{A})$.

2.1.5 Arbres d'exécutions

Les exécutions du système peuvent être représentées par un arbre $t : \Sigma'^* \to S^V$ donnant les différentes exécutions de $\mathcal{A}$ possibles : pour chaque nœud $\alpha \in \Sigma'^*$, l'étiquette du nœud $t(\alpha)$ donne l'état courant du système, et ce nœud a un successeur par action globale activable dans cet état. Formellement,

Définition 2.19 (Arbre d'exécutions)**.** *Un arbre $t : \Sigma'^* \to S^V$ est un* arbre d'exécution de $\mathcal{A}$ *si*

- $t(\varepsilon) = s_0$
- *pour tout $\alpha \in \Sigma'^*$, si $\alpha \in dom(t)$ et $t(\alpha) = s \in S^V$, alors l'ensemble $\{A \in \Sigma' \mid \alpha \cdot A \in dom(t)\} = en(s)$ et, pour tout $A \in en(s)$, $t(\alpha \cdot A) = s'$ avec $(s, A, s') \in \Rightarrow$.*

Si A est non-déterministe, on peut avoir plusieurs arbres d'exécutions de $\mathcal{A}$. On appelle RunTrees($\mathcal{A}$) l'ensemble des arbres d'exécutions de $\mathcal{A}$.

Soit $F : (\Sigma' \cdot S^V)^* \to \Sigma'$ une stratégie du système $\mathcal{A}$. Un arbre d'exécutions selon la stratégie F est un arbre d'exécutions dans lequel le branchement a été restreint : pour chaque nœud, on ne considère que les actions autorisées par la stratégie à cet instant. Pour $t : \Sigma'^* \to S^V$, on définit la fonction $\mathrm{Val}(t) : \Sigma'^* \to (\Sigma' \cdot S^V)^*$ qui associe à toute branche finie de t représentant un mot fini de $\mathcal{L}(\mathcal{A})$ l'exécution de $TS_{\mathcal{A}}$ correspondante (privée de l'état initial). Formellement, pour tout $\alpha \in \Sigma'^*$,

$$\mathrm{Val}(t)(\varepsilon) = \varepsilon$$
$$\mathrm{Val}(t)(\alpha \cdot A) = \mathrm{Val}(t)(\alpha) \cdot A \cdot t(\alpha \cdot A)$$

Définition 2.20 (Arbre d'exécutions selon une stratégie)**.** *Soit $F : (\Sigma' \cdot S^V)^* \to 2^{\Sigma'}$ une stratégie du système $\mathcal{A}$. Un arbre $t_F : \Sigma'^* \to S^V$ est un* arbre d'exécutions selon F *si*

- $t_F(\varepsilon) = s_0$
- *pour tout $\alpha \in \Sigma'^*$, si $\alpha \in dom(t_F)$ et $t_F(\alpha) = s \in S^V$, alors l'ensemble $\{A \in \Sigma' \mid \alpha \cdot A \in dom(t_F)\} = F(Val(t_F)(\alpha))$ et, pour tout $A \in F(Val(t_F)(\alpha))$, $t_F(\alpha \cdot A) = s'$ avec $(s, A, s') \in \Rightarrow$.*

Les arbres d'exécutions selon une stratégie F sont donc des sous-arbres des arbres d'exécutions du système. Pour une stratégie F fixée, on note $\mathrm{RunTrees}_F(\mathcal{A})$ l'ensemble des arbres d'exécutions selon F.

2.1.6 Le problème de synthèse de contrôleurs.

On peut à présent définir le problème de synthèse de contrôleurs : étant données une architecture $\mathcal{A}$ et une spécification φ, existe-t-il un programme F du système tel que l'ensemble des exécutions F-compatibles et F-maximales de $\mathcal{A}$ est inclus dans l'ensemble des exécutions acceptables défini par la specification φ ? Si un tel programme existe, peut-on le produire automatiquement ? On dira dans ce cas que F est un programme distribué vérifiant (ou une stratégie distribuée gagnante pour) $(\mathcal{A}, \varphi)$. Par ailleurs, si pour une architecture fixée, le problème est indécidable, on dira que l'architecture elle-même est indécidable (pour la variante du problème considérée).

En ne considérant que des variables et des actions, ce modèle fait abstraction de la notion de processus. Pour plus de clarté, nous allons faire réapparaître cette notion par la suite. Un processus est simplement un ensemble de registres. On considère que l'ensemble des processus

forme une partition de l'ensemble des variables, donc qu'une variable ne peut pas faire partie de deux processus distincts.

Par ailleurs, la façon dont nous avons défini un contrôleur pour un système ne tient pas compte de l'aspect distribué de ce dernier : il prend ses décisions en fonction de l'état global du système. Une approche plus intéressante est de chercher à synthétiser des contrôleurs qui respectent l'architecture. Il y a plusieurs façons de formaliser cette intuition ; nous allons maintenant en présenter quelques unes. Les principaux paramètres envisagés concernent la façon de distribuer les stratégies, le type de mémoire autorisée pour ces stratégies, et la façon de se donner la spécification. Par ailleurs, le problème de contrôle distribué implique que les processus peuvent se transmettre de l'information entre eux. La modélisation des communications autorisées est également un paramètre du problème. On va en distinguer principalement deux : le formalisme dans lequel un processus peut lire certains registres d'un autre (ce qu'on va appeler communication par variables partagées), et celui dans lequel il peut connaître certaines actions effectuées par un autre (qu'on va appeler communication par synchronisation d'actions). Enfin, deux types d'exécutions ont été envisagées pour les systèmes : les exécutions totalement synchrones, et les exécutions asynchrones.

2.2 Comportement synchrone

Dans un système à comportement synchrone, on considère qu'il existe une horloge globale réglant l'avancement de tous les composants du système. On exécute donc plusieurs actions de Σ de façon *simultanée*.

2.2.1 Caractéristiques du problème en sémantique synchrone

Communication par variables partagées On note Proc l'ensemble des processus et on partitionne les variables en sous-ensembles V_p, $V = \biguplus_{p \in \mathrm{Proc}} V_p$ tels que V_p est l'ensemble des registres constituant le processus p. Dans ce modèle, la communication entre processus se fait par variables partagées. Les actions sont toutes locales à un seul processus, dans le sens où elles ne modifient les registres que d'un processus : pour tout $a \in \Sigma$, il existe $p \in \mathrm{Proc}$ tel que $E(a) \subseteq V_p$. Pour tout $p \in \mathrm{Proc}$, on note $\Sigma^p = \{a \in \Sigma \mid E(a) \subseteq V_p\}$. Les variables modifiées par les actions incontrôlables du système forment un processus particulier, p_{env}, représentant l'environnement. On note $\mathrm{Proc_S} = \mathrm{Proc} \setminus \{p_{env}\}$ l'ensemble des processus contrôlables du système. Un processus p peut donc lire certaines variables écrites par d'autres en fonction du domaine de lecture des actions de Σ^p. Pour simplifier, on considère que, pour tout $p \in \mathrm{Proc}$, pour tout $a, b \in \Sigma^p$, $E^{-1}(a) = E^{-1}(b)$ et $E(a) = E(b)$, et on identifie les variables en lecture et en écriture d'un processus en fonction de ses actions : $E^{-1}(p) = E^{-1}(a)$ et $E(p) = E(a)$ pour $a \in \Sigma^p$.

On se place dans un contexte d'environnement maximal, donc on impose que toutes les actions incontrôlables soient proposables à chaque instant.[1]. On rappelle que les transitions associées aux actions de Σ sont déterministes (ceci afin de donner suffisamment de pouvoir aux programmes lorsqu'ils choisissent leurs actions). Cette restriction choisie par rapport au modèle plus général de [Zie87], qui autorisait des transitions non-déterministes nous impose de considérer un grand nombre d'actions. En particulier, les actions incontrôlables du système doivent permettre d'effectuer n'importe quelle transition. Dans ce modèle, cela revient à dire

[1] Le problème de synthèse de contrôleur interagissant avec un environnement *réactif* a été traité par exemple dans [KMTV00].

qu'à chaque instant l'environnement peut écrire n'importe quelle valeur sur les variables qu'il contrôle : on demande donc que pour tout $s \in S^V$, $\Sigma_{NC} \subseteq en(s)$ et, pour tout $s \in S^{E^{-1}(p_{env})}$, pour tout $s' \in S^{E(p_{env})}$, il existe une action incontrôlable $a \in \Sigma_{NC}$ telle que $\delta_a(s) = s'$.

Les variables d'un processus $p \in \text{Proc}_S$ se divisent en deux catégories : ses variables privées, et ses variables de communication (i.e. lues par un autre processus). Comme on peut choisir pour une variable donnée un domaine arbitrairement grand, on peut, sans perte de généralité, supposer que chaque processus n'a qu'une seule variable privée, v_p, lue et modifiée par le processus p, et représentant son état de contrôle. Formellement, il existe un sous-ensemble des variables $\{v_p \mid p \in \text{Proc}\} \subseteq V$ tel que pour tout $p \in \text{Proc}_S$, cette variable est à la fois lue et modifiée par le processus p : $v_p \in (E(p) \cap E^{-1}(p))$ et cette variable n'est lue ni modifiée par aucun autre processus : pour tout $p' \in \text{Proc}$ tel que $p' \neq p$, $v_p \notin E(p) \cup E^{-1}(p)$. On note Com l'ensemble des variables de communication du système : $\text{Com} = V \setminus \{v_p \mid p \in \text{Proc}_S\}$, $\text{In}(p) = E^{-1}(p) \cap \text{Com}$ l'ensemble des variables lues par le processus $p \in \text{Proc}_S$ en dehors de son état de contrôle, et $\text{Out}(p) = E(p) \cap \text{Com}$ l'ensemble des variables qu'il modifie. On peut donc réécrire, pour chaque action $a \in \Sigma^p$ la fonction de transition de la façon suivante : $\delta_a : S^{v_p} \times S^{\text{In}(p)} \to S^{v_p} \times S^{\text{Out}(p)}$.

On appelle *graphe d'une architecture* le graphe formé par les relations entre actions contrôlables et variables de communication : $(\Sigma_C \uplus \text{Com}, E \cap (\text{Com} \times \Sigma_C \cup \Sigma_C \times \text{Com}))$. C'est une restriction du graphe représentant la signature du système, puisqu'on rappelle que pour une architecture distribuée $\mathcal{A}$ ayant pour ensemble d'actions $\Sigma = \Sigma_C \uplus \Sigma_{NC}$ et ensemble de variables $V = \uplus_{p \in \text{Proc}} V_p$, la relation $E \subseteq (\Sigma \times V) \cup (V \times \Sigma)$. Or, ici $\Sigma_C \subsetneq \Sigma$ et $\text{Com} \subsetneq V$. On appelle *architecture acyclique* une architecture dont le graphe est acyclique. En particulier, dans une architecture acyclique, les actions associées à un processus ne dépendent pas des variables du processus - à l'exception de son état de contrôle. Pour représenter une architecture, on dessinera uniquement son graphe de communications, dans lequel on assimilera les actions de Σ^p et le processus $p \in \text{Proc}_S$ (voir figure 2.2). On a supprimé de cette définition de graphe de communication les états de contrôle et les actions incontrôlables. En fait, même si les états de contrôle et les actions incontrôlables induisent des cycles dans la signature initiale, on verra dans la définition des exécutions (plus particulièrement dans la remarque 2.23) que ces cycles n'induisent jamais de *dépendance cyclique*.

Exemple 2.21. La signature représentée sur la figure 2.1 ne peut donner lieu à une architecture distribuée respectant les contraintes que l'on vient de donner. En effet, tout d'abord l'environnement n'est pas maximal : les actions qu'il peut effectuer dépendent de l'état global du système, et les valeurs des variables écrites par les actions incontrôlables ne sont pas toutes possibles. De plus, la variable r_1 est modifiée par des actions ne pouvant appartenir au même processus, puisque a_1 est incontrôlable quand b l'est. Considérons par exemple la signature suivante :

$$\Sigma = \{a_1, a_2, a_3, a_4, b_1, b_2, b_3, c_1, c_2\}$$
$$V = \{r_1, r_2, r_3, r_4, r'_3, r'_4, v_p, v_q\}$$

avec $\Sigma_{NC} = \{a_1, a_2, a_3, a_4\}$, pour laquelle la relation E est représentée sur la figure 3.1(a). On a ici $\text{Proc} = \{p_{env}, p, q\}$ définissant la partition de V : $V_{p_{env}} = \{r_1, r_2\}$, $V_p = \{r_3, r'_3, v_p\}$, et $V_q = \{r_4, r'_4, v_q\}$. On a également $\Sigma^{p_{env}} = \Sigma_{NC} = \{a_1, a_2, a_3, a_4\}$, $\Sigma^p = \{b_1, b_2, b_3\}$ et $\Sigma^q = \{c_1, c_2\}$. On a donc $\text{Com} = \{r_1, r_2, r_3, r_4, r'_3, r'_4\}$, et par exemple $\text{In}(p) = \{r_1\}$ et $\text{Out}(p) = \{r_3, r'_3\}$. Le graphe de l'architecture associée est donc formé des nœuds $\{b_1, b_2, b_3, c_1, c_2\}$ et

$\{r_1, r_2, r_3, r_4, r'_3, r'_4\}$. On a représenté sur la figure 3.1(b) le graphe de communication correspondant. N'apparaissent dans le graphe de communication représenté que les informations « publiques ». Les actions Σ^p d'un processus p et son état de contrôle ne sont pas accessibles aux autres processus, elles sont donc assimilées au processus p vu comme une « boîte noire ». Seules ses variables $\mathrm{Out}(p)$ sont visibles. De même, on ne représente pas le cycle constitué par les variables en lecture et en écriture de p_{env} mais on représente les variables de $E(p_{env})$ avec un arc entrant, et celles de $E^{-1}(p_{env})$ avec un arc sortant.

Délai des processus On pourra considérer qu'un délai peut intervenir dans la transmission des valeurs entre processus, i.e., qu'un processus ne peut pas lire la valeur courante d'une variable d'un autre processus, mais uniquement celle qui a été écrite k instants auparavant. On associe donc à chaque processus un entier correspondant à cet intervalle de temps. Dans la littérature, seuls les délais 0 ou 1 ont été considérés. Pour ne pas alourdir la présentation, nous allons également considérer que la transmission est soit instantanée (0-délai), soit avec délai de 1 (1-délai). On présente au chapitre 3 un modèle et des résultats permettant de considérer des délais arbitraires sur les processus. Pour le moment, pour $p \in \mathrm{Proc_S}$, on note $d_p \in \{0, 1\}$ son délai d'accès à l'information. On considère que l'environnement a toujours un délai de 0. Le délai associé à un processus s'exprime dans la transition d'une action, comme on va le voir dans le paragraphe suivant.

Exécutions À chaque instant de l'exécution, chaque processus choisit une action à exécuter, et c'est la combinaison de toutes ces actions locales qui est jouée.

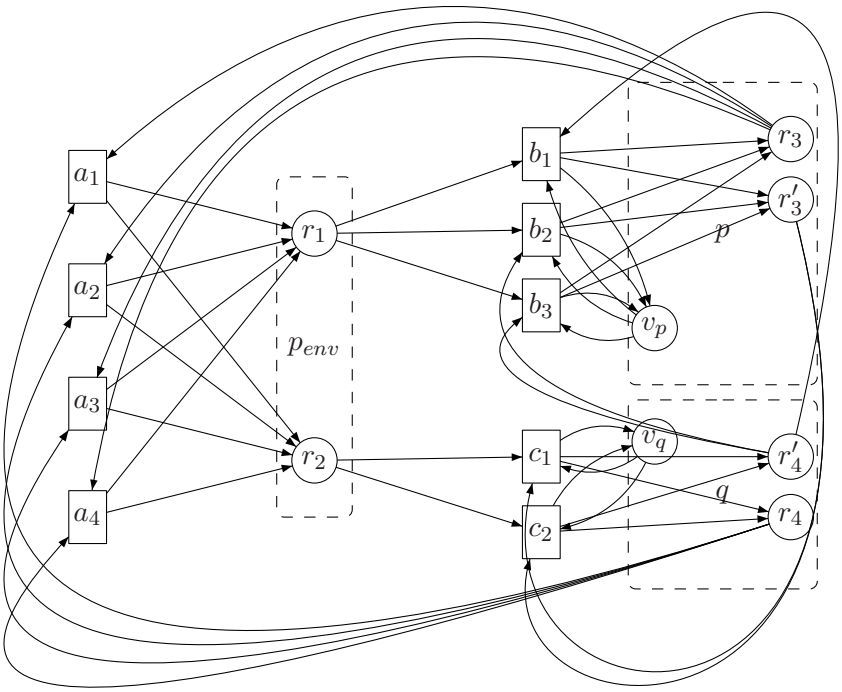
On définit donc les actions effectuées par le système en une unité de temps par $\Sigma' = \{A \in 2^\Sigma \mid \forall p \in \mathrm{Proc}, |\Sigma^p \cap A| = 1\}$. Pour tout $A \in \Sigma'$, on note $A(p)$ l'élément de $\Sigma^p \cap A$. On remarque que $D = (\Sigma')^2$ donc dans le cas synchrone, les traces de Mazurkiewicz de $\mathbb{R}(\Sigma', D)$ se confondent avec les mots de Σ'^∞. Soit $A \in \Sigma'$, on définit $\delta_A \subseteq S^{E^{-1}(A)} \times S^{E(A)}$ par l'ensemble $\{(s_1, s_2) \in S^{E^{-1}(A)} \times S^{E(A)}\}$ vérifiant

$$s_2^{E(p_{env})} = \delta_{A(p_{env})}(s_1^{E^{-1}(p_{env})}) \tag{2.5}$$

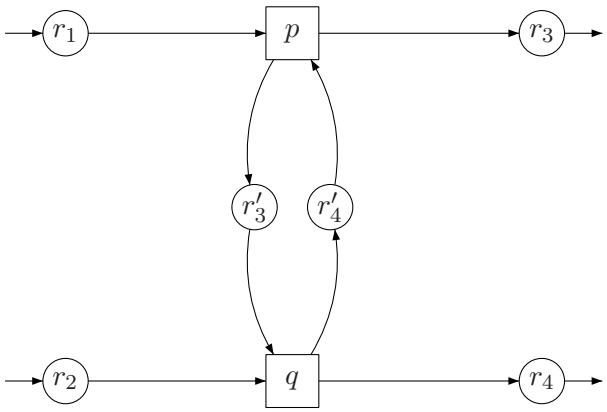
$$s_2^{E(p)} = \delta_{A(p)}((s_1^{v_p}, s_{2-d_p}^{\mathrm{In}(p)})) \text{ pour tout } p \in \mathrm{Proc_S} \tag{2.6}$$

L'effet de l'action choisie par le processus contrôlable $p \in \mathrm{Proc_S}$ dépend de la valeur de son état de contrôle à l'instant précédent, et de la valeur de ses variables en lecture *au même instant d'exécution* si le processus est 0-délai, et à l'instant précédent s'il est 1-délai.

Remarque 2.22. On remarque que pour exprimer des délais plus importants que 1, il est nécessaire de modifier un peu le modèle général : en effet, la valeur de l'état local $s_i^{E(p)}$ dépend de l'état local $s_{i-d_p}^{E^{-1}(p)}$. Lorsque le délai est inférieur ou égal à 1, il suffit pour déterminer le prochain état atteint lors d'une transition de connaître la valeur de l'état courant. Cependant, si le délai est strictement supérieur à 1, la valeur du prochain état est déterminé par des états locaux plus anciens. On a donc besoin de considérer qu'un état de $TS_{\mathcal{A}}$ est un tuple d'éléments de S^V : $(S^V)^D$ où $D = max(1, max_{p \in \mathrm{Proc_S}}\{d_p\})$: pour calculer une transition, on a besoin de connaître la valeur de l'état courant (pour mettre à jour le registre des processus p, v_p), et la valeur de l'état le plus ancien sur lequel peut s'appliquer une action de $A \in \Sigma'$. En effet, si $s_0 A_1 s_1 A_2 s_2 \in S^V \cdot (\Sigma' \cdot S^V)^2$ est une exécution de $TS_{\mathcal{A}}$, soit $p \in \mathrm{Proc_S}$ tel que $d_p = 2$, alors $s_2^{E(p)} = \delta_{A_2(p)}(s_1^{v_p}, s_0^{\mathrm{In}(p)})$. Une transition de $TS_{\mathcal{A}}$ est donc dans ce cas-là de la forme $(s_0 s_1, A_2, s_1 s_2)$, et on a besoin de considérer qu'un état global est un élément de $(S^V)^2$.



(a) Un exemple de signature



(b) Le graphe de l'architecture

FIG. 2.2 – Une architecture distribuée dans le modèle synchrone

Remarque 2.23. Si l'architecture distribuée $\mathcal{A}$ est acyclique, ou si les processus de $\mathcal{A}$ sont tous à délai, les relations δ_A sont en fait des fonctions déterministes. On voit à présent que l'effet des actions des processus ne dépend de la valeur de leur état de contrôle qu'à l'instant précédent, le cycle que ces registres constituent dans le graphe n'induit donc pas de non-déterminisme. C'est pour la même raison qu'on n'inclut pas non plus les actions de l'environnement dans la définition du graphe de communication, et donc d'une architecture acyclique.

Exemple 2.24. Reprenons la signature de l'exemple 2.21, dans laquelle on suppose que tous les processus sont 0-délai. On pose $S^v = \{0, 1\}$ pour tout $v \in V$. On pose également $\delta_{a_1}(s) = (1, 1)$ pour tout $s \in S^{r_3, r_4}$ et

δ_{b_1} :	r_1	r'_4	v_p	r_3	r'_3	v_p
	1	1	0	0	0	1
	1	0	0	0	1	1
	0	1	0	1	1	1
	0	0	0	1	0	1

δ_{c_1} :	r_2	r'_3	v_q	r_4	r'_4	v_q
	1	1	0	0	0	1
	1	0	0	0	1	1
	0	1	0	1	1	1
	0	0	0	1	0	1.

Soit l'action $A = (a_1, b_1, c_1) \in \Sigma'$. Alors, d'après les égalités (2.5) et (2.6), pour $s_1 \in S^{E^{-1}(A)}$ défini par, pour tout $v \in V$, $s_1^v = 0$, il existe $s_2, s'_2 \in S^V$ tels que (s_1, s_2) et $(s_1, s'_2) \in \delta_A$:

δ_A :	r_1	r_2	r_3	r_4	r'_3	r'_4	v_p	v_q	r_1	r_2	r_3	r_4	r'_3	r'_4	v_p	v_q
(s_1, s_2)	0	0	0	0	0	0	0	0	1	1	0	0	1	0	1	1
(s_1, s'_2)	0	0	0	0	0	0	0	0	1	1	0	0	0	1	1	1

Si par contre, les délais des processus p et q sont à 1, alors la fonction δ_A est déterministe.

Programmes Par la suite on va considérer que $en(s) \cap \Sigma^p \neq \emptyset$ pour tout état $s \in S^V$ et tout processus $p \in \text{Proc}$, c'est-à-dire qu'un processus n'est jamais bloqué. Comme par ailleurs on se restreint aux programmes non-bloquants du système, la condition (2.4) implique que pour tout $\alpha \in S^V \cdot (\Sigma' \cdot S^V)^*$, $F(\alpha) \cap \Sigma_C \neq \emptyset$, et donc que les exécutions F-maximales (voir définition 2.18) sont toutes *infinies*.

De plus, on s'intéresse aux contrôleurs distribués parmi les processus. On dit qu'un programme $F : (\Sigma' \cdot S^V)^* \to 2^{\Sigma'}$ est distribué parmi les processus s'il existe un tuple de stratégies *locales*, $f^p : (\Sigma' \cdot S^V)^* \to 2^{\Sigma^p}$ pour tout $p \in \text{Proc}$ tel que, pour tout $\alpha \in (\Sigma' \cdot S^V)^*$, $F(\alpha) = \{A \in \Sigma' \mid \forall p \in \text{Proc}, A(p) \in f^p(\alpha)\}$. On se restreint de plus à des programmes déterministes pour les processus du système, i.e., pour tout $p \in \text{Proc}_\text{S}$, $f^p : (\Sigma' \cdot S^V)^* \to \Sigma^p$.

La difficulté du contrôle de systèmes distribués vient du fait que chaque contrôleur n'a qu'une vision locale de l'état du système. On modélise cette contrainte par le fait que l'on demande que les stratégies des processus soient à mémoire *locale*, i.e., le programme de chaque processus ne peut tenir compte de toute l'histoire globale du système, mais uniquement de l'histoire sur les variables qu'il peut lire. Formellement, on impose aux stratégies de respecter la condition suivante :

Définition 2.25 (Stratégie à mémoire locale)**.** *On dit que la stratégie f^p du processus $p \in$ Proc est* à mémoire locale *si, pour tous $\alpha = A_1 s_1 \cdots A_i s_i$, $\alpha' = A'_1 s'_1 \cdots A'_i s'_i \in (\Sigma' \times S^V)^i$, tels que*

$$(s_1 \cdots s_i)^{E^{-1}(p)} = (s'_1 \cdots s'_i)^{E^{-1}(p)}$$

on a

$$f^p(\alpha) = f^p(\alpha').$$

On dit que la stratégie distribuée $F = (f^p)_{p\in \mathrm{Proc}}$ est à mémoire locale si pour tout $p \in \mathrm{Proc_S}$, f^p est à mémoire locale.

Remarque 2.26. La contrainte (2.2) impose que pour tout $p \in \mathrm{Proc}$, tout $\alpha = A_1 s_1 \cdots A_i s_i \in (\Sigma' \cdot S^V)^*$, $f^p(\alpha) \subseteq en(s_i)$, avec $s_i = s_0$ si $\alpha = \varepsilon$. Or, pour tout $s, s' \in S^V$ tels que $s^{E^{-1}(p)} = {s'}^{E^{-1}(p)}$, $en(s) \cap \Sigma^p = en(s') \cap \Sigma^p$, donc cette contrainte ne dépend que de la vision locale de la stratégie.

De plus, comme le système est non-bloquant et l'environnement est maximal, les conditions (2.2), (2.3) et (2.4) peuvent se réécrire au niveau local de la façon suivante : pour tout $\alpha = A_1 s_1 \cdots A_i s_i \in (\Sigma' \cdot S^V)^*$, avec $s_i = s_0$ si $\alpha = \varepsilon$

$$f^p(\alpha) \subseteq en(s_i) \cap \Sigma^p \qquad \text{pour tout } p \in \mathrm{Proc_S} \tag{2.7}$$

$$f^{p_{env}}(\alpha) = en(s_i) \cap \Sigma_{NC} = \Sigma_{NC} \tag{2.8}$$

$$f^p(\alpha) \text{ définie} \qquad \text{pour tout } p \in \mathrm{Proc_S} \tag{2.9}$$

Spécifications Dans ce modèle de communication par variables partagées, les spécifications vont porter sur les séquences d'états du système visités au cours d'une exécution. Par la suite, on va considérer plus précisément des spécifications données par des formules logiques, typiquement des formules de LTL, CTL, CTL*, μ-calcul ou MSO. Plusieurs types de spécifications ont été abordées dans la littérature :

- les spécifications dites *externes*, qui ne portent que sur les variables de communication avec l'environnement (i.e., lues et modifiées par le processus environnement)
- les spécifications *locales*, qui ne relient des valeurs de variables que si elles sont en lecture ou en écriture d'un même processus
- les spécifications *totales*, les plus générales, qui peuvent contraindre l'ensemble des valeurs des variables du système de façon non restreinte.

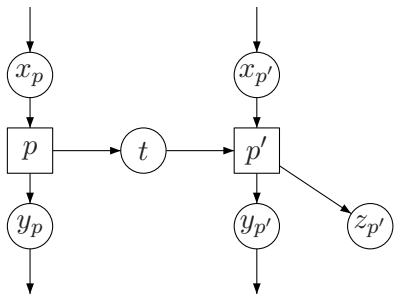


Fig. 2.3 – Exemple d'architecture

Exemple 2.27 (Exemples de spécifications informelles)**.** Considérons l'architecture représentée sur la figure 2.3, dans laquelle on respecte encore la convention que les variables écrites par l'environnement sont représentées avec un arc entrant, et les variables lues par l'environnement avec un arc sortant. Par exemple, la variable $z_{p'}$ est une variable du processus p' qui n'est lue par aucun processus. On peut choisir d'exprimer les contraintes suivantes :

Spécifications externes	1. À chaque instant, $x_p = y_p$ et $x_{p'} = y_{p'}$
	2. À chaque instant, $x_p = y_{p'}$
	3. À chaque instant, si $x_p = x_{p'}$ alors $y_p = y_{p'}$
Spécifications locales	1. À chaque instant, $x_p = y_p$ et $x_{p'} = y_{p'}$
	2. À chaque instant, $x_p = t$ et $t = y_{p'}$
	3. À chaque instant, si $t = x_{p'}$ alors $y_{p'} = z_{p'}$
Spécifications totales	1. À chaque instant, $x_p = y_p$ et $x_{p'} = y_{p'}$
	2. À chaque instant, $x_p = y_{p'}$ et $t = 0$
	3. À chaque instant, si $x_p = x_{p'}$ alors $y_p = y_{p'}$

Les deuxième et troisième spécifications externes ne sont pas des spécifications locales : on ne peut pas lier deux variables qui ne sont pas connectées au même processus. Par contre, on peut exprimer de façon équivalente pour le problème de contrôle la deuxième spécification par une spécification locale faisant intervenir la variable de communication t. Réciproquement, la troisième spécification locale n'est pas une spécification externe, car d'une part elle fait intervenir une variable de communication, t, et d'autre part, elle restreint une variable qui n'est pas une variable de sortie, mais une variable du processus p' qui lui est privée. La deuxième spécification totale n'est ni une spécification externe (elle contraint la variable de communication t, en l'empêchant de transmettre de l'information), ni une spécification locale (elle relie deux variables qui ne sont reliées à aucun processus en commun).

Formellement, on note respectivement $V_{\mathrm{I}} = E(\Sigma_{NC})$ et $V_{\mathrm{O}} = \{v \in V \mid E(v) = \Sigma_{NC}\}$ les variables d'entrée et de sortie du système. Soit $U \subseteq V$ un sous-ensemble des variables. Pour $\mathcal{L} \in \{\mathsf{LTL}, \mathsf{CTL}, \mathsf{CTL}^*, \mu\text{-calcul}, \mathsf{MSO}\}$, on note $\mathcal{L}(U)$ l'ensemble des fomules de $\mathcal{L}$ utilisant des propositions atomiques AP de la forme $(v = a)$ pour $v \in U$ et $a \in S^v$. Un mot $\sigma = s_0 s_1 \cdots \in (S^V)^\omega$ est donc assimilé à une séquence $u : \mathbb{N} \to (2^{AP})$ dans laquelle, pour tout $v \in U$, $a \in S^v$, $(v = a) \in u(i)$ si et seulement si $s_i^v = a$.

Remarque 2.28. Si $\varphi \in \mathcal{L}(U)$ avec $U \subseteq V$, soit $\sigma \in (S^V)^\omega$ un modèle de φ. Alors $\sigma \models \varphi$ si et seulement si $\sigma^U \models \varphi$.

Une spécification est dite *externe* si elle appartient à $\mathcal{L}(V_{\mathrm{I}} \cup V_{\mathrm{O}})$. Elle est dite *locale* si elle s'obtient par combinaison booléenne de formules dans $\mathcal{L}(E^{-1}(p) \cup E(p))$ pour tout $p \in \mathrm{Proc}$. Sinon elle est dite *totale*.

Pour déterminer si une exécution du système satisfait la spécification $\varphi \in \mathcal{L}(U)$, on va définir $Spec(\varphi) \subseteq S^V \cdot (\Sigma' \cdot S^V)^\omega$.

Pour φ formule de temps linéaire, on définit $Spec(\varphi) = \{\alpha \in S^V \cdot (\Sigma' \cdot S^V)^\omega \mid \pi_{S^V}(\alpha) \models \varphi\}$, i.e, l'ensemble des exécutions (infinies, car le problème de synthèse se restreint à des exécutions F-maximales pour un programme F fixé) qui visitent une séquence d'états qui est un modèle de φ.

Pour φ formule branchante, on définit $Spec(\varphi) = \{t : \Sigma'^* \to S^V \mid t \models \varphi\}$, i.e., l'ensemble des arbres (qu'on comparera avec les arbres d'exécutions selon une stratégie F donnée) qui sont modèles de φ.

La synthèse de contrôleur en comportement synchrone Le problème de synthèse se distingue du problème de contrôle par le fait que dans le premier cas, aucun programme n'existe. Cela signifie que chaque processus ne comporte qu'un seul état de contrôle à partir

duquel toutes les actions sont toujours possibles. Formellement, lorsque l'architecture $\mathcal{A}$ est telle que pour tout $p \in \text{Proc}_S$,

- $|S^{v_p}| = 1$, i.e., il n'y a qu'un seul état de contrôle,
- il existe une application bijective $\delta^p : \Sigma^p \to (S^{E(p)})^{S^{E^{-1}(p)}}$ telle que, pour tout $a \in \Sigma^p$, $\delta(a) = \delta_a$, i.e., pour tous les processus du système, toutes les actions possibles sont toujours activables (les actions incontrôlables sont toutes possibles et activables car on considère un environnement maximal, comme expliqué au début de la section 2.2.1),

on dit que $\mathcal{A}$ est une architecture *à synthétiser*. Dans les autres cas, on dit que $\mathcal{A}$ est une architecture *à contrôler*.

Définition 2.29 (Le problème de synthèse (respectivement de contrôle) de système distribué synchrone, avec spécifications externes (respectivement locales, totales))**.** *Étant donnés*

- *une architecture* $\mathcal{A} = (\Sigma, V, E, (S^v)_{v\in V}, s_0, (\delta_a)_{a\in\Sigma})$ *à synthétiser (respectivement à contrôler),*
- *un ensemble* Proc*, contenant un élément particulier* p_{env}*, partitionnant les variables*
- *un tuple* $(d_p)_{p\in\text{Proc}}$ *donnant les délais de chaque processus,*
- *une spécification externe (respecivement locale, totale)* φ*,*

existe-t-il une stratégie distribuée non-bloquante $F = (f^p)_{p\in\text{Proc}}$ *à mémoire locale telle que l'ensemble des exécutions* F*-maximales* $Runs_F^{max}(\mathcal{A}) \subseteq Spec(\varphi)$ *? Si* φ *est une formule branchante, existe-t-il* $F = (f^p)_{p\in\text{Proc}}$ *stratégie distribuée non-bloquante et à mémoire locale telle que* $RunTrees_F(\mathcal{A}) \subseteq Spec(\varphi)$*?*

Lorsque, pour tout $p \in \text{Proc}_S$, $d_p = 0$ on dit que l'architecture est 0-*délai*. Lorsque, pour tout $p \in \text{Proc}_S$, $d_p = 1$, on dit que l'architecture est 1-*délai*.

Remarque 2.30. Comme on l'a dejà relevé dans la remarque 2.23, lorsque l'architecture est acyclique ou 1-délai, les fonctions δ_A sont déterministes. Comme les stratégies que l'on cherche sont également déterministes pour les processus du système Proc_S, le choix fait au début de la section de se restreindre à des actions $a \in \Sigma$ déterministes a la conséquence que, en choisissant un tuple d'actions, la stratégie peut prévoir leur effet sur les variables du système. Formellement, si l'on fixe une séquence d'actions de l'environnement $\alpha_{env} \in \Sigma_{NC}^{\omega}$, alors il existe une *unique* exécution F-compatible $\alpha \in S^V \cdot (\Sigma' \cdot S^V)^{\omega}$, telle que, si on note $\pi_{\Sigma'}(\alpha) = A_1 A_2 \cdots$, alors $A_1(p_{env})A_2(p_{env}) \cdots = \alpha_{env}$.

2.2.2 Quelques résultats de la littérature

Le problème de synthèse de système *centralisé* est un cas particulier de synthèse de systèmes distribués dans lequel $|\text{Proc}_S| = 1$. Dans ce cas, les spécifications totales et locales sont équivalentes, et, si l'on suppose que toutes les variables sont lues par l'environnement, équivalentes aux spécifications externes. Historiquement, le problème de synthèse de système centralisé synchrone avec des spécifications données par des formules dans $\mathsf{MSO}(V)$ a été posé par Church dans [Chu63], et a été résolu pour la première fois dans [BL69], puis dans [Rab72]. Plus récemment, [PR89a] (voir aussi [Ros92]) a proposé un formalisme moderne, avec des spécifications LTL, et donne une solution plus simple au problème (avec une meilleure complexité). Dans notre modèle, certaines variables de l'environnement peuvent être invisibles pour le processus contrôlable (elles ne font pas partie de son domaine de lecture), et cependant prises en compte dans la spécification. Cette notion d'*information incomplète* est d'ailleurs un premier pas vers la résolution du problème de synthèse ou de contrôle de systèmes distribués, puisque

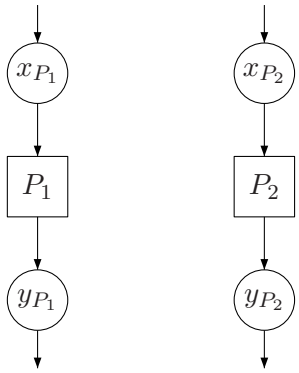


FIG. 2.4 – Architectures $\mathcal{A}_0$ indécidable

dans ce dernier cas, le programme que l'on cherche à synthétiser pour le processus, étant à mémoire locale, a une information seulement partielle de l'état global du système.

On dit donc que le problème de contrôle (ou de synthèse) pour $(\mathcal{A}, \varphi)$ est *à information incomplète* s'il existe $v \in V$, $p \in \text{Proc}_S$, tels que $v \notin E^{-1}(p)$. Sinon, il est *à information complète*.

Il est donc clair qu'en général, lorsque $|\text{Proc}_S| > 1$, le problème est à information incomplète. Le problème de synthèse avec information incomplète a été traité dans [KV97], qui a également étendu les résultats de [PR89a] aux spécifications branchantes. (Une version complète des résultats de [KV97] a été publiée dans [KV99]). On peut donc établir les résultats suivants :

Théorème 2.31 ([PR89a, Ros92])**.** *Le problème de synthèse de système centralisé synchrone avec information complète est décidable, et* 2EXPTIME*-complet, pour des architectures* 0*-délai et des spécifications dans* $\mathsf{LTL}(V)$*.*

Théorème 2.32 ([KV99, KV00])**.** *Le problème de synthèse de système centralisé synchrone avec information incomplète est décidable pour des architectures* 0*-délai, et est* EXPTIME*-complet pour des spécifications dans* $\mathsf{CTL}(V)$ *et du* μ*-calcul, et* 2EXPTIME*-complet, pour des spécifications dans* $\mathsf{LTL}(V)$ *et* $\mathsf{CTL}^*(V)$*.*

Le cas général de synthèse de système distribué synchrone a été étudié pour la première fois dans [PR90] (plus précisément la variante synthèse de système distribué synchrone sans délai avec des spécifications externes). En s'inspirant de résultats de [PR79] sur les jeux distribués, ils obtiennent l'indécidabilité de ce problème en général :

Théorème 2.33 ([PR90], adapté dans [FS05])**.** *Le problème de synthèse de système distribué synchrone est indécidable avec spécifications externes et totales de* LTL *et* CTL *pour les architectures 0-délai.*

En fait, on verra au chapitre suivant qu'on peut adapter ce résultat à des architectures ayant n'importe quels délais sur ses processus.

La preuve d'indécidabilité de [PR90] repose sur une réduction du problème de l'arrêt d'une machine de Turing au problème de synthèse de systèmes distribués synchrone sans délai pour une spécification $\varphi \in \mathsf{LTL}$ et l'architecture $\mathcal{A}_0$ suivante (voir figure 2.4) :

$$
\begin{aligned}
V &= \{x_{P_1}, y_{P_1}, x_{P_2}, y_{P_2}, v_{P_1}, v_{P_2}\} \\
\text{Proc} &= \{P_1, P_2, p_{env}\} \\
V_{P_1} &= \{y_{P_1}, v_{P_1}\}, V_{P_2} = \{y_{P_2}, v_{P_2}\} \\
\Sigma_C &= \{a_{00}^{P_1}, a_{10}^{P_1}, a_{01}^{P_1}, a_{11}^{P_1}, a_{00}^{P_2}, a_{10}^{P_2}, a_{01}^{P_2}, a_{11}^{P_2}\} \\
\Sigma_{NC} &= \{a_{00}^{env}, a_{01}^{env}, a_{10}^{env}, a_{11}^{env}\} \\
E &= (\Sigma_{NC} \times \{x_{P_1}, x_{P_2}\}) \cup (\{y_{P_1}, y_{P_2}\} \times \Sigma_{NC}) \\
&\quad \cup (\{x_{P_1}, v_{P_1}\} \times \{a_{00}^{P_1}, a_{10}^{P_1}, a_{01}^{P_1}, a_{11}^{P_1}\}) \cup (\{a_{00}^{P_1}, a_{10}^{P_1}, a_{01}^{P_1}, a_{11}^{P_1}\} \times \{y_{P_1}, v_{P_1}\}) \\
&\quad \cup (\{x_{P_2}, v_{P_2}\} \times \{a_{00}^{P_2}, a_{10}^{P_2}, a_{01}^{P_2}, a_{11}^{P_2}\}) \cup (\{a_{00}^{P_2}, a_{10}^{P_2}, a_{01}^{P_2}, a_{11}^{P_2}\} \times \{y_{P_2}, v_{P_2}\}) \\
S^v &= \{0,1\} \text{ pour tout } v \in \text{Com} \\
S^{v_{P_i}} &= \{0\} \text{ pour } i = 1,2 \\
s_0^v &= 0 \text{ pour tout } v \in V \\
\delta_{a_{ij}^{env}}(s) &= (i,j) \text{ pour tout } s \in S^{E^{-1}(a_{ij}^{env})}, \text{ avec } i,j = 0,1 \\
\delta_{a_{00}^i}(s)^{y_{P_i}} &= 0 \text{ pour tout } s \in S^{E^{-1}(a_{00}^i)}, \text{ avec } i = P_1, P_2 \\
\delta_{a_{01}^i}(s)^{y_{P_i}} &= s \text{ pour tout } s \in S^{E^{-1}(a_{01}^i)}, \text{ avec } i = P_1, P_2 \\
\delta_{a_{10}^i}(s)^{y_{P_i}} &= 1 - s \text{ pour tout } s \in S^{E^{-1}(a_{10}^i)}, \text{ avec } i = P_1, P_2 \\
\delta_{a_{11}^i}(s)^{y_{P_i}} &= 1 \text{ pour tout } s \in S^{E^{-1}(a_{11}^i)}, \text{ avec } i = P_1, P_2 \\
d_{P_1} &= d_{P_2} = 0
\end{aligned}
$$

Ce théorème établit donc l'indécidabilité du problème de contrôle de systèmes distribués synchrones en général. Cependant, il est possible d'identifier des cas particuliers pour lesquels le problème est décidable : les travaux suivants ont donc cherché à définir pour quelles sous-classes d'architecture ou de spécifications le problème était décidable. Même si un certain nombre de résultats positifs ont pu être obtenus, ces sous-classes restent assez restreintes. On commence par décrire certaines classes d'architectures pour lesquelles des résultats ont été prouvés (voir figure 2.5) :

Définition 2.34. *Une architecture distribuée* $\mathcal{A} = (\Sigma, V, E, (S^v)_{v \in V}, s_0, (\delta_a)_{a \in \Sigma})$ *comportant l'ensemble de processus* Proc *dans laquelle* Proc_S *est isomorphe à* $\{p_1, \cdots, p_n\}$ *est de type*

- pipeline *si*
  - $\text{In}(p_1) \subseteq V_I$
  - *pour tout* $2 \leq i \leq n$, $\text{In}(p_i) \subseteq E(p_{i-1})$.
- anneau *si*
  - $\text{In}(p_1) \subseteq (V_I \cup E(p_n))$
  - *pour tout* $2 \leq i \leq n$, $\text{In}(p_i) \subseteq E(p_{i-1})$.
- pipeline à double sens *si*
  - $\text{In}(p_1) \subseteq (V_I \cup E(p_2))$
  - $\text{In}(p_n) \subseteq E(p_{n-1})$
  - *pour tout* $2 \leq i \leq n-1$, $\text{In}(p_i) \subseteq (E(p_{i-1}) \cup E(p_{i+1}))$.
- anneau à double sens *si*
  - $\text{In}(p_1) \subseteq (V_I \cup E(p_2) \cup E(p_n))$

  - $\mathrm{In}(p_n) \subseteq (E(p_1) \cup E(p_{n-1}))$
  - *pour tout* $2 \leq i \leq n-1$, $\mathrm{In}(p_i) \subseteq (E(p_{i-1}) \cup E(p_{i+1}))$.
- pipeline à double entrée *si*
  - $V_I = V_1 \uplus V_2$
  - $\mathrm{In}(p_1) = V_1$
  - $\mathrm{In}(p_n) \subseteq V_2 \cup E(p_{n-1})$
  - *pour tout* $2 \leq i \leq n-1$, $\mathrm{In}(p_i) \subseteq E(p_{i-1})$

Le premier résultat de décidabilité a été prouvé dans [PR90] et établit la décidabilité du problème de synthèse de système distribué synchrone avec spécifications externes de LTL pour les architectures pipeline 0-délai. Plus tard, [KV01] ont étendu ce résultat en considérant des spécifications totales de CTL*. Comme une spécification externe est aussi une spécification totale, si le problème de synthèse de systèmes distribués est décidable pour une architecture donnée avec des spécifications totales, il l'est aussi avec des spécifications externes. De plus, CTL* étendant LTL, on établit le résultat plus général suivant :

Théorème 2.35 ([KV01])**.** *Le problème de synthèse de système distribué synchrone avec spécifications totales de* CTL* *est décidable en temps non-élémentaire pour les architectures de type pipeline 0-délai ou 1-délai.*

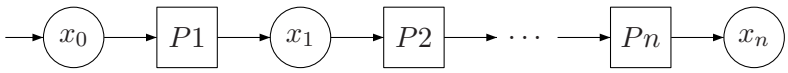
Cette complexité est également une borne inférieure, ceci découlant d'un résultat plus ancien sur les jeux distribués [PR79]. On remarque qu'en fait ce théorème recouvre et étend le théorème 2.32. Il nous donne donc également la décidabilité du problème de synthèse de systèmes centralisés synchrones avec information incomplète pour des architectures 1-délai. Par ailleurs, la preuve de décidabilité du pipeline de [KV01] s'étend également aux anneaux et pipelines à double sens. D'où :

Théorème 2.36 ([KV01])**.** *Le problème de synthèse de système distribué synchrone avec spécifications totales de* CTL* *est décidable pour les architectures* 1*-délai de type anneau et pipeline à double sens.*

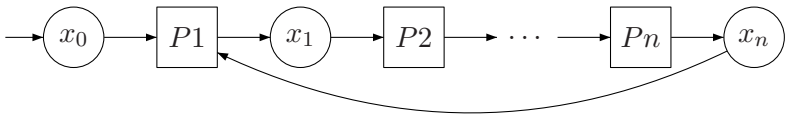
Toutes ces architectures décidables ont un point commun : l'information se transmet de façon linéaire parmi les processus. L'approche de [KV01] a été reprise et étendue dans [FS05] afin de montrer que cette caractéristique est en fait un critère de décidabilité du problème de synthèse de système distribué synchrone.

Définition 2.37 (Architecture ordonnée)**.** *Une architecture est ordonnée si on peut totalement ordonner les processus de* Proc *par la relation définie par* $p \leq q$ *si et seulement si, pour tout* $v_1 \in V_I$*, pour tout* $v_q \in E^{-1}(q)$ *et pour toute séquence* $v_2, \ldots, v_{n-1} \in V$ *telle que* $v_1 E^2 v_2 E^2 \ldots E^2 v_{n-1} E^2 v_q$ *il existe* $1 \leq i \leq n$ *tel que* $v_i \in E^{-1}(p)$*.*

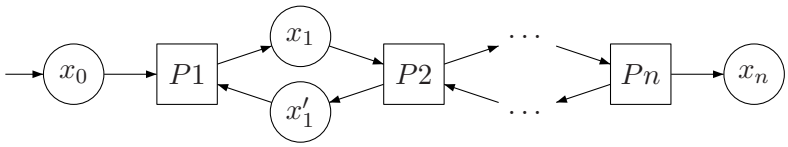
La valeur d'une variable lue par le processus q lui donne une certaine information sur ce qu'a joué l'environnement, et par là, les stratégies des autres processus lui étant connues, lui donne une connaissance partielle de l'état global. Ce que dit donc cette définition, c'est que si toute information de l'environnement parvenant à q a d'abord « transité » par p, alors $p \leq q$ dans le sens « p a une meilleure connaissance de l'état global que q ». Une architecture est donc ordonnée si on ne peut pas trouver deux processus ayant une connaissance incomparable de l'état global du système.



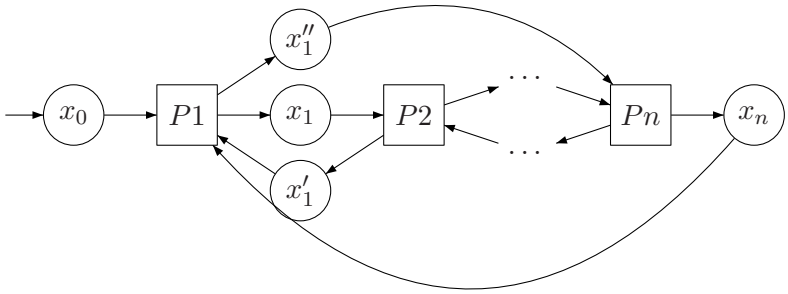
(a) Pipeline



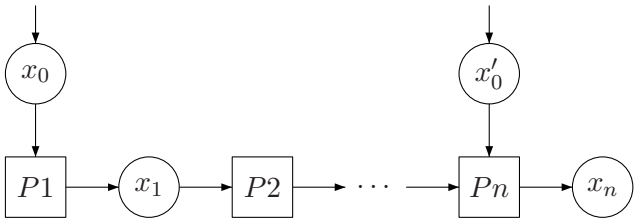
(b) Anneau



(c) Pipeline à double sens



(d) Anneau à double sens



(e) Pipeline à double entrée

FIG. 2.5 – Différentes classes d'architectures

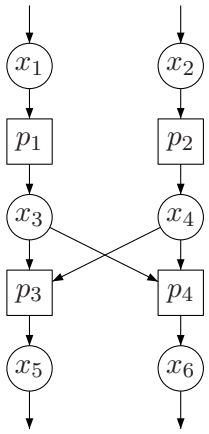


FIG. 2.6 – Une architecture décidable pour des spécifications externes

Théorème 2.38 ([FS05])**.** *Le problème de synthèse de système distribué synchrone avec spécifications totales du μ-calcul est décidable pour une architecture $\mathcal{A}$ 1-délai si et seulement si $\mathcal{A}$ est ordonnée.*

Ce critère permet de conclure à l'indécidabilité du problème de synthèse distribuée synchrone avec spécifications totales pour les pipelines à double entrée et les anneaux à double sens à partir d'une certaine taille.

Comme on l'a vu, si le problème de synthèse de système distribué est décidable pour des spécifications totales, il l'est pour des spécifications externes ou totales. Par contre, s'il est indécidable pour des spécifications totales, il ne l'est pas forcément pour des spécifications externes, le pouvoir d'expression supplémentaire des formules totales pouvant être trop fort – par exemple [PR90] ont montré que le problème de synthèse de système distribué synchrone avec spécifications *externes* pour l'architecture représentée figure 2.6 n'est pas plus difficile que la synthèse de système centralisé synchrone, et donc décidable. Or cette architecture n'est pas ordonnée au sens de la définition 2.37, les processus p_1 et p_2 ayant une connaissance incomparable de l'état global du système. Le théorème 2.38 assure donc que le problème de synthèse de système distribué pour cette architecture et des spécifications *totales* est indécidable. Ce critère de décidabilité n'est donc pas transposable aux autres types de spécifications. En particulier, [MT01] ont établi la décidabilité du problème pour les pipelines à double entrée si on se restreint aux spécifications locales :

Théorème 2.39 ([MT01])**.** *Le problème de contrôle de systèmes distribués synchrones avec spécifications locales de* LTL *est décidable pour les architectures* 0*-délai de type pipeline à double entrée.*

Remarque 2.40. Le théorème 2.39 a en fait été établi par [MT01] pour des spécifications locales de type Rabin portant sur les états locaux des processus, ce qui recouvre les spécifications LTL. Par ailleurs, en posant $V_2 = \emptyset$, on retrouve les pipelines de [PR90]. Le théorème 2.39 élargit donc d'une part la classe des architectures pour lesquelles le problème est décidable, et étend d'autre part la décidabilité du pipeline avec spécifications locales au cas du contrôle.

Cependant, le théorème suivant montre que même en considérant des spécifications locales, la classe d'architectures pour lesquelles le problème de contrôle de systèmes distribués synchrones est décidable reste limitée. On commence par définir la notion de sous-architecture.

Soit $G = (V, E)$ un graphe. On dit que $G' = (V', E')$ est un sous-graphe de G si $V' \subseteq V$ et $E' = E \cap (V' \times V')$.

Définition 2.41. *Une architecture distribuée $\mathcal{A}'$ est une* sous-architecture *de l'architecture $\mathcal{A}$ si le graphe de $\mathcal{A}'$ est isomorphe à un sous-graphe du graphe de $\mathcal{A}$.*

On peut maintenant établir :

Théorème 2.42 ([MT01])**.** *Le problème de contrôle de systèmes distribués synchrones avec des spécifications locales est décidable pour une architecture 0-délai $\mathcal{A}$ si et seulement si toutes les composantes connexes du graphe de $\mathcal{A}$ sont des sous-architectures d'un pipeline à double entrée.*

Les résultats présentés ci-dessus avaient tous la caractéristique de se placer dans des cas où les architectures étaient acycliques, ou 1-délai. Comme expliqué dans la remarque 2.23, les fonctions δ_A sont donc des fonctions déterministes, pour tout $A \in \Sigma'$. Dans un travail plus récent, [BJ06] étudie le problème de synthèse de système distribué synchrone 0-délai, dans des architectures comportant des cycles. Pour une même séquence d'actions de l'environnement, il existe donc plusieurs exécutions compatibles avec une stratégie. Tel qu'exprimé actuellement, le problème de synthèse est très exigeant vis-à-vis de la stratégie : il faut que toutes les exécutions compatibles avec la stratégies satisfassent la spécification. On peut envisager d'autres interprétations d'une stratégie gagnante. On va donc paramétrer le problème par le type *strict*, *angélique* ou *démoniaque* de la spécification (reprenant par là la terminologie de [BJ06] : pour $\varphi \in \mathsf{MSO}(U)$, on définit

$$
\begin{aligned}
Spec_{strict}(\varphi) =& \{\alpha \in S^V \cdot (\Sigma' \cdot S^V)^\omega \mid \text{ il existe un unique } \alpha' \in \mathrm{Runs}(\mathcal{A}), \pi_{\Sigma'}(\alpha) = \pi_{\Sigma'}(\alpha') \\
& \text{et } \pi_{S^V}(\alpha) \models \varphi\} \\
Spec_{angel}(\varphi) =& \{\alpha \in S^V \cdot (\Sigma' \cdot S^V)^\omega \mid \text{ il existe } \alpha' \in \mathrm{Runs}(\mathcal{A}), \text{ tel que } \pi_{\Sigma'}(\alpha) = \pi_{\Sigma'}(\alpha') \\
& \text{et } \pi_{S^V}(\alpha') \models \varphi\} \\
Spec_{demon}(\varphi) =& \{\alpha \in S^V \cdot (\Sigma' \cdot S^V)^\omega \mid \text{ pour tout } \alpha' \in \mathrm{Runs}(\mathcal{A}) \text{ tel que } \pi_{\Sigma'}(\alpha) = \pi_{\Sigma'}(\alpha') \\
& \pi_{S^V}(\alpha') \models \varphi\}
\end{aligned}
$$

Le problème exprimé dans la définition 2.29 correspond aux spécifications strictes si l'on se restreint aux architectures déterministes, et aux spécifications démoniaques sinon.

Ils obtiennent le résultat suivant :

Théorème 2.43 ([BJ06])**.** *Le problème de synthèse de système distribué synchrone avec spécifications totales de* MSO *strictes ou angéliques est décidable pour une architecture 0-délai $\mathcal{A}$ si et seulement si $\mathcal{A}$ est ordonnée.*

Le problème reste ouvert pour les interprétations démoniaques.

Dans toutes les variantes du problème énoncées ci-dessus, lorsque le problème est décidable, alors il existe une stratégie gagnante pour une donnée $(\mathcal{A}, \varphi)$ si et seulement si il existe une stratégie gagnante *à mémoire finie* pour $(\mathcal{A}, \varphi)$.

2.3 Comportement asynchrone

Dans un système à comportement asynchrone, les processus avancent à des vitesses variables, et se synchronisent de temps en temps pour communiquer. Un *ordonnanceur* permet aux processus de s'exécuter. C'est dans le modèle asynchrone que la notion de trace de Mazurkiewicz comme représentation des exécutions prend tout son sens. En effet, dans un système distribué asynchrone, un certain nombre d'actions peuvent avoir lieu sans que l'état d'une partie des processus en soit affecté, ou pendant qu'un sous-ensemble des processus effectuent d'autres actions qui n'ont pas d'effet sur les premières. Ceci est traduit par la notion d'actions *indépendantes* de la théorie des traces. Une exécution vue comme une trace permet donc de regrouper un ensemble d'exécutions équivalentes à ordonnancement près. Dans ce contexte, un autre type de mémoire pour les contrôleurs a été envisagé : les contrôleurs à mémoire *causale*. Si un contrôleur à mémoire locale ne dépend que de ce qu'il a lui-même observé (que ce soit les variables des autres processus qu'il a eu l'occasion de lire, ou les actions auxquelles il a participé), un contrôleur à mémoire causale par contre peut dépendre de toutes les actions qui ont eu lieu dans son passé causal, i.e., les actions qui ont eu lieu dans son passé dans *toutes* les linéarisations de la trace représentant l'exécution en cours.

2.3.1 Communication par variables partagées

Le modèle de communication par variables partagées lorsque l'on considère des systèmes asynchrones signifie que les processus ne peuvent lire les variables des autres processus et écrire sur leurs propres variables qu'aux instants décidés par l'ordonnanceur. En particulier, dans une exécution asynchrone, un processus lisant des variables écrites par un autre n'a aucun moyen de savoir combien de fois ce registre a été modifié depuis le précédent instant où il y a eu accès.

Caractéristiques des architectures La classe d'architectures utilisée est similaire à celle définie dans la section 2.2.1 pour les systèmes synchrones, avec les variations suivantes. Pour tout processus $p \in \mathrm{Proc_S}$, on distingue les actions de lecture seule Σ_r^p, les actions d'écriture seule Σ_w^p et les actions de lecture et écriture simultanées Σ_m^p, avec $\Sigma^p = \Sigma_r^p \uplus \Sigma_w^p \uplus \Sigma_m^p$.

Les actions de lecture seule permettent juste au processus de lire les variables auxquelles il a accès, sans modifier les siennes (excepté son état de contrôle). Dans ce cas précis, pour $a \in \Sigma_r^p$, $E(a) = \{v_p\}$. Les actions d'écriture seule permettent d'écrire sur les variables du processus, mais ne dépendent pas de la valeur courante des variables en lecture : $E^{-1}(a) = \{v_p\}$. On demande de plus que les actions de lecture et d'écriture soient alternées : pour tout processus $p \in \mathrm{Proc_S}$, le domaine de son état de contrôle v_p contient une composante indiquant si la dernière action jouée a modifié les variables. Formellement, $S^{v_p} = S \times \{R, W\}$ pour S domaine quelconque. Les fonctions de transitions locales respectent les restrictions suivantes :

$$\begin{aligned}
&\text{Pour tout } a \in \Sigma_r^p,\ \delta_a : S \times \{R\} \times S^{\mathrm{In}(p)} \to S \times \{W\}\\
&\text{Pour tout } a \in \Sigma_w^p,\ \delta_a : S \times \{W\} \to S \times \{R\} \times S^{\mathrm{Out}(p)}\\
&\text{Pour tout } a \in \Sigma_m^p,\ \delta_a : S \times \{R\} \times S^{\mathrm{In}(p)} \to S \times \{R\} \times S^{\mathrm{Out}(p)}\\
&\qquad s_0^{v_p} \in S \times \{R\}
\end{aligned}$$

On dit qu'une architecture est *à lecture et écriture alternées* si les actions de Σ_m^p ne sont jamais autorisées : pour tout $p \in \mathrm{Proc_S}$, pour tout $s \in S^V$, $en(s) \cap \Sigma_m^p = \emptyset$. On dit qu'elle

est *à lecture et écriture simultanées* si les processus n'utilisent que des actions de lecture et écriture simultanée, i.e., si, pour tout $p \in \text{Proc}_S$, tout $s \in S^V$, $en(s) \cap \Sigma^p \subseteq \Sigma^p_m$. Dans ce cas, l'asynchronisme du système se traduit dans les exécutions du système de transitions associé à $\mathcal{A}$.

Exécutions Classiquement, dans une exécution asynchrone d'un système distribué, on considère qu'à chaque transition de $TS_{\mathcal{A}}$, une seule action est exécutée. La *concurrence* possible entre deux actions effectuées par deux processus est capturée par la notion de trace de Mazurkiewicz. Dans ce modèle, $\Sigma' = \Sigma$ et donc les transitions locales considérées sont celles données dans la définition du système distribué, Δ (et dans ce cas, notre définition d'exécution d'un système rejoint la définition classique des exécutions linéarisées des automates de [Zie87]). Une autre approche des exécutions asynchrones, a été choisie dans [FS06], reprenant l'idée de modélisation de [MW03] pour les jeux distribués adaptés à la synthèse. Dans ce modèle, on conserve la notion d'horloge globale et, à chaque instant, l'ordonnanceur autorise certains processus (ou aucun) à s'exécuter. L'asynchronisme est modélisé par le fait qu'un processus ne connaît pas la relation entre son évolution et celle de l'horloge globale. La notion d'actions concurrentes n'est pas explicitement rendue, et on considère plutôt la *simultanéité* éventuelle, mais non garantie, de certains processus. Dans ce cas, $\Sigma' = \bigcup_{P \in 2^{\text{Proc}} \setminus \emptyset} \{A \in 2^{\Sigma} \mid \forall p \in P, |A \cap \Sigma^p| = 1\}$. Par la suite on dira que ce modèle est un modèle d'exécutions *pseudo-synchrones*, et on considérera que toutes les architectures sont 1-délai (afin de n'avoir que des architectures déterministes).

Stratégies On s'intéresse aux stratégies distribuées parmi les processus. Dans le modèle asynchrone (par opposition au pseudo-synchrone), une stratégie distribuée $F : (\Sigma \cdot S^V)^* \to 2^{\Sigma}$ est une stratégie telle qu'il existe un tuple de stratégies locales $(f^p)_{p \in \text{Proc}}$ avec $f^p : (\Sigma \cdot S^V)^* \to 2^{\Sigma^p}$, et telles que, pour tout $\alpha \in (\Sigma \cdot S^V)^*$, $F(\alpha) = \bigcup_{p \in \text{Proc} \mid f^p(\alpha) \text{ défini}} f^p(\alpha)$.

Une stratégie distribuée $F : (\Sigma' \cdot S^V)^* \to 2^{\Sigma'}$ dans le modèle pseudo-synchrone est une stratégie telle qu'il existe un tuple de stratégies locales $(f^p)_{p \in \text{Proc}_S}$ avec $f^p : (\Sigma' \cdot S^V)^* \to 2^{\Sigma^p}$, et telles que, pour tout $\alpha \in (\Sigma' \cdot S^V)^*$, $F(\alpha) = \bigcup_{\emptyset \neq P \subseteq \{p \in \text{Proc} \mid f^p(\alpha) \text{ défini}\}} \{(a^p)_{p \in P} \mid a^p \in f^p(\alpha)\}$. La stratégie distribuée F propose donc l'ensemble des tuples d'actions proposées par ses stratégies. Comme ce n'est pas le système, mais l'ordonnanceur qui va décider quel ensemble de processus sera autorisé à jouer, on impose à la stratégie F de proposer des tuples d'actions concernant tous les sous-ensembles possibles de processus.

Dans les deux cas, la condition (2.3) implique que, pour tout $\alpha = A_1 s_1 \cdots A_i s_i \in (\Sigma' \cdot S^V)^*$, $f^{p_{env}}(\alpha) = en(s_i) \cap \Sigma_{NC}$ (avec $s_i = s_0$ si $\alpha = \varepsilon$).

On dit que la stratégie distribuée F est *déterministe* si $f^p : (\Sigma' \times S^V)^* \to \Sigma^p$ pour tout $p \in \text{Proc}_S$.

On dit qu'elle est *totale* si f^p est une fonction totale pour tout $p \in \text{Proc}_S$.

Encore une fois, on se restreint aux programmes à mémoire locale, c'est-à-dire que le contrôleur de chaque processus ne dépend que de la valeur de ses variables en lecture. Ici, plus précisément, le contrôleur ne dépend que de la valeur des variables qu'il a effectivement eu l'occasion de lire au cours de l'exécution. Pour cela, on définit pour chaque processus $p \in \text{Proc}_S$, une fonction $view_p^{VP} : (\Sigma' \cdot S^V)^* \to (S^{E^{-1}(p)})^*$ la fonction qui associe à un mot représentant une exécution la séquence de valeurs des variables lue par p. Formellement, pour

$\alpha = a_1 s_1 \cdots a_i s_i$, $view_p^{VP}(\alpha) = (s_{i_1} \cdots s_{i_k})^{E^{-1}(p)}$ où la séquence $i_1 < \cdots < i_k$ vérifie :

$$\begin{aligned} i_1 &= \min\{0 \leq l < i \mid a_{l+1} \in \Sigma_r^p \cup \Sigma_m^p\} \\ i_k &= \max\{0 \leq l < i \mid a_{l+1} \in \Sigma_r^p \cup \Sigma_m^p\} \\ i_j &= \min\{l > i_{j-1} \mid a_{l+1} \in \Sigma_r^p \cup \Sigma_m^p\} \end{aligned}$$

Définition 2.44 (Stratégie à mémoire locale sur les variables)**.** *Une stratégie pour le processus $p \in \mathrm{Proc_S}$ $f^p : (\Sigma' \times S^V)^* \to \Sigma^p$ est* à mémoire locale sur les variables *si, pour tout $\alpha, \alpha' \in (\Sigma' \times S^V)^*$, si*

$$view_p^{VP}(\alpha) = view_p^{VP}(\alpha')$$

alors

$$f^p(\alpha) = f^p(\alpha')$$

Une stratégie distribuée $F = (f^p)_{p \in \mathrm{Proc}}$ est dite à mémoire locale sur les variables si f^p est à mémoire locale sur les variables pour tout $p \in \mathrm{Proc_S}$.

Exécutions équitables Dans le cas d'exécutions asynchrones, une exécution selon la stratégie peut ne comporter que des actions de l'environnement. On peut vouloir considérer de telles exécutions comme des cas dégénérés, et ne pas les inclure dans l'ensemble des exécutions devant satisfaire la spécification. Dans ce cas, le problème de contrôle est de déterminer l'existence d'une stratégie dont toutes les exécutions *équitables* satisfont la spécification. On peut définir plusieurs notions d'exécution équitable. En particulier,

Définition 2.45. *Une exécution F-compatible $\alpha = s_0 A_1 s_1 A_2 s_2 \cdots \in S^V \cdot (\Sigma' \cdot S^V)^\infty$ est*

- impartiale *si le système contrôlable a pu jouer une infinité de fois, i.e., si α est telle que $|\pi_{\{A \in \Sigma' \mid A \cap \Sigma_C \neq \emptyset\}}(\alpha)| = \omega$,*
- faiblement équitable *si le système contrôlable a pu jouer une infinité de fois si son programme était défini continûment, i.e., s'il existe i_0 tel que, pour tout $i \geq i_0$ il existe $p \in \mathrm{Proc_S}$ tel que $f^p(A_1 s_1 \cdots A_i s_i)$ est défini,*
- fortement équitable *si le système contrôlable a pu jouer une infinité de fois si son programme était défini infiniment souvent, i.e., si pour tout i il existe $j \geq i$ pour lequel il existe un processus $p \in \mathrm{Proc}$ tel que $f^p(A_1 s_1 \cdots A_j s_j)$ est défini.*

On note respectivement $\mathrm{Runs}_F^i(\mathcal{A})$, $\mathrm{Runs}_F^e(\mathcal{A})$ et $\mathrm{Runs}_F^E(\mathcal{A})$ l'ensembles des exécutions impartiales, faiblement équitables, fortement équitables, respectant la stratégie F .

Remarque 2.46. Les notions d'équité définies ci-dessus découlent d'une vision centralisée du système : ce qui nous intéresse, c'est si l'ensemble des processus vu comme une entité a pu jouer de façon équitable. On pourrait également définir des équités locales, tendant à s'assurer que chaque processus a pu jouer de façon équitable.

Remarque 2.47. La notion d'exécution impartiale n'est pertinente que lorsqu'on se restreint à des stratégies *totales*. Sinon le problème a une solution triviale avec une stratégie qui n'est jamais définie. Par la suite, lorsque l'on considérera des exécutions impartiales, il sera implicitement admis que l'on se restreint à des stratégies totales.

Définition 2.48 (Le problème de synthèse (respectivement de contrôle) général (respectivement impartial, faiblement équitable, fortement équitable) de systèmes distribués asynchrones (respectivement pseudo-synchrones), à communication par variables partagées, pour des architectures à lecture et écritures alternées (respectivement simultanées))**.** *Étant donnés*

- *une architecture* $\mathcal{A} = (\Sigma, V, E, (S^v)_{v\in V}, s_0, (\delta_a)_{a\in\Sigma})$ *à synthétiser (respectivement à contrôler), à lecture et écriture strictement alternées (respectivement simultanées),*
- *un ensemble* Proc *donnant une partition des variables de* V *et donc des actions, et* $\Sigma' = \Sigma$ *(respectivement* $\Sigma' = \bigcup_{P\in 2^{\text{Proc}}\setminus\emptyset}\{A \in 2^\Sigma \mid \forall p \in P, |A \cap \Sigma^p| = 1\}$*),*
- *une spécification totale sur les valeurs de variables,*

existe-t-il une stratégie distribuée et non-bloquante F *déterministe à mémoire locale sur les variables telle que* $Runs_F^{max}(\mathcal{A})$ *(respectivement* $Runs_F^i(\mathcal{A})$*,* $Runs_F^e(\mathcal{A})$*,* $Runs_F^E(\mathcal{A})$*)*$\subseteq$ *Spec ?*

Le cas asynchrone (par opposition au modèle pseudo-synchrone) avec variables partagées comme seul moyen de communication n'a été considéré que pour des systèmes centralisés, i.e., dans lequel $|\text{Proc}_S| = 1$.

Théorème 2.49 ([PR89b, WTD91])**.** *Le problème de synthèse impartial de systèmes centralisés asynchrones à communication par variables partagées, pour des architectures à lecture et écriture strictement alternées, est décidable et 2EXPTIME-complet pour les spécifications de* $\mathsf{LTL}(V)$*.*

On rappelle que dans une architecture à synthétiser, toutes les actions sont toujours possibles, donc un processus n'est jamais bloqué. Une stratégie non-bloquante sera donc toujours totale.

Le problème de synthèse équitable a été explicitement posé et résolu dans [AM94], puis dans [Var95] qui a présenté une méthode de résolution par automates :

Théorème 2.50 ([AM94, Var95])**.** *Le problème de synthèse impartial (respectivement faiblement équitable, fortement équitable) de systèmes centralisés asynchrones à communication par variables partagées, pour des architectures à lecture et écriture simultanées, est décidable pour des spécifications* ω*-régulières.*

De plus, lorsque la spécification est donnée par un automate de Büchi, [Var95] a montré que le problème est 2EXPTIME-complet.

En fait, ce mécanisme de communication se révèle très faible dans un système asynchrone : en effet, lorsqu'un processus écrit sur une variable à destination d'un autre processus, il n'a aucune garantie que l'information sera effectivement transmise. De même, quand un processus lit une variable d'un autre processus, il n'a aucun moyen de savoir s'il a perdu de l'information, et combien. Cette intuition est formalisée par le résultat suivant :

Théorème 2.51 ([FS06])**.** *Le problème de synthèse général de systèmes distribués pseudo-synchrones à communication par variables partagées, pour des architectures à lecture et écriture simultanées (et des stratégies totales) est décidable pour les spécifications du* μ*-calcul si et seulement si* $|\text{Proc}_S| = 1$*.*

2.3.2 Communication par synchronisation d'actions

Dans cette modélisation des communications, les processus évoluent de façon totalement asynchrone sur leurs variables, et, de temps en temps, décident de faire une action commune.

L'information se transmet donc à travers cette synchronisation des processus sur les actions. Les variables ne servent donc plus à communiquer, mais uniquement aux calculs locaux des processus. On peut donc toutes les regrouper en une seule variable privée au processus, et assimiler l'ensemble des processus à l'ensemble des variables : $\text{Proc} = V$.

Caractéristiques des architectures Une action $a \in \Sigma$ peut donc lire les états de l'ensemble de processus $E^{-1}(a)$ et modifier ceux de $E(a)$. On se place ici dans le cadre restreint dans lequel les actions ne dépendent que de l'état des processus qu'elles modifient, i.e., pour tout $a \in \Sigma$, $E(a) = E^{-1}(a)$. On note $\Sigma^p = \{a \in \Sigma \mid p \in E(a)\}$ l'ensemble des actions du processus $p \in \text{Proc}$. Par ailleurs, on fait abstraction des processus incontrôlables interagissant avec le système, et on considère que les actions incontrôlables sont locales : pour tout $a \in \Sigma^p \cap \Sigma_{NC}$, $E(a) = E^{-1}(a) = \{p\}$.

Exemple 2.52. La signature représentée sur la figure 2.7 est constituée de l'ensemble d'actions $\Sigma = \{a, b, c, d\}$ et de l'ensemble de registres (assimilés donc aux processus) $V = \text{Proc} = \{p_1, p_2, p_3, p_4\}$. L'action a est une action partagée par p_1, p_2 et p_3, c'est-à-dire que chaque fois qu'elle est exécutée, elle modifie les états de contrôle de ces trois processus, en fonction de leur état de départ. L'action b est une action locale au processus p_2, l'action c est partagée entre p_3 et p_4 et l'action d est locale au processus p_4. On peut l'enrichir en l'architecture distribuée très simple suivante : $S^{p_1} = \{a_1\}$, $S^{p_2} = \{a_2, b_2\}$, $S^{p_3} = \{a_3, c_3\}$, $S^{p_4} = \{c_4, d_4\}$ et

$$\begin{aligned} \delta_a(a_1, a_2, a_3) &= (a_1, b_2, c_3) \\ \delta_b(b_2) &= a_2 \\ \delta_c(c_3, c_4) &= (a_3, d_4) \\ \delta_d(d_4) &= (c_4) \end{aligned}$$

Une autre représentation classique de ce type de systèmes est le produit d'automates synchronisés. Les valeurs S^p pour $p \in \text{Proc}$ donnent l'ensemble des états de l'automate de p, et les actions partagées par plusieurs processus sont les transitions étiquetées par la même lettre dans plusieurs automates, et qui sont donc effectuées en même temps : dans notre exemple, cela correspond à des automates à un ou deux états, et la transition a n'est possible que si les trois processus concernés sont dans l'état a_i, $i = 1, 2, 3$, et elle change l'état de p_2 en b_2 et l'état de p_3 en c_3. Dans l'état b_2, la seule action possible pour le processus b_2 est l'action locale b.

On dit qu'une architecture est *bipartite* si le domaine de chaque processus p est partitionné en états contrôlables et états incontrôlables : pour tout $p \in \text{Proc}$, $S^p = S^p_C \uplus S^p_{UC}$. Les actions incontrôlables ne sont activables qu'à partir d'un état incontrôlable, et les actions contrôlables uniquement à partir des états contrôlables. De plus, chaque processus alterne entre ses états contrôlables et ses états incontrôlables. Formellement, pour tout $a \in \Sigma$, on définit $\delta_a : S^{E^{-1}(a)} \to S^{E(a)}$, fonction partielle, qui vérifie :

$$\begin{aligned} &\text{Pour tout } a \in \Sigma_{NC} \text{ tel que } E(a) = E^{-1}(a) = \{p\} \\ &\delta_a : S^p_{UC} \to S^p_C \\ &\text{Pour tout } a \in \Sigma_C \\ &\delta_a : S_C^{E^{-1}(a)} \to S_{UC}^{E(a)}. \end{aligned}$$

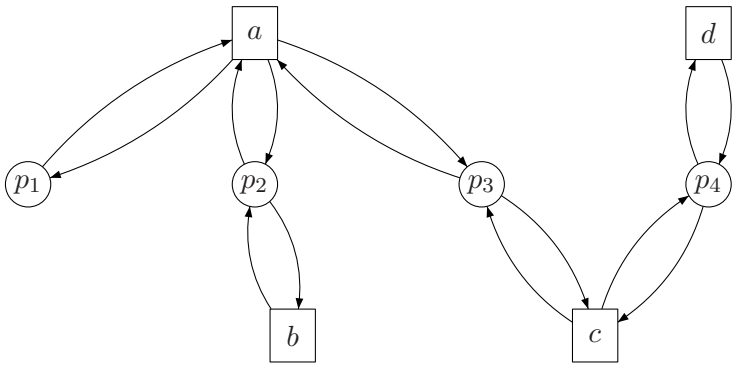


FIG. 2.7 – Un exemple de signature dans le cas asynchrone avec communication par synchronisation d'actions

Exécutions On se place ici dans un modèle dans lequel $\Sigma' = \bigcup_{a\in\Sigma}\{a\}$ et on va donc assimiler Σ' et Σ. Les exécutions sont alors des mots de $S^{\mathrm{Proc}} \cdot (\Sigma \cdot S^{\mathrm{Proc}})^\infty$, et correspondent aux exécutions d'un automate asynchrone de [Zie87].

Stratégies On peut choisir de distribuer les contrôleurs de plusieurs façons différentes : parmi les processus (variables), dans ce cas, on associe un contrôleur à chaque processus, qui décide des actions qu'il autorise à chaque instant, ou parmi les actions, et alors la décision d'autoriser une action ou pas est prise au niveau de l'action, en fonction des états de tous les processus concernés.

Un programme du système $F : (\Sigma \cdot S^{\mathrm{Proc}})^* \to 2^\Sigma$ est *distribué parmi les processus* s'il existe un ensemble de stratégies $(f^p)_{p\in\mathrm{Proc}}$, $f^p : (\Sigma \cdot S^{\mathrm{Proc}})^* \to 2^{\Sigma^p}$, pour $p \in \mathrm{Proc}$, telles que pour tout $\alpha \in (\Sigma \cdot S^{\mathrm{Proc}})^*$, $F(\alpha) = \{a \in \Sigma \mid a \in f^p(\alpha) \text{ pour tout } p \in E^{-1}(a) = E(a)\}$. La stratégie distribuée ne propose une action a que si tous les processus qui y participent la proposent. On distingue les stratégies distribuées à mémoire *locale* des stratégies à mémoire *causale*.

Pour $p \in \mathrm{Proc}$, on définit la *vue locale du processus* p, $\mathrm{vue}^l_p : (\Sigma \cdot S^{\mathrm{Proc}})^* \to (\Sigma^p \cdot S^p)^*$ par : pour tout $\alpha = a_1 s_1 \cdots a_i s_i \in (\Sigma \cdot S^{\mathrm{Proc}})^*$, $\mathrm{vue}^l_p(\alpha) = a_{i_1} s^p_{i_1} \cdots a_{i_k} s^p_{i_k}$ avec $i_1 < \cdots < i_k$ et $\{i_1, \cdots, i_k\} = \{1 \le j \le i \mid a_j \in \Sigma^p\}$.

Définition 2.53 (Stratégie distribuée parmi les processus, à mémoire locale)**.** *Pour tout* $p \in \mathrm{Proc}$*, la stratégie* $f^p : (\Sigma \cdot S^{\mathrm{Proc}})^* \to 2^{\Sigma^p}$ *est* à mémoire locale *si, pour tous* $\alpha, \alpha' \in (\Sigma \cdot S^{\mathrm{Proc}})^*$ *tels que*

$$vue^l_p(\alpha) = vue^l_p(\alpha'),$$

on a

$$f^p(\alpha) = f^p(\alpha').$$

On dit qu'une stratégie distribuée parmi les processus $F = (f^p)_{p\in\mathrm{Proc}}$ *est à mémoire locale si, pour tout* $p \in \mathrm{Proc}$*,* f^p *est à mémoire locale.*

Une stratégie à mémoire locale observe donc uniquement la séquence d'actions et d'états locaux correspondant aux instants où il a effectivement joué. En particulier, le contrôleur ne

peut savoir si les autres processus ont effectué des actions concurrentes entre deux de ses propres actions. Pour reprendre le parallèle avec les automates synchronisés, la vue locale du processus p correspond à l'exécution sur son propre automate.

Les contrôleurs peuvent avoir un autre type de mémoire, plus abstrait : une mémoire causale. De tels contrôleurs dépendent de toutes les actions ayant eu lieu dans leur passé causal, i.e., toutes les actions apparaissant dans leur passé dans chaque linéarisation de la trace correspondant à l'exécution en cours. On suppose ainsi qu'en se synchronisant sur les actions, les contrôleurs se transmettent les uns aux autres toute l'information qu'ils possèdent.

Formellement, on définit la *vue causale du processus* p par $\text{vue}_p^c : (\Sigma \cdot S^{\text{Proc}})^* \to \mathbb{M}(\Sigma, D)$ tel que pour α tel que $\pi_\Sigma(\alpha) = a_1 \cdots a_n \in \Sigma^*$, $\text{vue}_p^c(\alpha)$ est la plus petite trace préfixe de $[\pi_\Sigma(\alpha)] = (\{1, \cdots n\}, \leq, \lambda)$, $t' = (X', \leq, \lambda)$ vérifiant, pour tout $1 \leq i \leq n$, si $a_i \in \Sigma_p$, $i \in X'$.

Remarque 2.54. Formellement, la vue causale d'un processus ne tient pas compte de la séquence d'états visités. On remarque cependant, que $\mathcal{A}$ étant déterministe, si le mot $\alpha \in \Sigma^*$ est bien dans $\mathcal{L}(\mathcal{A})$, il est possible de reconstruire, en procédant par récurrence, la séquence d'états locaux visités au cours de l'exécution, i.e., si on note $\text{vue}_p^c(\alpha) = (X, \leq, \lambda)$, il est possible d'étiqueter chaque événement $e \in X$ par une valeur supplémentaire, $\sigma(e) = s \in S^{E(\lambda(e))}$, vérifiant $s = \delta_{\lambda(e)}((\sigma(e'))_{e' \in \text{Pred}(e)}^{E^{-1}(\lambda(e))})$, où $\text{Pred}(e)$ est l'ensemble des événements $e' \in X$ tels que $e' \lessdot e$. Cette étiquette donne la valeur des états locaux qui ont été modifiés par l'action $\lambda(e)$. De plus, comme par définition, vue_p^c n'a qu'un seul événement maximal, on peut reconstruire l'état global atteint après avoir lu $\text{vue}_p^c(\alpha)$.

Définition 2.55 (Stratégie distribuée parmi les processus, à mémoire causale)**.** *On dit que la stratégie* $f^p : (\Sigma \cdot S^{\text{Proc}})^* \to 2^{\Sigma_p}$ *du processus* $p \in \text{Proc}$ *est* à mémoire causale *si, pour tous* $\alpha = (a_1 s_1 \cdots a_i s_i), \alpha' = (a'_1 s'_1 \cdots a'_i s'_i) \in (\Sigma \cdot S^{\text{Proc}})^*$ *tels que*

$$vue_p^c(\alpha) = vue_p^c(\alpha'),$$

on a

$$f^p(\alpha) = f^p(\alpha').$$

Une stratégie distribuée parmi les processus $F = (f^p)_{p \in \text{Proc}}$ *est à mémoire causale si pour tout* $p \in \text{Proc}$, f^p *est à mémoire causale.*

Pour que la stratégie distribuée vérifie bien les conditions (2.2) et (2.3), les stratégies locales aux processus doivent vérifier les conditions exprimées ci-après. On appelle, pour tout $p \in \text{Proc}$, tout $s \in S^p$, $en_p(s) = \{a \in \Sigma^p \mid \text{ il existe } \overline{s} \in S^{\text{Proc}}, \overline{s}^p = s \text{ et } a \in en(\overline{s})\}$ l'ensemble des actions localement activables par le processus p. Alors, pour tout $p \in \text{Proc}$, pour tout $\alpha = (a_1 s_1 \cdots a_i s_i) \in (\Sigma' \cdot S^V)^\infty$, avec $s_i = s_0$ si $\alpha = \varepsilon$,

$$f^p(\alpha) \subseteq en_p(s_i^p) \tag{2.10}$$

$$f^p(\alpha) \cap \Sigma_{NC} = en_p(s_i^p) \cap \Sigma_{NC} \tag{2.11}$$

Remarque 2.56. Par la remarque 2.54, si $\alpha = a_1 s_1 \cdots a_i s_i, \alpha' = a'_1 s'_1 \cdots a'_i s'_i \in \text{Runs}(\mathcal{A})$ sont tels que $\text{vue}_p^c(\alpha) = \text{vue}_p^c(\alpha')$, alors $s_i^p = s_i'^p$.

On décrit maintenant formellement comment distribuer un contrôleur parmi les *actions* : une stratégie $F : (\Sigma \cdot S^{\text{Proc}})^* \to 2^\Sigma$ est distribuée parmi les actions si elle est formée par un

tuple $F = (f^a)_{a\in\Sigma}$ tel que pour tout $a \in \Sigma$, $f^a : (\Sigma \cdot S^{\text{Proc}})^* \to \{\top, \bot\}$. On définit, pour tout $\alpha \in (\Sigma \cdot S^{\text{Proc}})^*$, $F(\alpha) = \{a \in \Sigma \mid f^a(\alpha) = \top\}$.

On cherche également des stratégies à mémoire causale. On définit la *vue d'une action* $a \in \Sigma$, $\text{vue}^c_a : (\Sigma \cdot S^{\text{Proc}})^* \to \mathbb{M}(\Sigma, D)$, de la façon suivante : pour tout $\alpha = a_1 s_1 \cdots a_n s_n \in (\Sigma \cdot S^{\text{Proc}})^*$, $\text{vue}^c_a(\alpha)$ est la plus petite trace préfixe $t' = (X', \leq, \lambda)$ de $[\pi_\Sigma(\alpha)] = (\{1, \cdots, n\}, \leq, \lambda)$ telle que pour tout $1 \leq i \leq n$, pour tout $p \in E(a)$, si $a_i \in \Sigma^p$, $i \in X'$.

Remarque 2.57. De la même manière que pour la vue causale des processus, pour $\alpha \in \text{Runs}(\mathcal{A})$, à partir d'une trace $\text{vue}^c_a(\alpha)$, on peut reconstituer les états locaux parcourus au cours de l'exécution de $\mathcal{A}$.

Définition 2.58 (Stratégie distribuée parmi les actions, à mémoire causale)**.** *Une stratégie* $f^a : (\Sigma \cdot S^{\text{Proc}})^* \to \{\top, \bot\}$ *pour l'action* $a \in \Sigma$ *est* à mémoire causale *si, pour tout* $\alpha = (a_1 s_1 \cdots a_i s_i)$, $\alpha' = (a'_1 s'_1 \cdots a'_i s'_i) \in (\Sigma \cdot S^{\text{Proc}})^i$ *tels que*

$$vue^c_a(\alpha) = vue^c_a(\alpha'),$$

on a

$$f^a(\alpha) = f^a(\alpha').$$

Une stratégie distribuée parmi les actions $F = (f^a)_{a\in\Sigma}$ *est à mémoire causale si pour tout* $a \in \Sigma$, f^a *est à mémoire causale.*

Pour que la stratégie distribuée vérifie bien les conditions (2.2) et (2.3), les stratégies locales aux actions doivent vérifier, pour tout $a \in \Sigma$, pour tout $\alpha = (a_1 s_1 \cdots a_i s_i) \in \text{Runs}(\mathcal{A})$, avec $s_i = s_0$ si $\alpha = \varepsilon$,

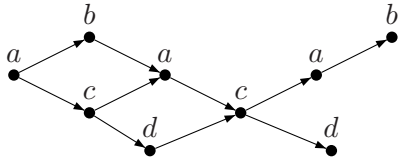
$$f^a(\alpha) = \top \text{ implique que } a \in en(s_i) \tag{2.12}$$

$$\text{si } a \in \Sigma_{NC}\ f^a(\alpha) = \top \text{ si et seulement si } a \in en(s_i) \tag{2.13}$$

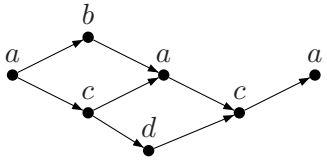
Exemple 2.59. On va illustrer les différences entre les trois vues possibles que l'on vient de décrire en reprenant l'architecture décrite dans l'exemple 2.52. Dans cet exemple, la relation de dépendance est donnée par $D = \{(a,b),(b,a),(a,c),(c,a),(c,d),(d,c)\}$. Considérons l'exécution $\pi_\Sigma(\alpha) = abcadcadb$. La trace correspondante est représentée sur la figure 2.8(a). La vue locale du processus p_1 est alors $\text{vue}^l_{p_1}(\alpha) = aaa$, tandis que sa vue causale est représentée sur la figure 2.8(b). La vue causale de l'action a (voir figure 2.8(c)) contient en plus la dernière action b jouée. En effet, le processus p_2 participe à l'action a, donc le contrôleur de l'action a connaît également la dernière action jouée sur p_2, et tout son passé.

Spécifications Afin de simplifier la présentation, on supposera par la suite que les systèmes sont non-bloquants. On va également se restreindre à des stratégies non-bloquantes, donc les exécutions compatibles avec une stratégie, et maximales, seront toutes infinies. Les spécifications seront données par un automate de Büchi $\mathcal{A}_\varphi$ sur les mots de Σ^ω.

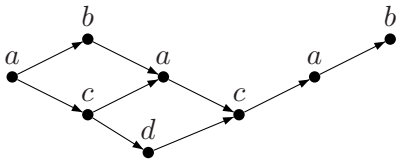
Le problème de synthèse de contrôleurs Soit $\mathcal{A} = (\Sigma, \text{Proc}, (S^p)_{p\in\text{Proc}}, s_0, (\delta_a)_{a\in\Sigma})$ une architecture distribuée. On dit que $\mathcal{A}$ est *à synthétiser* si, pour tout $p \in \text{Proc}$, $|S^p| = 1$, et pour tout $s \in S$, $en(s) = \Sigma$. On peut maintenant définir différentes variantes du problème de synthèse de contrôleurs pour des systèmes distribués en sémantique asynchrone, avec des communications par synchronisation d'actions :



(a) La trace



(b) La vue causale du processus p_1



(c) La vue causale de l'action a

FIG. 2.8 – La trace $[\pi_\Sigma(\alpha)]$ et deux vues causales

Définition 2.60 (Le problème de contrôle (respectivement de synthèse) de systèmes distribués asynchrones avec stratégies à mémoire locale distribuées parmi les processus (respectivement à mémoire causale distribuées parmi les processus, à mémoire causale distribuées parmi les actions)**.** *Étant donnés*

- *une architecture* $\mathcal{A} = (\Sigma, V, E, (S^v)_{v\in V}, s_0, (\delta_a)_{a\in\Sigma})$ *à contrôler (respectivement, à synthétiser),*
- *un langage* ω-régulier $\mathcal{L}_\varphi \subseteq \Sigma^\omega$,

existe-t-il une stratégie F *non bloquante, à mémoire locale distribuée parmi les processus (respectivement à mémoire causale distribuée parmi les processus, à mémoire causale distribuée parmi les actions) telle que* $Runs_F^{max}(\mathcal{A}) \subseteq \mathcal{L}_\varphi$ *?*

2.3.3 Résultats de la littérature

On commence par remarquer que, dans les cas que nous avons décrits de stratégies à mémoire locale ou causale, $\mathrm{Runs}_F(\mathcal{A})$ est toujours clos par équivalence de trace. Par contre, le langage $\mathcal{L}_\varphi$ n'a *a priori* pas de raison de l'être. Il se peut donc que $\mathrm{Runs}_F^{max}(\mathcal{A}) \not\subseteq \mathcal{L}_\varphi$ uniquement parce que le langage de spécification discrimine deux linéarisations de la même trace.

Les stratégies à mémoire locale distribuées parmi les processus Soit $f^p : (\Sigma \cdot S^{\mathrm{Proc}})^* \to (2^{\Sigma^p})$ une stratégie à mémoire locale pour le processus $p \in \mathrm{Proc}$. Tout d'abord, on définit une notion de mémoire locale restreinte, utilisée par [MT02b]. Avec ce type de mémoire, les contrôleurs ne tiennent pas compte de la séquence d'états visités au cours de l'exécution, mais uniquement de l'état courant du processus. En effet, les spécifications ne portant que sur les séquences d'actions exécutées, le contrôleur ne tient compte que de ce qui a été joué localement, ainsi que de l'état courant du processus qu'il contrôle (ceci afin d'être à même de proposer des actions activables). On remarque que cette définition donne une

mémoire plus restreinte que la définition 2.53, car, bien que l'architecture soit déterministe, la séquence des états locaux visités dépend également de la séquence des actions jouées par les autres processus, que l'on ne peut pas reconstruire avec une simple mémoire locale.

Définition 2.61 (Stratégies à mémoire locale restreinte)**.** *On dit que* $f^p : (\Sigma \cdot S^{\text{Proc}}) \to 2^{\Sigma^p}$ *est une stratégie* à mémoire locale restreinte *si, pour tout* $\alpha = a_1 s_1 \cdots a_i s_i$ *et pour tout* $\alpha' = a'_1 s'_1 \cdots a'_i s'_i$ *tels que*

$$\pi_{\Sigma^p}(a_1 \cdots a_i) = \pi_{\Sigma^p}(a'_1 \cdots a'_i)$$

et

$$s_i^p = s_i'^p$$

alors

$$f^p(\alpha) = f^p(\alpha')$$

On définit maintenant deux restrictions sur les stratégies : la première concerne la mémoire des stratégies ; on se restreint aux stratégies qui ne dépendent *que de l'état local courant du processus*, et du nombre de fois que le processus a effectué une transition. Le contrôleur est donc complètement ignorant de l'histoire des actions qui a été effectuée.

Définition 2.62 (Stratégies à mémoire temporelle)**.** *Soit* $f^p : (\Sigma \cdot S^{\text{Proc}})^* \to 2^{\Sigma^p}$ *une stratégie pour le processus* $p \in \text{Proc}$*. On dit que* f^p *est* à mémoire temporelle *si, pour toutes séquences* $\alpha = a_1 s_1 \cdots a_i s_i$*,* $\alpha' = a'_1 s'_1 \cdots a'_i s'_i \in (\Sigma \cdot S^{\text{Proc}})^*$ *telles que*

$$s_i^p = s_i'^p$$

on a

$$f^p(\alpha) = f^p(\alpha').$$

On dit qu'une stratégie distribuée $F = (f^p)_{p \in \text{Proc}}$ *est à mémoire temporelle si pour tout* $p \in \text{Proc}$*,* f^p *est à mémoire temporelle.*

La seconde restriction concerne l'ensemble des actions conseillées par la stratégie à un instant donné. On demande que l'ensemble des actions proposées par la stratégie après avoir vu une histoire donnée soit des actions partagées avec les mêmes processus. La stratégie commence donc par décider avec quels processus elle veut communiquer, avant de proposer un choix d'actions à effectuer de façon synchronisée avec ces derniers :

Définition 2.63 (Stratégies à communication rigide)**.** *Soit* $f^p : (\Sigma \cdot S^{\text{Proc}})^* \to 2^{\Sigma^p}$ *une stratégie du processus* $p \in \text{Proc}$*. On dit que* f^p *est* à communication rigide *si pour toute séquence* $\sigma \in (\Sigma^p)^*$*, il existe un ensemble* $X \subseteq \text{Proc}$ *tel que pour toute séquence* $\alpha \in (\Sigma \cdot S^{\text{Proc}})^*$ *telle que*

$$\pi_{\Sigma^p}(\alpha) = \sigma$$

pour tout $a \in f^p(\alpha)$*, on a*

$$X = E(a).$$

Dans [MT02b], a été définie la restriction du problème de contrôle de systèmes distribués suivante, ne considérant que des architectures bipartites (définies au début de la section 2.3.2) :

Définition 2.64 (Le problème restreint de contrôle de systèmes distribués asynchrones avec stratégies à mémoire locale distribuées parmi les processus)**.** *Étant donnés*

– *une architecture* $\mathcal{A} = (\Sigma, V, E, (S^v)_{v\in V}, s_0, (\delta_a)_{a\in\Sigma})$ bipartite
– $\mathcal{L}_\varphi$, *un langage* ω*-régulier* clos par équivalence de traces
existe-t-il une stratégie distribuée F*, non-bloquante,* à mémoire temporelle *et* à communication rigide *telle que* $Runs_F^{max}(\mathcal{A}) \subseteq \mathcal{L}_\varphi$ *?*

Théorème 2.65 ([MT02b])**.** *Le problème restreint de contrôle de systèmes distribués asynchrones avec stratégies à mémoire locale distribuées parmi les processus est décidable. De plus, s'il existe une stratégie distribuée non-bloquante gagnante, on peut effectivement synthétiser une stratégie distribuée gagnante à états finis.*

On appelle $R1$, $R2$ et $R3$ les restrictions du problème de contrôle considérées : $R1$ est la restriction aux langages de spécification clos par équivalence de traces, $R2$ celle aux programmes à mémoire temporelle, et $R3$ celle aux programmes à communication rigide.

Si les restrictions $R3$ et surtout $R1$ peuvent sembler naturelles, la restriction $R2$ est très forte en ce qu'elle n'autorise presque pas de mémoire aux contrôleurs. Cependant, si l'on relâche une de ces trois restrictions, le problème devient indécidable :

Théorème 2.66 ([MT02b])**.** *Le problème restreint de contrôle de systèmes distribués asynchrones avec stratégies à mémoire locale distribuées parmi les processus dans lequel on a retiré une ou plusieurs des restrictions* R_1*,* R_2 *ou* R_3 *est indécidable.*

Les stratégies à mémoire causale En augmentant la mémoire des contrôleurs avec l'ensemble des informations dans son passé causal, on peut s'affranchir de ces restrictions. En particulier, à partir du moment où l'on se restreint à des spécifications closes par équivalence de traces, le problème de contrôle de systèmes distribués asynchrones avec stratégies à mémoire causale distribuée parmi les processus devient décidable pour une classe d'architectures dites *communiquant de façon connexe.*

On commence définir la notion de processus séparés au cours d'une exécution (définie dans [MTY05]) : on dit que deux processus sont séparés dans une exécution finie s'ils ne communiquent pas, même indirectement :

Définition 2.67 (Processus séparés)**.** *Soient* $p, q \in$ Proc*, et* $\alpha \in \Sigma^*$*. On dit que* p *et* q *sont* séparés dans α *si il existe* $\tau = t_1 \cdots t_n, \tau' = t'_1 \cdots t'_m \in \Sigma^*$ *tels que*
– $\alpha \sim \tau\tau'$
– *pour tout* $1 \leq i \leq n$*,* $1 \leq j \leq m$*,* $E(t_i) \cap E(t'_j) = \emptyset$*, i.e.,* $\tau \sim \tau'$*,*
– *pour tout* $1 \leq i \leq n$*,* $t_i \notin \Sigma_q$*, pour tout* $1 \leq j \leq m$*,* $t'_j \notin \Sigma_p$

Une architecture est k-communiquante si, quand deux processus cessent de communiquer au cours d'une exécution pendant au moins k transitions, ils perdent la possibilité de communiquer dans toutes les prolongations d'exécutions possibles (voir également [MTY05]).

Définition 2.68 (Architecture k-communiquante)**.** *Une architecture* $\mathcal{A}$ *est* k-communiquante *si et seulement si, pour tout* $\alpha \in \Sigma^*$ *préfixe d'un mot de* $\mathcal{L}(\mathcal{A})$*, pour tout* $p, q \in$ Proc*, s'il existe* $\alpha_1 \in \Sigma^*$ *tel que*
– $\alpha\alpha_1 \in \mathcal{L}(\mathcal{A})$
– $|\pi_{\Sigma_p}(\alpha_1)| \geq k$ *et* $|\pi_{\Sigma_q}(\alpha_1)| = 0$
alors p *et* q *sont séparés dans* α_2*, pour tout* $\alpha_2 \in \Sigma^*$ *tel que* $\alpha\alpha_1\alpha_2 \in \mathcal{L}(\mathcal{A})$*.*

À partir de ces notions, [MTY05] définissent :

Définition 2.69 (Architectures communiquant de façon connexe (CCP))**.** *Une architecture* $\mathcal{A} = (\Sigma, V, E, (S^v)_{v\in V}, s_o, (\delta_a)_{a\in\Sigma})$ *est un CCP si et seulement si il existe un entier* k *telle qu'elle est* k*-communiquante.*

Proposition 2.70 ([MTY05])**.** *Déterminer si une architecture est un CCP est décidable.*

Théorème 2.71 ([MTY05])**.** *Le problème de contrôle de systèmes distribués asynchrones avec stratégies à mémoire causale distribuées parmi les processus est décidable pour les spécifications* w*-régulières closes par équivalence de traces et les architectures communiquant de façon connexe.*

Les stratégies à mémoire causale distribuées parmi les actions sont encore plus puissantes. Le principal résultat de décidabilité pour ce formalisme peut être énoncé comme suit.[2] On définit tout d'abord la notion d'alphabet de dépendance de type *co-graphe*, formé par produit série et produit parallèle d'alphabets de dépendance. Le produit série de deux alphabets de dépendance est l'union des deux alphabets, dans laquelle toutes les actions de l'un sont dépendantes des actions de l'autre. Dans le produit parallèle par contre, toutes les actions de l'un sont indépendantes des actions de l'autre. Formellement :

Définition 2.72 (Produit série)**.** *Soient* (Σ_1, D_1) *et* (Σ_2, D_2) *deux alphabets de dépendance. Alors le* produit série *de* (Σ_1, D_1) *et* (Σ_2, D_2) *est l'alphabet de dépendance* $(\Sigma_1, D_1)\cdot(\Sigma_2, D_2) = (\Sigma_1 \uplus \Sigma_2, D_1 \cup D_2 \cup \Sigma_1 \times \Sigma_2 \cup \Sigma_2 \times \Sigma_1)$.

Définition 2.73 (Produit parallèle)**.** *Soient* (Σ_1, D_1) *et* (Σ_2, D_2) *deux alphabets de dépendance. Alors le* produit parallèle *de* (Σ_1, D_1) *et* (Σ_2, D_2) *est donné par l'alphabet de dépendance* $(\Sigma_1, D_1) \parallel (\Sigma_2, D_2) = (\Sigma_1 \uplus \Sigma_2, D_1 \cup D_2)$

Définition 2.74 (Alphabet co-graphe)**.** *Un alphabet de dépendance est un* alphabet co-graphe *s'il appartient à la plus petite classe d'alphabets de dépendance contenant les singletons, et fermée par produit série et produit parallèle.*

Théorème 2.75 ([GLZ04, Ler05])**.** *Le problème de contrôle de systèmes distribués asynchrones avec stratégies à mémoire causale distribuées parmi les actions est décidable pour les architectures dont l'alphabet de dépendance est un co-graphe et des spécifications* ω*-régulières closes par équivalence de traces.*

En fait, les stratégies à mémoire causale ont plus de puissance lorsqu'elles sont distribuées parmi les actions que parmi les processus. Cette différence réside dans la quantité d'information disponible au contrôleur pour prendre une décision. Dans le cas de contrôleurs attachés aux processus, ce dernier prend la décision de faire une action partagée avec un autre processus sans connaître l'état de cet autre processus. Par contre, un contrôleur attaché à une action a la possibilité de connaître les états courants de tous les processus participant à cette action avant de l'autoriser ou non. En particulier, [MWZ09] ont exhibé une architecture et une spécification pour laquelle il existait une stratégie gagnante distribuée parmi les actions, mais pas de stratégie distribuée parmi les processus.

Par ailleurs on peut réduire le contrôle de systèmes distribués avec stratégies distribuées parmi les processus à celui avec stratégies distribuées parmi les actions :

[2] Originellement le théorème 2.75 a été établi dans un modèle un peu plus général que celui présenté ici. En particulier, les architectures ne sont pas nécessairement déterministes. De plus les domaines de lecture et d'écriture des actions peuvent être distincts. Enfin, les stratégies à synthétiser ne sont pas nécessairement non-bloquantes. On a ici suivi le choix de présentation de [MWZ09] afin d'unifier les résultats.

Théorème 2.76 ([MWZ09]). *Pour toute architecture $\mathcal{A}$ et langage ω-régulier clos par équivalence de traces $\mathcal{L}_{Spec}$ on peut construire une architecture $\overline{\mathcal{A}}$ et un langage $\overline{\mathcal{L}_{Spec}}$ tels qu'il existe F, une stratégie distribuée parmi les processus gagnante pour $(\mathcal{A}, \mathcal{L}_{Spec})$ si et seulement si il existe $\overline{F}$ une stratégie distribuée parmi les actions gagnante pour $(\overline{\mathcal{A}}, \overline{\mathcal{L}_{Spec}})$.*

Malheureusement, cette réduction ne conserve pas les alphabets de dépendance, et en particulier ne conserve pas la propriété d'être un co-graphe. Elle ne peut donc servir à transférer les résultats de décidabilité de [GLZ04] au problème de contrôle avec stratégies distribuées parmi les processus.

Par contre,

Théorème 2.77 ([MWZ09]). *Le problème de contrôle de systèmes distribués asynchrone avec stratégies à mémoire causale distribuées parmi les actions est décidable pour les architectures communiquant de façon connexes, pour des spécifications ω-régulières closes par équivalence de traces.*

Chapitre 3

Synthèse de systèmes synchrones

Sommaire

Dans ce chapitre, on traite du problème de synthèse de système distribué synchrone. Plus particulièrement, on cherche à étudier sous quelles restrictions, les plus naturelles possibles, le problème devient décidable. En fait, l'indécidabilité est très vite atteinte lorque l'on s'autorise des spécifications *totales*. Il est vrai que, dans les cas positifs où l'on obtient de la décidabilité, comme dans le cas des architectures pipeline, considérer des spécifications totales renforce le résultat. Cependant, une telle hypothèse affaiblit les résultats d'indécidabilité. En effet, les spécifications totales permettent de supprimer des liens de communication existant dans l'architecture (en imposant qu'une variable de communication entre deux processus prenne toujours la même valeur par exemple) et donc de se réduire facilement à l'architecture indécidable de Pnueli-Rosner (voir figure 2.4 page 32).

Par ailleurs, les spécifications *externes* sont très naturelles d'un point de vue pratique : lorsque l'on définit une spécification, on s'intéresse surtout au comportement visible (de type entrée-sortie) du système. La façon dont les processus communiquent de façon interne afin de satisfaire cette spécification devrait être laissée totalement libre.

Pour toutes ces raisons, on prétend qu'établir un critère nécessaire et suffisant de décidabilité pour le problème de contrôle de systèmes distribués synchrones est plus intéressant et plus utile (bien que plus difficile) lorsqu'on se restreint aux spécifications externes, par rapport aux spécifications totales. En effet, on sait déjà qu'avec des spécifications externes, l'architecture représentée sur la figure 2.6 page 36 est décidable, et on va montrer dans ce chapitre que l'architecture dessinée figure 2.3 page 29 devient décidable, alors que ce n'est pas une

architecture ordonnée au sens de [FS05] et donc qu'elle est indécidable pour des spécifications totales.

Pour exposer les résultats et les démonstrations de ce chapitre, il sera plus agréable d'utiliser un formalisme légèrement différent de celui utilisé dans le chapitre 2, en faisant abstraction de la notion d'action, pour se concentrer sur celle de variable. On commence donc par présenter les notations que nous allons utiliser, ainsi que par montrer formellement que le problème considéré est bien équivalent au problème de synthèse de système distribué synchrone présenté dans la section 2.2. Ainsi, on situe clairement le modèle choisi par rapport aux autres résultats de la littérature. On définit ensuite trois grandes classes d'architectures : les architectures *à information incomparable*, dont on montre qu'elles sont toutes indécidables pour les spécifications externes, sont présentées section 2. Puis on introduit les architectures *uniformément bien connectées* pour lesquelles on montre que si elles ne sont pas à information incomparable, elles sont décidables pour des spécifications externes de CTL^*. Formulé autrement, on montre donc que si le fait d'être à information incomparable est une condition *suffisante* pour obtenir l'indécidabilité du problème de synthèse de système distribué avec spécifications externes, dès lors que l'on se restreint aux architectures uniformément bien connectées, cette condition devient également *nécessaire*. La complexité, et même la décidabilité d'une telle propriété sur une architecture donnée n'étant pas triviale, on consacre une partie de ce chapitre à l'analyse de cette complexité. Enfin on présente la classe plus étendue des architectures *bien connectées* pour lesquelles on montre qu'être à information incomparable n'est plus une condition suffisante d'indécidabilité (voir figure 3.12 pour un récapitulatif).

1 Le modèle

On s'intéresse au problème de synthèse de système distribué synchrone avec spécifications externes tel qu'exposé dans la définition 2.29 page 31. Comme annoncé, on étend par ailleurs les délais des processus à des valeurs arbitraires : pour tout $p \in \mathrm{Proc_S}$, $d_p \in \mathbb{N}$. De plus, quand l'architecture considérée est à synthétiser, les registres v_p de chaque processus $p \in \mathrm{Proc}$ et l'alphabet précis d'actions Σ sont inutiles. On va donc utiliser par la suite un formalisme dans lequel on fait abstraction des registres et des actions, dans lequel la notion de délai variable s'exprime facilement, et qui sera également plus pratique pour les démonstrations.

On montre que, dans le cas d'architectures 0-délai et 1-délai, le problème qu'on décrit est équivalent au modèle précédent.

Remarque 3.1. Le modèle général présenté au chapitre 2 peut être légérement modifié afin de capturer la notion de délai arbitraire : lorsqu'un processus a un délai supérieur à 1, les actions qu'il joue s'appliquent sur des états locaux plus anciens que l'état courant, ou l'état précédent. Il faudrait donc définir une transition sur des séquences d'états globaux suffisamment longues pour que la valeur sur laquelle l'action va s'appliquer apparaisse dans la séquence. Pour modéliser un délai 2 par exemple, il faudrait considérer un système de transition dont les états globaux sont des mots de $(S^V)^2$.

Architectures. Les architectures que nous allons considérer sont maintenant de la forme

$$\mathcal{A} = (\mathrm{Proc}, V, E, (S^v)_{v\in V}, s_0, (d_p)_{p\in \mathrm{Proc}})$$

dans lesquelles Proc est l'ensemble des processus, V est l'ensemble des variables, $E \subseteq (\mathrm{Proc} \times V) \cup (V \times \mathrm{Proc})$ indique quelles variables sont lues ou modifiées par quels processus, S^v est le

domaine de la variable $v \in V$, $s_0 \in S^V$ et $d_p \in \mathbb{N}$ le délai associé au processus $p \in \text{Proc}$. On ne représente plus explicitement non plus le processus environnement, ce qui a pour conséquence que certaines variables n'ont pas de prédécesseur ou de successeur dans le graphe formé par $(V \cup \text{Proc}, E)$.

Dans ce modèle, les notions de variables d'entrée et de sortie du système correspondent respectivement à

$$V_{\text{I}} = \{v \in V \mid E^{-1}(v) = \emptyset\}$$

et

$$V_{\text{O}} = \{v \in V \mid E(v) = \emptyset\},$$

et le fait qu'une variable n'est modifiée que par le processus à qui elle appartient s'écrit : pour tout $v \in V$, $|E^{-1}(v)| \leq 1$. On se restreint aux architectures acycliques, i.e., celles dont le graphe formé par les sommets $V \cup \text{Proc}$ et les arcs E est également acyclique. On remarque que la notion de graphe d'une architecture coïncide dans les deux définitions : celle présentée au chapitre précédent et celle que nous allons utiliser ici (voir la figure 3.1 pour un exemple de signature d'une architecture acyclique, et sa traduction dans le modèle utilisé dans ce chapitre). Par ailleurs, dans les figures de ce chapitre, on ne représente plus sur les graphes les variables d'entrée et de sortie du système comme ayant des arcs entrants et sortants : avec la définition de V_{I} et V_{O} que l'on prend, les variables d'entrée et de sortie sont simplement celles n'ayant respectivement aucun prédécesseur ou aucun successeur.

Exécutions. Une *exécution* de $\mathcal{A}$ est un mot $\sigma = s_0 s_1 \cdots \in (S^V)^\omega$ commençant par l'état initial de $\mathcal{A}$. C'est donc simplement une exécution de $TS_{\mathcal{A}}$ dans laquelle on ne tient pas compte de la séquence d'actions réellement effectuée. Comme de plus, dans le cas de la synthèse, toutes les transitions locales sont autorisées, tous les mots de $(S^V)^\omega$ sont virtuellement des exécutions possibles de $TS_{\mathcal{A}}$.

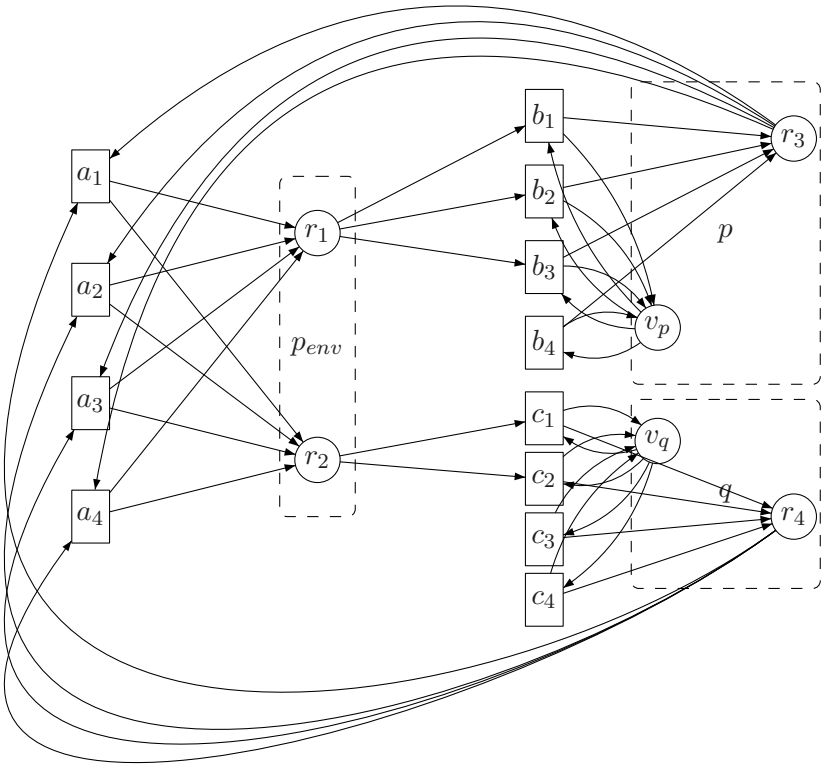
Arbres d'exécutions Un *arbre d'exécutions* de $\mathcal{A}$ est un arbre *complet* $t : (S^{V_{\text{I}}})^* \to S^V$ vérifiant $t(\varepsilon) = s_0$ et, pour tout $\rho \in (S^{V_{\text{I}}})^*$, tout $r \in S^{V_{\text{I}}}$, $t(\rho \cdot r)^{V_{\text{I}}} = r^{V_{\text{I}}}$. Pour $U \subseteq V$, la projection t^U d'un arbre d'exécutions t sur U est définie par $t^U(\rho) = t(\rho)^U$, pour tout $\rho \in (S^{V_{\text{I}}})^*$.

Programmes, stratégies On rappelle qu'une stratégie distribuée pour le problème donné par la définition 2.29 page 31 est un tuple $F = (f^p)_{p \in \text{Proc}}$ tel que $f^p : (\Sigma' \cdot S^V)^* \to \Sigma^p$ est à mémoire locale, pour tout $p \in \text{Proc}_{\text{S}}$, et tel que $f^{p_{env}} : (\Sigma' \cdot S^V)^* \to 2^{\Sigma_{NC}}$ vérifie, pour tout $\alpha \in (\Sigma' \cdot S^V)^*$, $f^{p_{env}}(\alpha) = \Sigma_{NC}$.

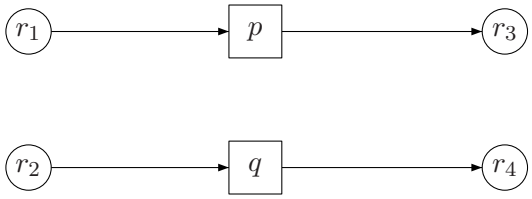
Une *stratégie distribuée pour* $\mathcal{A}$ est maintenant un tuple $F = (f^p)_{p \in \text{Proc}}$ tel que $f^p : (S^{E^{-1}(p)})^+ \to S^{E(p)}$. Au lieu de conseiller une action, la stratégie du processus p calcule maintenant son effet sur les variables de p et conseille donc un nouvel état local. Le calcul du nouvel état revient donc à la stratégie distribuée. C'est donc à présent elle qui doit tenir compte du délai à appliquer à l'action. D'où la définition suivante :

Définition 3.2 (Stratégie d-compatible)**.** *Soit* $\mathcal{A} = (\text{Proc}, V, E, (S^v)_{v \in V}, s_0, (d_p)_{p \in \text{Proc}}$ *une architecture,* $p \in \text{Proc}$ *un processus, et* $f^p : (S^{E^{-1}(p)})^+ \to S^{E(p)}$ *une stratégie pour le processus* p*. On dit que* f^p *est* compatible avec son délai *(ou* d-compatible*) si, pour tout* $i > 0$*, pour tous* $\sigma, \sigma' \in (S^{E^{-1}(p)})^i$ *tels que*

$$\sigma[i - d_p] = \sigma'[i - d_p],$$



(a) La signature d'une architecture acyclique à synthétiser $\mathcal{A}$



(b) Architecture $\mathcal{A}'$ correspondante

FIG. 3.1 – Une architecture acyclique

on a

$$f^p(\sigma) = f^p(\sigma').$$

On dit que $F = (f^p)_{p\in\text{Proc}}$ *est d-compatible si* f^p *est d-compatible, pour tout* $p \in \text{Proc}$.

Exécutions et arbres d'exécutions selon une stratégie distribuée Une *exécution selon la stratégie distribuée* F (ou *exécution* F*-compatible*) est un mot $\sigma = s_0 s_1 \cdots \in (S^V)^\omega$ tel que, pour tout $i > 0$, pour tout $p \in \text{Proc}$, $s_i^{E(p)} = f^p((\sigma[i])^{E^{-1}(p)})$.

Remarque 3.3. Nos architectures sont acycliques. Donc si on fixe $\rho \in (S^{V_\text{I}})^\omega$, il existe une unique exécution σ qui soit F-compatible telle que $\sigma^{V_\text{I}} = \rho$. On dit que σ est *l'exécution induite par* ρ.

Un *arbre d'exécutions selon la stratégie* F (ou *arbre* F*-compatible*) $t_F : (S^{V_\text{I}})^* \to S^V$ est un arbre d'exécutions de $\mathcal{A}$ tel que pour tout $\rho \in (S^{V_\text{I}})^*$, l'étiquette de la branche ρ est une exécution F-compatible. Autrement dit, tel que pour tout $\rho \in (S^{V_\text{I}})^*$, pour tout $p \in \text{Proc}$, $t_F(\rho)^{E(p)} = f^p((t(\rho[1])t(\rho[2])\cdots t(\rho))^{E^{-1}(p)})$.

Spécifications On rappelle qu'on s'intéresse à des spécifications externes sur les variables. Une spécification sera donnée par une formule $\varphi \in \mathcal{L}(V_\text{I} \cup V_\text{O})$, avec $\mathcal{L} = \{\mathsf{LTL}, \mathsf{CTL}, \mathsf{CTL}^*\}$ dans laquelle les propositions atomiques sont de la forme $(v = a)$ avec $v \in V_\text{I} \cup V_\text{O}$ et $a \in S^v$. La validité d'une exécution σ (respectivement d'un arbre d'exécutions t) sur une formule $\varphi \in \mathcal{L}(V_\text{I} \cup V_\text{O})$ ne dépend que de sa projection $\sigma^{V_\text{I}\cup V_\text{O}}$ (respectivement, $t^{V_\text{I}\cup V_\text{O}}$) (voir page 29).

On définit à présent la nouvelle version du problème de synthèse de système distribué synchrone :

Définition 3.4 (Le problème de synthèse de système distribué synchrone)**.** *Étant données une architecture* $\mathcal{A} = (V, \text{Proc}, E, (S^v)_{v\in V}, s_0, (d_p)_{p\in\text{Proc}})$ *et une spécification externe* $\varphi \in \mathcal{L}(V_I \cup V_O)$*, existe-t-il une stratégie distribuée* $F = (f^p)_{p\in\text{Proc}}$ *d-compatible telle que pour toute exécution* F*-compatible* σ*,* $\sigma \models \varphi$ *(telle que l'arbre* F*-compatible* $t_F \models \varphi$ *si* φ *est une formule branchante) ?*

Par la suite, on fera référence au problème de synthèse de système distribué avec spécifications externes de la définition 2.29 sous le terme de *problème de synthèse général* et au problème de synthèse défini ci-dessus sous le terme de *problème de synthèse avec actions abstraites.*

On montre que

Théorème 3.5. *Le problème de synthèse général est équivalent au problème de synthèse avec actions abstraites.*

Démonstration. Tout d'abord on met en relation de façon explicite les deux notions d'arbres d'exécutions présentées. Soit $\mathcal{A}$ un système distribué, entrée du problème de synthèse général. Soit F un programme du système $\mathcal{A}$, à mémoire locale, et $t : (\Sigma')^* \to S^V$ l'arbre d'exécutions selon F (voir la définition 2.20 page 23). En fait, chaque séquence de valeurs d'entrée du système peut être engendrée par une séquence d'actions de $\mathcal{A}$. On rappelle que $V_\text{I} = E(\Sigma_{NC}) = E(p_{env})$. Formellement, pour toute séquence de valeurs pour les entrées du système $\rho \in (S^{V_\text{I}})^*$, il existe une séquence d'actions $\alpha \in dom(t)$ telle que $(\pi_{S^V}(\text{Val}(t)(\alpha)))^{V_\text{I}} = \rho$. En effet, soit $\rho \in (S^{V_\text{I}})^*$ et $\alpha \in dom(t)$ tel que $\rho = (\pi_{S^V}(\text{Val}(t)(\alpha))^{V_\text{I}}$, et $r \in S^{V_\text{I}}$. Alors, par l'hypothèse d'environnement maximal, il existe $a \in \Sigma_{NC}$ tel que $\delta_a(t(\alpha)^{E^{-1}(p_{env})}) = r$. Comme

$f^{p_{env}}(\alpha) = \Sigma_{NC}$, il existe $A \in F(\alpha)$ tel que $A(p_{env}) = a$ et $(t(\alpha), A, t(\alpha \cdot A)) \in \Rightarrow$. Donc $t(\alpha{\cdot}A)^{V_\mathrm{I}} = \delta_{A(p_{env})}(t(\alpha)^{V_\mathrm{O}})) = r$ et donc $(\pi_{S^V}(\mathrm{Val}(t)(\alpha{\cdot}A))^{V_\mathrm{I}} = (t(\alpha{\cdot}A[1]\cdots t(\alpha{\cdot}A))^{V_\mathrm{I}} = \rho{\cdot}r$.

On peut alors définir $\Phi_t : \Sigma'^* \to (S^{V_\mathrm{I}})^*$ fonction surjective telle que, pour tout $\alpha \in dom(t)$,

$$\Phi_t(\alpha) = (\pi_{S^V}(\mathrm{Val}(t)(\alpha)))^{V_\mathrm{I}}.$$

Comme Φ_t est croissante, on peut étendre son domaine de définition aux mots de $(\Sigma')^\omega$ en posant, pour $\alpha \in (\Sigma')^\omega$ branche de t, $\Phi_t(\alpha) = \Phi_t(\bigsqcup_{\alpha' \sqsubseteq \alpha} \alpha') = \bigsqcup_{\alpha' \sqsubseteq \alpha} \Phi_t(\alpha')$. La fonction Φ_t ainsi étendue aux mots infinis est aussi surjective. En fait, cette fonction associe à une séquence d'actions donnée son effet sur les valeur des variables d'entrée du système.

Remarque 3.6. Comme l'architecture $\mathcal{A}$ est déterministe, et que F est à mémoire locale, la définition 2.25 page 28 implique que, si deux séquences d'actions induisent les mêmes séquences de valeurs sur les variables d'entrée, alors les variables du système vont prendre les mêmes valeurs. En effet, la décision de chaque stratégie ne dépend que de la valeur des variables que le processus associé peut lire, et l'effet combiné des actions choisies par les processus est prévisible. Formellement, pour tout $\alpha, \alpha' \in \Sigma'^*$, tels que

$$(\pi_{S^V}(\mathrm{Val}(t)(\alpha)))^{V_\mathrm{I}} = (\pi_{S^V}(\mathrm{Val}(t)(\alpha')))^{V_\mathrm{I}}$$

on a

$$t(\alpha) = t(\alpha').$$

Cette remarque nous permet donc de justifier le fait de représenter les arbres d'exécutions du système comme branchant sur les valeurs d'entrée et non sur les différentes actions possibles.

On pose $t' : (S^{V_\mathrm{I}})^* \to S^{V'}$ arbre *complet* défini par, pour tout $\rho \in (S^{V_\mathrm{I}})^*$,

$$t'(\rho) = t(\alpha)^{V'} \tag{3.1}$$

avec $\alpha \in \Phi_t^{-1}(\rho)$. Par la remarque 3.6, t' est bien défini.

Cet arbre t' contient moins de branches que l'arbre d'exécutions selon F dont il est issu, car il réunit en une même branche toutes les séquences d'actions ayant la même exécution sur $\mathcal{A}$. On montre à présent qu'il est suffisant de vérifier la spécification sur cette version restreinte des arbres d'exécutions.

Lemme 3.7. *Pour tout $t : \Sigma'^* \to S^V$ arbre d'exécutions selon F, pour tout $\varphi \in \mathsf{CTL}^*(V_I \cup V_O)$, pour tout $\alpha \in (\Sigma')^\omega$ branche de t, pour tout $i \geq 0$,*

$$t, \alpha, i \models \varphi \text{ si et seulement si } t', \Phi_t(\alpha), i \models \varphi.$$

Démonstration. Par récurrence sur la structure de φ. Le cas $(v = a)$ pour $v \in V_\mathrm{I} \cup V_\mathrm{O}$ et $a \in S^v$ découle de la définition de $t'(\Phi_t(\alpha[i])) = t(\alpha[i])^{V'}$.

Les cas de formules formées par opérations booléennes sont triviaux.

Si $t, \alpha, i \models \mathsf{A}\varphi$ alors pour tout $\alpha' \in (\Sigma')^\omega$ tel que $\alpha[i]\alpha'$ est une branche de t, on a $t, \alpha[i] \cdot \alpha', i \models \varphi$. De plus, pour tout $\alpha' \in \Sigma'^\omega$ tel que $\alpha[i]\alpha'$ est une branche de t, l'hypothèse de récurrence assure que $t', \Phi_t(\alpha)[i]\Phi_t(\alpha'), i \models \varphi$. En effet, $\Phi_t(\alpha[i] \cdot \alpha') = \Phi_t(\alpha)[i] \cdot \Phi_t(\alpha')$. Alors, par surjectivité de Φ_t, pour tout $\rho' \in (S^{V_\mathrm{I}})^\omega$, il est vrai que $t', \Phi_t(\alpha)[i] \cdot \rho', i \models \varphi$, et donc $t', \Phi_t(\alpha), i \models \mathsf{A}\varphi$. Réciproquement, si $t', \Phi_t(\alpha), i \models \mathsf{A}\varphi$, alors $t', \Phi_t(\alpha)[i] \cdot \rho', i \models \varphi$ pour tout $\rho' \in (S^{V_\mathrm{I}})^\omega$. Donc, pour tout $\alpha' \in \Sigma'^\omega$ tel que $\alpha[i] \cdot \alpha'$ est une branche de t et

$\Phi_t(\alpha[i]\cdot\alpha') = \Phi_t(\alpha)[i]\cdot\rho'$, l'hypothèse de récurrence assure que $t, \alpha[i]\cdot\alpha', i \models \varphi$ et donc, pour tout $\alpha' \in \Sigma'^{\omega}$ tel que $\alpha[i]\cdot\alpha'$ est une branche de t, on a $t, \alpha[i]\cdot\alpha' \models \varphi$. D'où $t, \alpha, i \models \mathsf{A}\,\varphi$.

Par ailleurs, $t, \alpha, i \models \varphi\,\mathsf{U}\,\psi$, si et seulement s'il existe $j \geq i$ tel que $t, \alpha, j \models \psi$ et $t, \alpha, k \models \varphi$ pour tout $i \leq k < j$. Par hypothèse de récurrence, c'est équivalent à l'existence de $j \geq i$ tel que $t', \Phi_t(\alpha), j \models \psi$ et tel que $t', \Phi_t(\alpha), k \models \varphi$ pour tout $i \leq k < j$, ce qui est équivalent à $t', \Phi_t(\alpha), i \models \varphi\,\mathsf{U}\,\psi$.

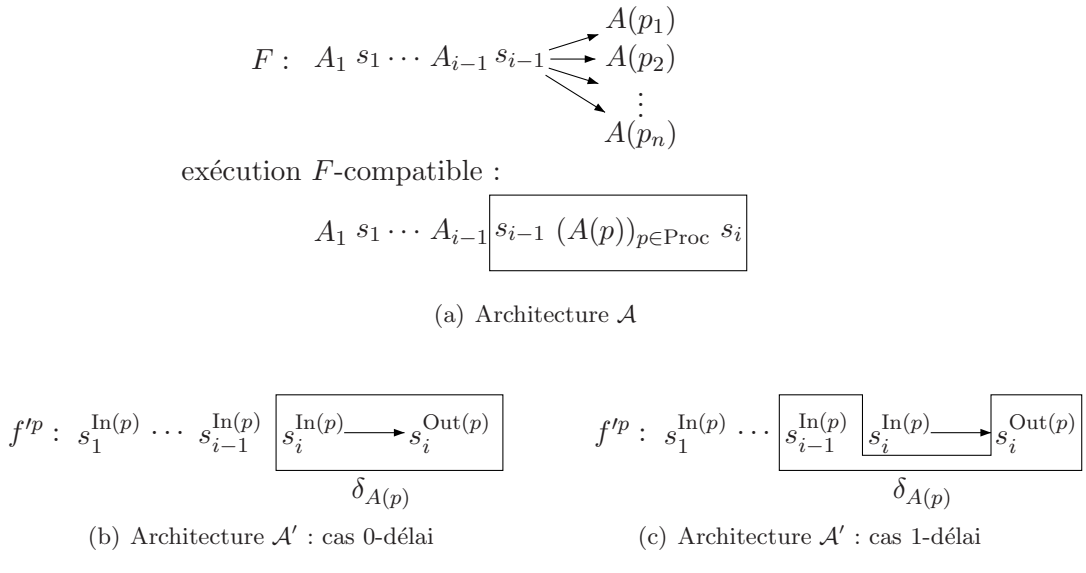
Les autres cas de formules de chemins sont également immédiats. □

On montre à présent comment construire une architecture du problème de synthèse avec actions abstraites à partir d'une architecture du problème de synthèse général. Soient $\mathcal{A} = (V = \uplus_{p\in\mathrm{Proc}} V_p, \Sigma, E, (S^v)_{v\in V}, s_0, (\delta_a)_{a\in\Sigma})$ une architecture à synthétiser acyclique, avec $(d_p)_{p\in\mathrm{Proc_S}}$ les délais associés, et $\varphi \in \mathsf{CTL}^*(V_\mathrm{I} \cup V_\mathrm{O})$ la spécification. (On ne considère que la logique CTL^* car les deux autres en sont des cas particuliers : pour montrer la réduction avec une formule $\psi \in \mathsf{LTL}(V_\mathrm{I} \cup V_\mathrm{O})$, on se ramène à une formule de CTL^* en considérant le problème pour la formule $\mathsf{A}\,\psi$). On rappelle que dans le cas général, les processus du système possèdent une variable particulière, v_p, représentant leur état de contrôle, et qu'ils peuvent lire et modifier. Or, dans une architecture à synthétiser, aucun programme n'est pré-établi, et on suppose que, pour tout $p \in \mathrm{Proc_S}$, $|S^{v_p}| = 1$. On considère, sans perte de généralité, que pour tout $p \in \mathrm{Proc_S}$, $S^{v_p} = \{0\}$. De plus, on suppose que, pour tout $v \in V$, $E^{-1}(v) \neq \emptyset$, i.e., que toute variable du système est lue par un processus. Dans le cas contraire, on supprime l'ensemble $\{v \in V \mid E^{-1}(v) = \emptyset\}$ de V'.

On construit $\mathcal{A}' = (V', \mathrm{Proc}', E', (S'^v)_{v\in V'}, s'_0, (d'_p)_{p\in\mathrm{Proc}'})$ de la façon suivante :

$$
\begin{array}{rcl}
V' & = & \mathrm{Com} \\
\mathrm{Proc}' & = & \mathrm{Proc_S} \\
S'^v & = & S^v \text{ pour tout } v \in V' \\
s'_0 & = & s_0^{V'} \\
d'_p & = & d_p \text{ pour tout } p \in \mathrm{Proc}' \\
E' & \subseteq & (V' \times \mathrm{Proc}') \cup (\mathrm{Proc}' \times V') \text{ défini par} \\
E' & = & \{(v,p) \mid \exists a \in \Sigma^p, (v,a) \in E\} \cup \\
 & & \{(p,v) \mid v \in V_p\}.
\end{array}
$$

On rappelle que $\mathrm{Com} = V \setminus \{v_p \mid p \in \mathrm{Proc_S}\}$. On a donc supprimé les registres v_p des variables du système car dans ce cas précis, ils n'apportent aucune information. De même, on fait abstraction du processus environnement, puisque le calcul de sa stratégie est trivial. Enfin, chaque processus pouvant utiliser des actions ayant toutes les sémantiques possibles, et les spécifications ne contraignant que les valeurs des variables, donc les *effets* des actions, on peut s'abstenir de mentionner explicitement ces dernières : seul compte le résultat sur les variables de la transition d'une action. Ainsi, comme on l'a déjà mentionné, les stratégies peuvent décider directement de la valeur qu'elles veulent donner à une variable, peu importe le nom de l'action utilisée pour obtenir cet effet. On redéfinit donc le graphe de l'architecture qui lie maintenant les variables et les processus. Comme on s'y attend, les variables modifiables par un processus sont ses propres variables, et celles qu'il peut lire sont celles pour lesquelles il existe une action qu'il contrôle qui en dépend. (On pourra à nouveau se référer à la figure 3.1 pour une illustration du passage de l'architecture $\mathcal{A}$ à l'architecture $\mathcal{A}'$).



(a) Architecture $\mathcal{A}$

(b) Architecture $\mathcal{A}'$: cas 0-délai

(c) Architecture $\mathcal{A}'$: cas 1-délai

FIG. 3.2 – Les stratégies et les exécutions induites dans $\mathcal{A}$ et $\mathcal{A}'$

Il est clair que ${V_\mathrm{I}}' = V_\mathrm{I}$, ${V_\mathrm{O}}' = V_\mathrm{O}$. La spécification φ reste donc inchangée au cours de la réduction. Par ailleurs, pour tout $p \in \mathrm{Proc}' = \mathrm{Proc_S}$, ${E'}^{-1}(p) = \mathrm{In}(p)$ et $E'(p) = \mathrm{Out}(p)$.

On rappelle que les variables de $\mathcal{A}$ diffèrent des variables de $\mathcal{A}'$ simplement par la présence ou l'absence des états de contrôle des processus du système. Pour passer simplement d'un état de $\mathcal{A}'$ à un état de $\mathcal{A}$, on utilisera la notation suivante : pour $s \in S^{\mathrm{In}(p)}$, on pose $\overline{s} \in S^{E^{-1}(p)}$ défini par $\overline{s}^v = s^v$ pour tout $v \in V'$ et $\overline{s}^{v_p} = 0$ pour tout $p \in \mathrm{Proc_S}$.

Lemme 3.8. *S'il existe une stratégie distribuée gagnante pour $(\mathcal{A}, \varphi)$ dans le problème de synthèse général, alors il existe une stratégie distribuée gagnante pour $(\mathcal{A}', \varphi)$ dans le problème de synthèse avec actions abstraites.*

Démonstration du lemme 3.8. Soit $F = (f^p)_{p\in\mathrm{Proc}}$ une stratégie distribuée gagnante pour $(\mathcal{A}, \varphi)$. On construit $F' = (f'^p)_{p\in\mathrm{Proc}'}$ stratégie distribuée gagnante pour $(\mathcal{A}', \varphi)$ de la façon suivante : soit $p \in \mathrm{Proc}'$, et soit $\sigma = s_1 \cdots s_i \in (S'^{E'^{-1}(p)})^+$. Alors $\sigma \in (S^{\mathrm{In}(p)})^+$. La stratégie f'^p va imiter le comportement de f^p après un préfixe d'exécution ayant engendré la même séquence d'états : on s'intéresse donc aux exécutions F-compatibles $\alpha \in \mathrm{Runs}_F(\mathcal{A})$ vérifiant

$$(\pi_{S^V}(\alpha))^{\mathrm{In}(p)} = s_1 \cdots s_{i-1}. \tag{3.2}$$

On remarque que la séquence d'actions choisie n'est pas confrontée à la dernière valeur s_i. En effet, dans le modèle initial, la valeur courante d'une variable est déterminée par l'action effectuée par le processus la modifiant au cours du même instant d'exécution. Le processus qui nous concerne n'a donc pas accès à cette valeur au moment où il détermine sa stratégie. Cependant, c'est l'action qui va être choisie par la stratégie du processus p qui va prendre en compte la valeur de s_i, puisque la valeur de la variable à modifier sera déterminée par la

transition de cette action à partir de s_i (voir la figure 3.2). On pose

$$f'^p(\sigma) = \begin{cases} (\delta_{f^p(\alpha)}(\overline{s_{i-d_p}}))^{\mathrm{Out}(p)} & \text{avec } \alpha \in \mathrm{Runs}_F(\mathcal{A}) \text{ vérifiant (3.2)} \\ (0)_{v\in\mathrm{Out}(p)} & \text{si aucune exécution } F\text{-compatible ne vérifie (3.2)} \end{cases}$$

Le choix d'une séquence d'actions α nous permet d'utiliser l'action $f^p(\alpha)$ choisie par la stratégie du processus p. On rappelle qu'on passe d'un état s_{i-1} à un état s_i par l'action $A \in \Sigma'$ au cours d'une exécution synchrone avec s_i vérifiant, pour tout $p \in \mathrm{Proc_S}$,

$$s_i^{E(p)} = \delta_{A(p)}(s_{i-1}^{v_p}, s_{i-d_p}^{\mathrm{In}(p)})$$

(voir l'égalité (2.6) page 26). Ici, $A(p) = f^p(\alpha)$, et on voit que la stratégie f'^p calcule exactement la valeur des variables en écriture en fonction de l'action choisie par f^p. Comme l'architecture $\mathcal{A}$ est à synthétiser, $|S^{v_p}| = 1$. On rappelle que $E^{-1}(p) = \mathrm{In}(p) \cup \{v_p\}$. Donc tous $\alpha, \alpha' \in \mathrm{Runs}_F(\mathcal{A})$ vérifiant (3.2) sont tels que $\pi_{S^V}(\alpha)^{E^{-1}(p)} = \pi_{S^V}(\alpha')^{E^{-1}(p)}$. Comme de plus, f^p est à mémoire locale, tous $\alpha, \alpha' \in \mathrm{Runs}_F(\mathcal{A})$ vérifiant (3.2) sont tels que $f^p(\alpha) = f^p(\alpha')$. Ceci assure que f'^p est bien défini. Par ailleurs, si on considère $\sigma = s_1 \cdots s_i, \sigma' = s'_1 \cdots s'_i \in (S^{\mathrm{In}(p)})^+$ tels que $\sigma[i - d_p] = \sigma'[i - d_p]$, alors par définition $f'^p(\sigma) = f'^p(\sigma')$ (on rappelle que $d_p \in \{0, 1\}$). La stratégie $F' = (f'^p)_{p\in\mathrm{Proc}'}$ est donc bien définie et d-compatible.

On montre à présent qu'elle est gagnante pour $(\mathcal{A}', \varphi)$. Soit $t : (\Sigma')^* \to S^V$ l'arbre des exécutions F-compatibles. Soit $t' : (S^{V_\mathrm{I}})^* \to S^{V'}$ défini par l'équation (3.1). On montre que t' est bien l'arbre d'exécutions selon F' : $t'(\varepsilon) = s_0^{V'} = s'_0$. Soit $\rho \in (S^{V_\mathrm{I}})^*$ et $r \in S^{V_\mathrm{I}}$. Soit $\alpha \cdot A \in dom(t)$ tel que $(\pi_{S^V}(\mathrm{Val}(t)(\alpha \cdot A)))^{V_\mathrm{I}} = \rho \cdot r$. Alors $t'(\rho \cdot r) = (t(\alpha \cdot A))^{V'} = s \in S^{V'}$. Donc, par définition, $A \in F(\mathrm{Val}(t)(\alpha))$, et $(t(\alpha), A, t(\alpha \cdot A)) \in \Rightarrow$, ce qui implique que, pour tout $p \in \mathrm{Proc_S} = \mathrm{Proc}'$,

$$s^{\mathrm{Out}(p)} = \begin{cases} (\delta_{A(p)}(\overline{t(\alpha)^{\mathrm{In}(p)}}))^{\mathrm{Out}(p)} & \text{si } d_p = 1 \\ (\delta_{A(p)}(\overline{t(\alpha \cdot A)^{\mathrm{In}(p)}}))^{\mathrm{Out}(p)} & \text{si } d_p = 0, \end{cases}$$

avec $A(p) = f^p(\mathrm{Val}(t)(\alpha))$. Or, $(\pi_{S^V}(\mathrm{Val}(t)(\alpha)))^{V'} = t'(\rho[1]) \cdots t'(\rho)$, donc par définition de f'^p,

$$f'^p((t'(\rho[1]) \cdots t'(\rho)t'(\rho \cdot r))^{\mathrm{In}(p)}) = \begin{cases} (\delta_{A(p)}(\overline{t(\alpha)^{\mathrm{In}(p)}}))^{\mathrm{Out}(p)} & \text{si } d_p = 1 \\ (\delta_{A(p)}(\overline{t(\alpha \cdot A)^{\mathrm{In}(p)}}))^{\mathrm{Out}(p)} & \text{si } d_p = 0, \end{cases}$$

et

$$t'(\rho \cdot r)^{\mathrm{Out}(p)} = f'^p((t'(\rho[1]) \cdots t'(\rho)t'(\rho \cdot r))^{\mathrm{In}(p)})$$

Par ailleurs, par construction, $t'(\rho \cdot r)^{V_\mathrm{I}} = t(\alpha \cdot A)^{V_\mathrm{I}} = r$, et donc t' est l'arbre des exécutions selon F'.

Comme F est une stratégie distribuée gagnante pour $(\mathcal{A}, \varphi)$, $t \models \varphi$, et par le lemme 3.7, $t' \models \varphi$. Donc, F' est une stratégie distribuée gagnante pour $(\mathcal{A}', \varphi)$.

□

Réciproquement, à partir d'une architecture du problème de synthèse avec actions abstraites, on peut construire une architecture du problème général de la façon suivante. Soit $\mathcal{A}' = (\mathrm{Proc}', V', E', (S'^v)_{v\in V}, s'_0, (d'_p)_{p\in\mathrm{Proc}})$ une architecture pour le problème de synthèse

avec actions abstraites, dans laquelle, pour tout $p \in \text{Proc}'$, $d'_p \in \{0,1\}$. On construit une architecture à synthétiser $\mathcal{A} = (V = \biguplus_{p\in\text{Proc}} V_p, \Sigma, E, (S^v)_{v\in V}, s_0, (\delta_a)_{a\in\Sigma})$ avec les délais $(d_p)_{p\in\text{Proc}_\text{S}}$ pour le problème de synthèse général de la façon suivante :

$$\begin{aligned}
\text{Proc} &= \text{Proc}' \cup \{p_{env}\} \\
V_p &= \{v \in V' \mid (p,v) \in E'\} \cup \{v_p\} \text{ pour } p \in \text{Proc}' \\
V_{p_{env}} &= {V_\text{I}}' \\
S^v &= S'^v \text{ pour } v \in V' \\
S^{v_p} &= \{0\} \text{ pour } p \in \text{Proc}_\text{S} \\
s_0^v &= s_0'^v \text{ pour } v \in V' \\
s_0^{v_p} &= 0 \text{ pour } p \in \text{Proc}' \\
d_p &= d'_p \text{ pour } p \in \text{Proc}' \\
\Sigma &\text{ un alphabet isomorphe à l'ensemble } \bigcup_{p\in\text{Proc}'} {S^{E'(p)}}^{S^{E'^{-1}(p)}} \cup S^{{V_\text{I}}'} \\
E &\subseteq (V \times \Sigma) \cup (\Sigma \times V).
\end{aligned}$$

Les processus de $\mathcal{A}$ sont les mêmes que ceux de $\mathcal{A}'$ à qui on a ajouté le processus environnement. Pour tout processus contrôlable du système, les variables de ce processus sont les variables que le processus peut modifier dans l'architecture $\mathcal{A}'$. Comme l'architecture $\mathcal{A}'$ a la contrainte qu'une variable ne peut être modifiée que par un unique processus, on obtient bien une partition des variables. De plus, on rajoute pour chaque processus de Proc_S son état de contrôle v_p, qui a un domaine de taille 1, puisqu'on construit une architecture à synthétiser. De même, on se donne un très large alphabet d'actions, puisqu'on veut que toutes les fonctions de transition soient disponibles dans le système. Soit

$$\delta : \Sigma \to \bigcup_{p\in\text{Proc}} {S^{E'(p)}}^{S^{E'^{-1}(p)}} \cup S^{{V_\text{I}}'}$$

une application bijective, qui associe à chaque action sa fonction de transition locale. Les actions contrôlables du système sont $\Sigma_C = \delta^{-1}(\bigcup_{p\in\text{Proc}} {S^{E'(p)}}^{S^{E'^{-1}(p)}})$, et les actions incontrôlables sont $\Sigma_{NC} = \delta^{-1}(S^{{V_\text{I}}'})$.

On définit à E de la façon suivante : pour tout $p \in \text{Proc}'$, $a \in \delta^{-1}({S^{E'(p)}}^{S^{E'^{-1}(p)}})$ action contrôlable du système,

$$\begin{aligned}
E^{-1}(a) &= E'^{-1}(p) \cup \{v_p\} \\
E(a) &= V_p
\end{aligned}$$

pour tout $a \in \delta^{-1}(S^{{V_\text{I}}'})$ action incontrôlable du système,

$$\begin{aligned}
E^{-1}(a) &= {V_\text{O}}' \\
E(a) &= {V_\text{I}}'
\end{aligned}$$

Pour tout $a \in \Sigma_C$, on pose à présent

$$\delta_a = \delta(a)$$

et pour tout $a \in \Sigma_{NC}$, pour tout $s \in S^{V_O{}'}$,

$$\delta_a(s) = \delta(a)$$

Par ailleurs, on remarque que, comme précédemment, $V_I = E(\Sigma_{NC}) = V_I{}'$ et $V_O = \{v \in V \mid E(v) = \Sigma_{NC}\} = V_O{}'$, et pour tout $p \in \mathrm{Proc}_S = \mathrm{Proc}'$, $E'^{-1}(p) = \mathrm{In}(p)$ et $E'(p) = \mathrm{Out}(p)$.

On utilisera à nouveau la notation : pour $s \in S^{\mathrm{In}(p)}$, on pose $\overline{s} \in S^{E^{-1}(p)}$ défini par $\overline{s}^v = s^v$ pour tout $v \in V'$ et $\overline{s}^{v_p} = 0$ pour tout $p \in \mathrm{Proc}_S$.

Lemme 3.9. *S'il existe une stratégie distribuée gagnante pour le problème de synthèse avec actions abstraites sur $(\mathcal{A}', \varphi)$ alors il existe une stratégie distribuée gagnante pour le problème de synthèse général sur $(\mathcal{A}, \varphi)$.*

Démonstration du lemme 3.9. Soit $F' = (f'^p)_{p \in \mathrm{Proc}'}$ une stratégie distribuée gagnante pour $(\mathcal{A}', \varphi)$. Soit $p \in \mathrm{Proc}_S$ et $\alpha = A_1 s_1 \cdots A_{i-1} s_{i-1} \in (\Sigma' \cdot S^V)^*$, et on va déterminer quelle action la stratégie f^p va proposer. Pour cela, on calcule en fait sa fonction de transition associée $\delta_\alpha : S^{E^{-1}(p)} \to S^{E(p)}$ à partir des décisions prises par f'^p. On rappelle que les stratégies dans l'architecture $\mathcal{A}'$ intègrent dans leur calcul l'effet de l'action à jouer en fonction de la valeur à laquelle elle va s'appliquer. Pour retrouver une stratégie de $\mathcal{A}$, on doit à nouveau « externaliser » la fonction de transition. Formellement, on pose

$$\begin{array}{lll} \delta_\alpha & : S^{E^{-1}(p)} & \to S^{E(p)} \\ & s & \mapsto \begin{cases} \overline{f'^p(\pi_{S^V}(\alpha)^{\mathrm{In}(p)} \cdot s^{\mathrm{In}(p)})} & \text{si } d_p = 0 \\ \overline{f'^p(\pi_{S^V}(\alpha[|\alpha|-1])^{\mathrm{In}(p)} \cdot s^{\mathrm{In}(p)} \cdot s')} & \text{si } d_p = 1, \text{ avec } s' \in S^{\mathrm{In}(p)} \text{ quelconque} \end{cases} \end{array}$$

Dans le cas où le processus p est à délai 0, après une histoire $\alpha = A_1 s_1 \cdots A_{i-1} s_{i-1}$ sur l'architecture $\mathcal{A}$, les processus choisissent un tuple d'actions $(A_i(p))_{p \in \mathrm{Proc}}$ et l'exécution F-compatible correspondante est $\alpha \cdot A_i s_i$ avec $s_i^{E(p)} = \delta_{A_i(p)}(s_i^{E^{-1}(p)})$. Dans l'architecture $\mathcal{A}'$, la stratégie f'^p associe à l'histoire $\pi_{S^V}(\alpha)^{\mathrm{In}(p)} \cdot s_i^{\mathrm{In}(p)}$ la valeur $s_i^{\mathrm{Out}(p)}$. Dans le cas où le délai est 1 par contre, la stratégie F des processus de $\mathcal{A}$ ne change pas, seule change l'exécution F-compatible associée : à présent elle est $\alpha \cdot A_i s_i$ avec $s_i^{E(p)} = \delta_{A(p)}(s_{i-1}^{E^{-1}(p)})$. Sur l'architecture $\mathcal{A}'$, la stratégie f'^p intégrant le calcul de la transition de l'action, est *d-compatible* : elle ne dépend pas de la dernière valeur lue. Elle associe donc à l'histoire $(s_1 \cdots s_{i-2})^{\mathrm{In}(p)} \cdot s_{i-1}^{\mathrm{In}(p)} \cdot s_i^{\mathrm{In}(p)} = \pi_{S^V}(\alpha[|\alpha|-1])^{\mathrm{In}(p)} \cdot s_{i-1}^{\mathrm{In}(p)} \cdot s_i^{\mathrm{In}(p)}$ la valeur $s_i^{\mathrm{Out}(p)}$ qui, comme dans l'architecture $\mathcal{A}$, ne dépend pas de $s_i^{\mathrm{In}(p)}$ (voir à nouveau la figure 3.2).

Comme $\mathcal{A}$ est une architecture à synthétiser, il existe $a \in \Sigma^p$ tel que $\delta_a = \delta_\alpha$, et on pose

$$f^p(\alpha) = a$$

et

$$f^{p_{env}}(\alpha) = \Sigma_{NC}.$$

Pour tout $p \in \mathrm{Proc}_S$, si $d_p = 1$ alors par d-compatibilité de f'^p (voir définition 3.2), pour tout $s', s'' \in S^{\mathrm{In}(p)}$, $f'^p(\pi_{S^V}(\alpha[|\alpha|-1])^{\mathrm{In}(p)} \cdot s_{i-1}^{\mathrm{In}(p)} \cdot s') = f'^p(\pi_{S^V}(\alpha[|\alpha|-2])^{\mathrm{In}(p)} \cdot s_{i-1}^{\mathrm{In}(p)} \cdot s'')$, donc f^p est bien définie. De plus, il est clair que f^p est à mémoire locale. Donc $F = (f^p)_{p \in \mathrm{Proc}}$ est bien une stratégie du système $\mathcal{A}$ à mémoire locale.

On montre maintenant que la stratégie F ainsi calculée engendre bien les mêmes exécutions que la stratégie F'. Soit $t : \Sigma'^* \to S^V$ l'arbre des exécutions selon F, et $t' : (S^{V_I})^* \to S^{V'}$

défini par l'équation (3.1). On montre une fois de plus que t' est l'arbre des exécutions selon F'. Tout d'abord, il est clair que $t'(\varepsilon) = t(\varepsilon)^{V'} = s'_0$. Soit maintenant $\rho \in (S^{V_\mathrm{I}})^*$, et $r \in S^{V_\mathrm{I}}$. Par définition, $t'(\rho \cdot r)^{V_\mathrm{I}} = t(\alpha \cdot A)^{V_\mathrm{I}}$ avec $\Phi_t(\alpha \cdot A) = \rho \cdot r$, donc $\pi_{S^V}(\mathrm{Val}(t)(\alpha \cdot A))^{V_\mathrm{I}} = \rho \cdot r$, et donc $t'(\rho \cdot r)^{V_\mathrm{I}} = r$. Comme t est un arbre d'exécutions selon F, alors $A \in F(\mathrm{Val}(t)(\alpha))$. Soit $p \in \mathrm{Proc_S} = \mathrm{Proc}'$. Alors $A(p) = f^p(\mathrm{Val}(t)(\alpha))$ et, en appliquant les définitions,

$$t(\alpha \cdot A)^{\mathrm{Out}(p)} = \begin{cases} \delta_{A(p)}(t(\alpha \cdot A)^{E^{-1}(p)})^{\mathrm{Out}(p)} & \text{si } d_p = 0 \\ \delta_{A(p)}(t(\alpha)^{E^{-1}(p)})^{\mathrm{Out}(p)} & \text{si } d_p = 1. \end{cases}$$

$$= \begin{cases} f'^p(\pi_{S^V}(\mathrm{Val}(t)(\alpha))^{\mathrm{In}(p)} \cdot t(\alpha \cdot A)^{\mathrm{In}(p)}) & \text{si } d_p = 0 \\ f'^p(\pi_{S^V}(\mathrm{Val}(t)(\alpha[|\alpha|-1]))^{\mathrm{In}(p)} \cdot t(\alpha)^{\mathrm{In}(p)} \cdot t(\alpha \cdot A)^{\mathrm{In}(p)}) & \text{si } d_p = 1. \end{cases}$$

Or, $\pi_{S^V}(\mathrm{Val}(t)(\alpha))^{V'} = (t(\alpha[1]) \cdots t(\alpha))^{V'} = t'(\rho[1]) \cdots t'(\rho)$, donc

$$t(\alpha \cdot A)^{\mathrm{Out}(p)} = f'^p((t'(\rho[1] \cdots t'(\rho))^{\mathrm{In}(p)} \cdot t'(\rho \cdot r)^{\mathrm{In}(p)}) \tag{3.3}$$

On remarque que, quel que soit le délai, l'étiquette de t correspond à la valeur calculée par la stratégie f'^p sur l'étiquette de toute la branche, mais la sémantique différente de f'^p dans les deux cas entraîne donc des résultats différents. Par ailleurs, on remarque que le paramètre de la fonction $\delta_{A(p)}$ se retrouve bien comme dernière valeur de la séquence d'entrée de f'^p dans le cas 0-délai, et comme avant-dernière valeur de cette séquence dans le cas 1-délai, ce qui correspond à la définition faite de δ_α au début de cette démonstration.

Comme $t'(\rho \cdot r) = t(\alpha \cdot A)^{V'}$ on conclut à l'aide de l'égalité (3.3) que t' est bien l'arbre des exécutions selon F'.

La stratégie distribuée F' est gagnante pour $(\mathcal{A}', \varphi)$ dans le problème de synthèse avec actions abstraites, donc $t' \models \varphi$, donc, par le lemme 3.7, $t \models \varphi$, et F est une stratégie distribuée gagnante pour $(\mathcal{A}, \varphi)$ dans le problème de synthèse général. □

On a donc montré qu'on peut construire une architecture $\mathcal{A}'$ du problème de synthèse avec actions abstraites à partir d'une architecture $\mathcal{A}$ du problème de synthèse général, et que toute stratégie distribuée gagnante pour le problème de synthèse général se traduit en une stratégie distribuée gagnante pour le problème de synthèse avec actions abstraites (lemme 3.8). Par ailleurs on a montré qu'on peut construire $\mathcal{A}$, une architecture à synthétiser pour le problème de synthèse général à partir de toute architecture $\mathcal{A}'$ du problème de synthèse avec actions abstraites, et qu'à nouveau toute stratégie distribuée gagnante pour $\mathcal{A}'$ se traduit en une stratégie distribuée gagnante pour $\mathcal{A}$ (lemme 3.9). Les hypothèses utilisées dans ces deux lemmes permettent en fait de démontrer que chacun des problèmes se réduit à l'autre, ce qui nous permet de conclure la preuve du théorème 3.5. □

Il s'agit donc du même problème que celui énoncé en section 2.2, mais présenté de façon plus concise : en effet, le déterminisme des actions de Σ combiné aux contraintes de la synthèse entraîne une démultiplication du nombre d'actions nécessaires à la modélisation.

Par la suite, on considérera de plus qu'une stratégie distribuée pour l'architecture $\mathcal{A} = (\mathrm{Proc}, V, E, (S^v)_{v \in V}, s_0, (d_p)_{p \in \mathrm{Proc}})$ l'est parmi les *variables* en remarquant que pour tout $p \in \mathrm{Proc}$, $f^p = (f^v)_{v \in E(p)}$ avec $f^v : (S^{R(v)})^+ \to S^v$ où $R = E^{-2}$. Comme annoncé, on autorisera également les processus à avoir des délais dans $\mathbb{N}$, et on notera souvent d_v pour d_p, $p \in E^{-1}(v)$. La notion de délai arbitraire se traduit de façon beaucoup plus agréable dans ce modèle, puisqu'elle est incluse dans la définition de d-compatibilité d'une stratégie (on reformule ici la définition 3.2 en parlant de stratégies de variables) :

Définition 3.10 (Stratégie d-compatible)**.** *On dit qu'une stratégie* $f^v : (S^{R(v)})^+ \to S^v$ *est* compatible avec son délai *(ou* d-compatible*) si, pour tout* $i > 0$*, pour tous* $\sigma, \sigma' \in (S^{R(v)})^i$ *tels que*

$$\sigma[i - d_v] = \sigma'[i - d_v]$$

on a

$$f^v(\sigma) = f^v(\sigma')$$

On dit que $F = (f^v)_{v \in V \setminus V_I}$ *est* d*-compatible si, pour tout* $v \in V \setminus V_I$*,* f^v *est* d*-compatible.*

On reformule, pour être complet, le problème de synthèse tel qu'on va l'étudier dans ce chapitre :

Définition 3.11 (Le problème de synthèse de système distribué synchrone à délais variables)**.** *Étant données une architecture* $\mathcal{A} = (\mathrm{Proc}, V, E, (S^v)_{v \in V}, s_0, (d_p)_{p \in \mathrm{Proc}})$ *et une spécification* $\varphi \in \mathcal{L}(V_I \cup V_O)$ *(où* $\mathcal{L} \in \{\mathsf{LTL}, \mathsf{CTL}, \mathsf{CTL}^*\}$*), existe-t-il une stratégie distribuée* $F = (f^v)_{v \in V \setminus V_I}$ *compatibles avec les délais telle que toute exécution* F*-compatible* $\sigma \models \varphi$ *(telle que l'arbre des exécutions selon* F*,* $t \models \varphi$ *si* φ *est une formule branchante) ?*

On dira qu'une telle stratégie distribuée est *une stratégie distribuée gagnante pour* $(\mathcal{A}, \varphi)$. Par la suite, lorsque l'on parlera du problème de synthèse de système distribué synchrone (ou problème de SSD synchrone), c'est à la variante précise définie ci-dessus (définition 3.11) que l'on fera référence.

On considérera que, pour tout $v \in V$, $|S^v| \geq 2$ et $\{0, 1\} \subseteq S^v$.

On utilisera par la suite également les notations et définitions suivantes :

Stratégies sans mémoire Pour $v \in V \setminus V_\mathrm{I}$, la stratégie f^v est dite *sans mémoire* si elle ne dépend pas du passé, c'est-à-dire, s'il existe une fonction $g : S^{R(v)} \to S^v$ telle que $f^v(s_1 \cdots s_i \cdots s_{i+d_v}) = g(s_i)$, avec $s_1 \cdots s_i \cdots s_{i+d_v} \in (S^{R(v)})^+$. Dans le cas où $d_v = 0$, cela correspond à la définition habituelle de stratégie sans mémoire.

Résumés Pour une variable $v \in V$, on définit sa vue $\mathrm{Vue}(v) = (E^{-2})^*(v) \cap V_\mathrm{I}$ comme étant l'ensemble des variables modifiées par l'environnement dont v peut dépendre. En effet, à stratégie distribuée fixée, la valeur d'une variable ne dépend que des choix non-déterministes de l'environnement sur les variables auxquelles le processus en question a indirectement accès : pour toute séquence $\sigma \in (S^V)^\omega$ exécution selon la stratégie F, pour tout $v \in V \setminus V_\mathrm{I}$, pour tout $i \geq 0$, s_i^v ne dépend que de $\sigma^{\mathrm{Vue}(v)}[i]$. Ceci nous amène à définir la fonction correspondant à la composition de toutes les stratégies locales utilisées pour calculer la valeur de la variable v. On appelle *résumé* la fonction $\hat{f}^v : (S^{\mathrm{Vue}(v)})^+ \to S^v$ vérifiant $\hat{f}^v(\sigma^{\mathrm{Vue}(v)}[i]) = s_i^v$. Avec cette notation, on peut caractériser les arbres d'exécutions selon une stratégie distribuée $F = (f^v)_{v \in V \setminus V_\mathrm{I}}$ par les arbres d'exécutions t tels que, pour tout $\rho \in (S^{V_\mathrm{I}})^+$, $t(\rho)^v = \hat{f}^v(\rho^{\mathrm{Vue}(v)})$, pour tout $v \in V \setminus V_\mathrm{I}$.

Délai cumulatif minimal Dans ce chapitre, on utilisera la notion de *délai cumulatif minimal* de transmission d'information de u à v. Il est défini par $d(u, u) = 0$, $d(u, v) = \infty$ si $v \notin (E^2)^+(u)$, c'est-à-dire s'il n'y a pas de chemin de u à v dans le graphe de communication de l'architecture, et, pour tout $u \neq v \in (E^2)^+(u)$,

$$d(u, v) = d_v + \min\{d(u, w) \mid w \in R(v) \text{ et } w \in (E^2)^*(u)\}.$$

d-compatibilité pour les résumés La compatibilité avec les délais de la stratégie distribuée $F = (f^v)_{v\in V\setminus V_I}$ s'étend aux résumés $\hat{F} = (\hat{f}^v)_{v\in V\setminus V_I}$. Formellement, on dit qu'une application $h : (S^{\mathrm{Vue}(v)})^+ \to S^v$ est d-compatible (ou compatible avec les délais $(d_v)_{v\in V\setminus V_I}$) si pour tout $\rho \in (S^{\mathrm{Vue}(v)})^i$, $h(\rho)$ ne dépend que des préfixes $(\rho^u[i - d(u, v)])_{u\in \mathrm{Vue}(v)}$.

2 Architectures à information incomparable

Comme expliqué dans l'introduction de ce chapitre, le critère de décidabilité de [FS05] ne peut pas s'appliquer lorsque l'on se restreint aux spécifications externes. En particulier, il établit des conditions trop fortes sur les architectures pour la décidabilité du problème.

Dans cette section, on définit une condition suffisante d'indécidabilité pour le problème de synthèse avec spécifications *externes*.

Définition 3.12 (Architecture à information incomparable)**.** *On dit qu'une architecture* $\mathcal{A} = (\mathrm{Proc}, V, E, (S^v)_{v\in V}, s_0, (d_p)_{p\in\mathrm{Proc}})$ *est* à information incomparable *s'il existe deux variables de sortie* $x, y \in V_O$ *telles que* $Vue(x)\setminus Vue(y) \neq \emptyset$ *et* $Vue(y)\setminus Vue(x) \neq \emptyset$*. Dans le cas contraire, on dit que l'architecture est* à information linéairement préordonnée*.*

Une architecture ordonnée telle que définie dans [FS05, BJ06] (voir définition 2.37) est nécessairement à information linéairement préordonnée, mais la réciproque n'est pas vraie (voir la figure 3.3 pour des exemples d'architectures ordonnées, à information incomparable, à information linéairement préordonnée).

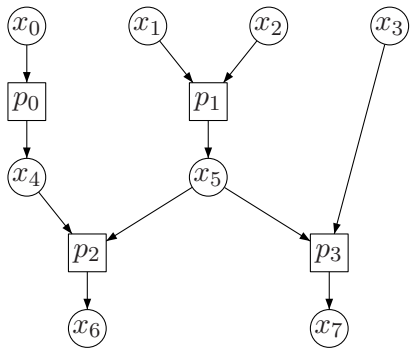
Il a été prouvé dans [PR90, FS05] que l'architecture $\mathcal{A}'$ reproduite figure 3.4 est indécidable pour les spécifications LTL et CTL dans le cas de processus 0-délai. On étend à présent ce résultat, en montrant que toute architecture à information incomparable est indécidable. Tout d'abord, il faut démontrer que le problème SSD synchrone est indécidable pour l'architecture $\mathcal{A}'$ dessinée figure 3.4 dans laquelle on considère des délais arbitraires pour p_1 et p_2. Il suffit simplement d'adapter la démonstration de [PR90] pour les cas LTL et CTL, afin que les spécifications tiennent compte des délais des processus.

Théorème 3.13. *Le problème de SSD synchrone est indécidable pour les architectures à délais arbitraires et les spécifications externes de* LTL *ou* CTL*.*

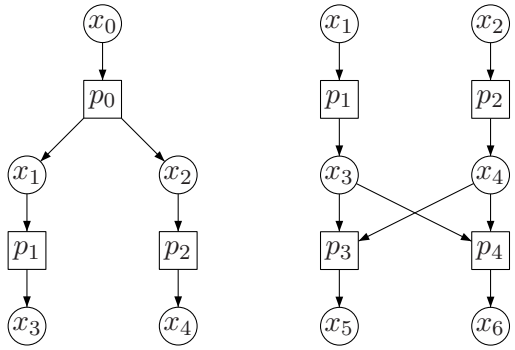
Démonstration. On démontre le résultat pour les spécifications LTL. Si l'idée est similaire à celle de [PR90], on présente ici une réduction légèrement différente de celle habituellement utilisée. Par ailleurs, cela fournit une démonstration entièrement formalisée de ce résultat, dont seules les idées de preuve figuraient dans [Ros92].

Pour une machine de Turing M, on définit une spécification LTL φ_M et on va réduire le problème du non-arrêt de M sur bande vide au problème de synthèse de système distribué synchrone pour $(\mathcal{A}', \varphi_M)$. Soit donc $\mathcal{A}' = (\mathrm{Proc}', V', E', (S'^v)_{v\in V'}, s'_0, (d_p)_{p\in\mathrm{Proc}'})$ l'architecture représentée sur la figure 3.4, dans laquelle $S'^{x_0} = S'^{y_0} = \{0, 1\}$, et $S'^{x_n} = S'^{y_m} = \Gamma \uplus Q \uplus \{\#\}$ avec $\#$ un symbole distinct de ceux de Γ et Q. Les alphabets Γ et Q sont respectivement les alphabets de bande et l'ensemble d'états de M. On considère que $s'^v_0 = 0$ pour $v \in \{x_0, y_0\}$ et $s'^v_0 = \#$ pour $v \in \{x_n, y_m\}$. On suppose que $d_{p_1} \leq d_{p_2}$.

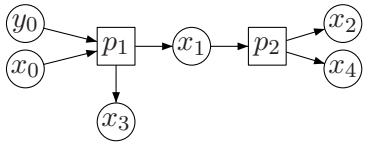
Une configuration de la machine de Turing définie par un état q et un contenu de bande $\gamma_1\gamma_2$, avec la tête de lecture située sur le premier symbole de γ_2 est représentée par le mot $\gamma_1 q\gamma_2 \in \Gamma^* Q\Gamma^+$ (on impose que γ_2 soit non vide pour des raisons techniques, si besoin on peut rajouter des symboles blancs). Une séquence de valeurs prises par la variable x_0, $\sigma^{x_0} \in$



(a) Une architecture à information incomparable



(b) Des architectures à information linéairement préordonnée mais non ordonnées



(c) Une architecture ordonnée

FIG. 3.3 – Exemples d'architectures à information incomparable ou linéairement préordonnée

$0^*1^p0\{0,1\}^\omega$, code l'entier $n(x_0) = p$. On prend le même codage pour la variable y_0. On construit donc une spécification LTL φ_M forçant toute stratégie distribuée gagnante à écrire sur la variable x_n la $n(x_0)$-ième configuration de M lorsqu'elle commence sur la bande vide.

La spécification φ_M est une conjonction des propriétés décrites ci-dessous.

1. Si la séquence de valeurs prises par x_0 (respectivement par y_0) est dans $0^q1^p0\{0,1\}^\omega$, alors la séquence des valeurs prises par x_n est dans $\#^{d_{p_1}+q+p}\Gamma^*Q\Gamma^+\#^\omega$ (respectivement y_m est dans $\#^{d_{p_2}+q+p}\Gamma^*Q\Gamma^+\#^\omega$). Ceci est formalisé par la formule $\alpha \in \mathsf{LTL}(V'_{\mathrm{I}} \cup V'_{\mathrm{O}})$:

$$\alpha = \alpha_{x_0,x_n,d_{p_1}} \wedge \alpha_{y_0,y_m,d_{p_2}}$$

où

$$\alpha_{z,t,d} \stackrel{\text{def}}{=} ((z=0) \wedge (t=\#)) \,\mathsf{W}\, (((z=1) \wedge (t=\#)) \,\mathsf{W}\, ((z=0) \wedge t \in \#^d\Gamma^*Q\Gamma^+\#^\omega))$$

où $t \in \#^d \Gamma^* Q \Gamma^+ \#^\omega$ signifie

$$(t = \#)\, \mathsf{U}_d\, ((t \in \Gamma)\, \mathsf{U}\, (t \in Q) \wedge \mathsf{X}((t \in \Gamma)\, \mathsf{U}\, ((t \in \Gamma) \wedge \mathsf{X}\, \mathsf{G}(t = \#))))$$

avec

$$\varphi\, \mathsf{U}_i\, \psi \stackrel{\text{def}}{=} \varphi \wedge (\mathsf{X}\, \varphi) \wedge \cdots \wedge (\mathsf{X}^{i-1}\, \varphi) \wedge X^i \psi$$

pour tout $i > 0$, et

$$\varphi\, \mathsf{U}_0\, \psi \stackrel{\text{def}}{=} \psi$$

2. Si la séquence de valeurs prises par x_0 est dans $0^q 10\{0,1\}^\omega$, i.e., $n(x_0) = 1$, alors la séquence de valeurs prises par x_n doit être $\#^{d_{p_1}+q+1} \mathcal{C}_1 \#^\omega$, i.e., elle doit correspondre à $\mathcal{C}_1$, la première configuration de M commençant sur bande vide. On décrit formellement β :

$$\beta \stackrel{\text{def}}{=} (x_0 = 0)\, \mathsf{W}\, ((x_0 = 1) \wedge \mathsf{X}((x_0 = 0) \rightarrow (x \in \#^{d_{p_1}} \mathcal{C}_1 \#^\omega)))$$

où $x \in \#^{d_{p_1}} \mathcal{C}_1 \#^\omega$ s'exprime simplement.

3. On dit que, dans une exécution $\sigma \in (S^V)^\omega$, les mots d'entrée sont *synchronisés* si, soit $\sigma^{x_0}, \sigma^{y_0} \in 0^q 1^p 0\{0,1\}^\omega$ (i.e., les séquences sur x_0 et y_0 codent le même entier, et de la même façon), soit $\sigma^{x_0} \in 0^q 1^{p+1} 0\{0,1\}^\omega$ et $\sigma^{y_0} \in 0^{q+1} 1^p 0\{0,1\}^\omega$ (i.e., $n(x_0) = n(y_0) + 1$, mais le codage « finit » au même moment). La formule γ exprime le fait que, si les mots d'entrée sont synchronisés, et que $n(x_0) = n(y_0)$, alors les séquences écrites sur x_n et y_m sont identiques, au décalage dû à leurs délais respectifs près. On commence par exprimer le fait que $n(x_0) = n(y_0)$:

$$(n(x_0) = n(y_0)) \stackrel{\text{def}}{=} (x_0 = y_0 = 0)\, \mathsf{U}\, ((x_0 = y_0 = 1) \wedge (x_0 = y_0 = 1)\, \mathsf{U}\, (x_0 = y_0 = 0))$$

La formule γ est définie par

$$\gamma \stackrel{\text{def}}{=} (n(x_0) = n(y_0)) \rightarrow \mathsf{G}(\mathrm{Egal}(x_n, y_m))$$

où $\mathrm{Egal}(x_n, y_m)$ exprime le fait que les valeurs de x_n et y_m sont égales, au décalage de leurs délais près :

$$\mathrm{Egal}(x, y) = \bigvee_{i \in \Gamma \uplus Q \uplus \{\#\}} ((x = i) \wedge \mathsf{X}^{d_{p_2} - d_{p_1}} (y = i))$$

4. Enfin, on exprime avec la formule δ que si x_0 et y_0 sont synchronisés, et que $n(x_0) = n(y_0) + 1$, alors on veut que la configuration codée sur x_n soit la configuration successeur dans l'exécution de la machine de Turing M de celle encodée sur y_m, encore une fois, en tenant compte du décalage dû à leurs délais respectifs. On utilise la formule $(n(x_0) = n(y_0) + 1)$ définie par

$$(x_0 = y_0 = 0)\, \mathsf{U}\, ((x_0 = 1) \wedge (y_0 = 0) \wedge \mathsf{X}((x_0 = y_0 = 1) \wedge (x_0 = y_0 = 1)\, \mathsf{U}\, (x_0 = y_0 = 0)))$$

La formule δ est donc

$$\delta = (n(x_0) = n(y_0) + 1) \rightarrow (\mathrm{Egal}(x_n, y_m)\, \mathsf{U}\, (\mathrm{Trans}(y_m, x_n) \wedge \mathsf{X}^3\, \mathsf{G}\, \mathrm{Egal}(x_n, y_m)))$$

où $\mathrm{Trans}(y,x)$ exprime le fait que le facteur de trois lettres de x est obtenu à partir de celui de y en effectuant une transition de la machine de Turing M. On a

$$\begin{aligned}\mathrm{Trans}(y,x) = &\bigvee_{(p,a,q,b,\leftarrow)\in T, c\in\Gamma} (\mathsf{X}^{d_{p_2}-d_{p_1}}(y = cpa)) \wedge (x = qcb)\\ \vee &\bigvee_{(p,a,q,b,\rightarrow)\in T, c\in\Gamma} (\mathsf{X}^{d_{p_2}-d_{p_1}}(y = pac)) \wedge (x = bqc)\\ \vee &\bigvee_{(p,a,q,b,\rightarrow)\in T} (\mathsf{X}^{d_{p_2}-d_{p_1}}(y = pa\#)) \wedge (x = bq\Box)\end{aligned}$$

On utilise l'abréviation $(x = abc)$ pour $(x = a) \wedge \mathsf{X}(x = b) \wedge \mathsf{X}^2(x = c)$. De plus, $\Box$ est le symbole vide de la bande et T est l'ensemble des transitions de M (la transition (p, a, q, b, dir), prise quand M est dans l'état p avec la tête de lecture sur le symbole a, change l'état de la machine en q, écrit sur la bande le symbole b et déplace la tête de lecture dans la direction $dir \in \{\leftarrow, \rightarrow\}$).

La spécification φ_M est donc $\varphi_M = \alpha \wedge \beta \wedge \gamma \wedge \delta$

Il est clair que la stratégie d-compatible qui écrit une séquence de # sur sa variable jusqu'à ce que la séquence 0^q1^p0 ait été jouée, puis écrit le codage de la p-ième configuration de la machine de Turing est une stratégie gagnante pour $(\mathcal{A}', \varphi_M)$.

On montre qu'il n'en existe pas d'autre : soit $F = (f^{x_n}, f^{y_m})$ une stratégie distribuée gagnante pour $(\mathcal{A}', \varphi_M)$. Soit $\sigma \in (S^V)^\omega$ une exécution F-compatible. Alors, on montre par récurrence sur $p > 0$, que

$$\forall p > 0, \forall q \geq 0, \sigma^{x_0} \in 0^q1^p0\{0,1\}^\omega \implies \sigma^{x_n} = \#^{d_{p_1}+q+p}\mathcal{C}_p\#^\omega \tag{3.4}$$

Pour $p = 1$ la propriété (3.4) découle directement de la spécification (en particulier, des formules α et β).

Soit $p > 1$. Supposons que $\sigma^{x_0} \in 0^q1^{p+1}0\{0,1\}^\omega$ et $\sigma^{y_0} \in 0^{q+1}1^p0\{0,1\}^\omega$. On considère alors une autre exécution F-compatible, σ_0, telle que $\sigma_0^{x_0} \in 0^{q+1}1^p0\{0,1\}^\omega$ et $\sigma_0^{y_0} = \sigma^{y_0} \in 0^{q+1}1^p0\{0,1\}^\omega$. Par hypothèse de récurrence, $\sigma_0^{x_n} = \#^{d_{p_1}+q+1+p}\mathcal{C}_p\#^\omega$. Comme F est une stratégie gagnante, pour satisfaire φ_M, (et en particulier γ), on en déduit que $\sigma_0^{y_m} = \#^{d_{p_2}+p+q+1}\mathcal{C}_p\#^\omega$. De plus, $\sigma_0^{y_m} = \sigma^{y_m}$. Or σ est tel que $n(x_0) = n(y_0)+1$, donc pour satisfaire la spécification (et en particulier la sous-formule δ), nécessairement $\sigma^{x_n} = \#^{d_{p_1}+p+q+1}\mathcal{C}_{p+1}\#^\omega$.

Pour une machine de Turing M, on a montré que toute stratégie distribuée gagnante pour $(\mathcal{A}', \varphi_M)$ est contrainte d'écrire sur la variable x_n la $n(x_0)$-ième configuration de M. par conséquent, il existe une stratégie distribuée gagnante pour $(\mathcal{A}', \varphi_M \wedge \mathsf{G}(x_n \neq \mathrm{halt}))$ si et seulement si M ne s'arrête pas lorsque son entrée est la bande vide. On a donc réduit le problème du non-arrêt d'une machine de Turing au problème de SSD synchrone pour $(\mathcal{A}, \varphi)$ avec $\mathcal{A}$ architecture avec délais arbitraires, et $\varphi \in \mathsf{LTL}(V_\mathrm{I} \cup V_\mathrm{O})$, démontrant par là que ce dernier problème est indécidable.

On donne à présent les idées nécessaires pour obtenir le résultat pour les spécifications CTL : il faut adapter la formule φ_M pour la rendre exprimable en CTL. La formule utilise le fait que, dans un arbre de stratégie modèle de la spécification, à partir du moment où on a vu une séquence finie particulière de la forme 0^q1^p0 sur x_0, alors toutes les branches partant de ce nœud ont les mêmes contraintes sur x_n (et respectivement pour y_0 et y_m). La formule suivante permet d'obtenir le résultat :

$$\varphi = \alpha \wedge \beta \wedge \gamma$$

où

1. la formule α correspond à la formule α de LTL précédemment décrite, dans laquelle on décrit maintenant les différentes branches des arbres de stratégies que l'on veut accepter : la projection de l'étiquette de toute branche sur sa composante x_0 (respectivement y_0) est une séquence de 0 et de 1, et sur les branches pour lesquelles cette séquence vaut $0^q1^p0\{0,1\}^\omega$, alors les valeurs prises par x_n (respectivement y_m) sont dans $\#^{d_{p_1}+q+p}\Gamma^*Q\Gamma^+\#^\omega$. Soit formellement la formule $\alpha \in \mathsf{CTL}(V_\mathrm{I}' \cup V_\mathrm{O}')$ suivante :

$$\alpha = \alpha_{x_0,x_n,d_{p_1}} \wedge \alpha_{y_0,y_m,d_{p_2}}$$

où

$$\alpha_{z,t,d} \stackrel{\text{def}}{=} \mathsf{A}(z=0 \wedge t=\#)\mathsf{W}\big((z=1 \wedge t=\#) \wedge (\mathsf{A}(z=1 \wedge t=\#)\mathsf{W}((z=0) \wedge \mathrm{Config}(t,d)))\big)$$

avec $\mathrm{Config}(t,d)$ exprimé par

$$\mathsf{A}(t=\#)\,\mathsf{U}_d\,\Big(\mathsf{A}(t\in\Gamma)\,\mathsf{U}\,\big((t\in Q)\wedge\mathsf{AX}(\mathsf{A}(t\in\Gamma)\,\mathsf{U}\,((t\in\Gamma)\wedge\mathsf{AX}\,\mathsf{AG}(t=\#))))\Big)$$

et

$$\begin{aligned}&\mathsf{A}\,\varphi\,\mathsf{U}_0\,\psi = \psi\\ &\mathsf{A}\,\varphi\,\mathsf{U}_i\,\psi = \varphi \wedge \mathsf{AX}\,\varphi \wedge \cdots \wedge ((\mathsf{AX})^{i-1}\varphi) \wedge ((\mathsf{AX})^i\psi)\end{aligned}$$

2. la formule β correspond également à la formule β de la formule LTL que l'on a donnée au début de la démonstration : toutes les branches des modèles que l'on cherche vérifient la propriété suivante : soit la séquence de valeurs prises par x_0 ne vérifie pas $n(x_0)=1$, soit la séquence de valeurs prise par x_n doit correspondre à $\mathcal{C}_1$. Soit :

$$\beta \stackrel{\text{def}}{=} \mathsf{A}(x_0=0)\,\mathsf{W}\,\Big((x_0=1)\wedge\mathsf{AX}\big((x_0=1)\vee((x_0=0)\wedge\mathsf{A}(x\in\#^{d_{p_1}}\mathcal{C}_1\#^\omega))\big)\Big)$$

dans laquelle $\mathsf{A}(x \in \#^{d_{p_1}}\mathcal{C}_1\#^\omega)$ est la formule CTL reprenant la formule de chemin $x \in \#^{d_{p_1}}\mathcal{C}_1\#^\omega$ où on a remplacé les modalités X et U respectivement par AX et $\mathsf{A}\,\mathsf{U}$.

3. la formule γ reprend les formules γ et δ de la spécification LTL. Elle décrit à nouveau toutes les branches de l'arbre que l'on cherche : soit les séquences de valeurs prises par x_0 et y_0 ne sont pas synchronisées, soit elles sont synchronisées et $n(x_0)=n(y_0)$ et alors les valeurs prises par x_n et y_m sur tous les états accessibles à partir de ce moment sont les mêmes (mais la spécification α implique qu'en fait les séquences de valeurs prises par x_n et y_m sont les mêmes sur toute les branches *initiales* concernées), soit elles sont synchronisées et $n(x_0)=n(y_0)+1$ et alors, sur toutes les branches correspondant à ce cas de figure, la configuration codée sur x_n est la configuration successeur de celle encodée sur y_m, au décalage de leurs délais près. Soit :

$$\begin{aligned}\gamma \stackrel{\text{def}}{=} \mathsf{A}(x_0=y_0=0)\,\mathsf{W}\,\Big((x_0=0\wedge y_0=1)\vee&\\ \big((x_0=y_0=1)\wedge\mathsf{AX}(\mathsf{A}(x_0=y_0=1)\,\mathsf{W}\,((x_0\neq y_0)\vee&\\ \mathrm{SyncEgal}(x_0,y_0,x_n,y_m)))\big)\vee&\\ \big((x_0=1\wedge y_0=0)\wedge\mathsf{EX}((x_0=y_0=1)\wedge\mathsf{AX}(\mathsf{A}(x_0=y_0=1)&\\ \mathsf{W}\,((x_0\neq y_0)\vee\mathrm{SyncDecal}(x_0,y_0,x_n,y_m)))))\big)\Big)\end{aligned}$$

avec

$$\mathrm{SyncEgal}(x,y,z,t) = (x=y=0) \wedge \mathsf{AG}\big(\bigvee_{i\in\Gamma\uplus Q\uplus\{\#\}} ((z=i)\wedge(\mathsf{AX})^{d_{p_2}-d_{p_1}}(t=i))\big)$$

et

$$\begin{aligned}\mathrm{SyncDecal}(x,y,z,t) = (x=y=0)\wedge \mathsf{A}\big(&\bigvee_{i\in\Gamma\uplus Q\uplus\{\#\}} ((z=i)\wedge(\mathsf{AX})^{d_{p_2}-d_{p_1}}(t=i))\big)\,\mathsf{U}\\ &\big(\mathsf{A}\,\mathrm{Trans}(t,z)\wedge\\ &\mathsf{AX}^3\,\mathsf{AG}\big(\bigvee_{i\in\Gamma\uplus Q\uplus\{\#\}} ((z=i)\wedge(\mathsf{AX})^{d_{p_2}-d_{p_1}}(t=i))\big)\big).\end{aligned}$$

et $\mathsf{A}\,\mathrm{Trans}(t,z)$ la formule $\mathrm{Trans}(t,z)$ définie dans le cas LTL et dans laquelle on remplace les modalités X par AX.

Ainsi, si on considère une stratégie gagnante pour $(\mathcal{A}',\varphi)$ dont l'arbre des exécutions est $t : (S'^{V_I'})^* \to S'^{V'}$, on peut montrer que, si on prend une branche initiale finie de t étiquetée par 0^q1^p0 sur sa composante x_0 alors toutes les branches infinies ayant ce segment initial sont étiquetées par $\#^{d_{p_1}+p+q}\mathcal{C}_p\#^\omega$ sur leur composante x_n. Une fois de plus, on peut le démontrer par récurrence, le cas de base étant assuré par α et β. Si on considère ensuite une branche de t étiquetée par $0^q1^{p+1}0\{0,1\}^\omega$ sur sa composante x_0, alors l'hypothèse de récurrence nous assure que toutes les branches étiquetées par une séquence dans $0^{q+1}1^p0\{0,1\}^\omega$ sur x_0 et $0^{q+1}1^p0\{0,1\}^\omega$ sur y_0 sont étiquetées par $\#^{d_{p_1}+p+q+1}\mathcal{C}_p\#^\omega$ sur x_n. Donc la formule γ impose que ces branches soient étiquetées par $\#^{d_{p_2}+p+q+1}\mathcal{C}_p\#^\omega$ sur y_m. Cela implique, la stratégie pour y_m ne dépendant que de y_0, que toutes les branches étiquetées par $0^{q+1}1^p0\{0,1\}^\omega$ sur y_0 sont étiquetées par $\#^{d_{p_2}+p+q+1}\mathcal{C}_p\#^\omega$ sur y_m. Puis, la formule γ impose à nouveau que toutes les branches étiquetées par $0^q1^{p+1}0\{0,1\}^\omega$ sur x_0 et $0^{q+1}1^p0\{0,1\}^\omega$ sur y_0 soient étiquetées par $\#^{d_{p_1}+p+q+1}\mathcal{C}_{p+1}\#^\omega$ sur x_n. Le fait que la stratégie de x_n ne dépende que de x_0 permet de conclure. □

On montre à présent que le fait d'être à information incomparable est une condition suffisante pour entraîner l'indécidabilité du problème de SSD synchrone pour les spécifications externes.

Proposition 3.14. *Le problème de SSD synchrone est indécidable pour les architectures à information incomparable et les spécifications externes de* LTL *ou* CTL.

Pour toute architecture $\mathcal{A}$ à information incomparable, on réduit le problème de SSD synchrone pour $(\mathcal{A}',\varphi)$ avec φ spécification externe de LTL ou CTL au problème de SSD synchrone pour $(\mathcal{A},\overline{\varphi})$ avec $\overline{\varphi}$ spécification externe de LTL ou CTL. Si le fait qu'il existe une réduction n'est pas surprenant, la démonstration n'est pas complètement immédiate. En particulier, la spécification utilisée doit être modifiée au cours de la réduction. La fin de cette section est consacrée à la preuve de la proposition 3.14.

Considérons maintenant $\mathcal{A} = (\mathrm{Proc}, V, E, (S^v)_{v\in V}, s_0, (d_p)_{p\in\mathrm{Proc}})$ une architecture à information incomparable. Sans perte de généralité, on peut considérer que $s_0^v = 0$ pour tout $v \in V$. Par définition, on peut trouver deux variables de sortie x_n et $y_m \in V_O$ et deux variables d'entrée $x_0 \in \mathrm{Vue}(x_n)$ et $y_0 \in \mathrm{Vue}(y_m)$ telles que $x_0 \notin \mathrm{Vue}(y_m)$ et $y_0 \notin \mathrm{Vue}(x_n)$. On considère deux chemins $x_0E^2x_1E^2\dots E^2x_n$ allant de x_0 à x_n et $y_0E^2y_1E^2\dots E^2y_m$ allant de y_0 à y_m

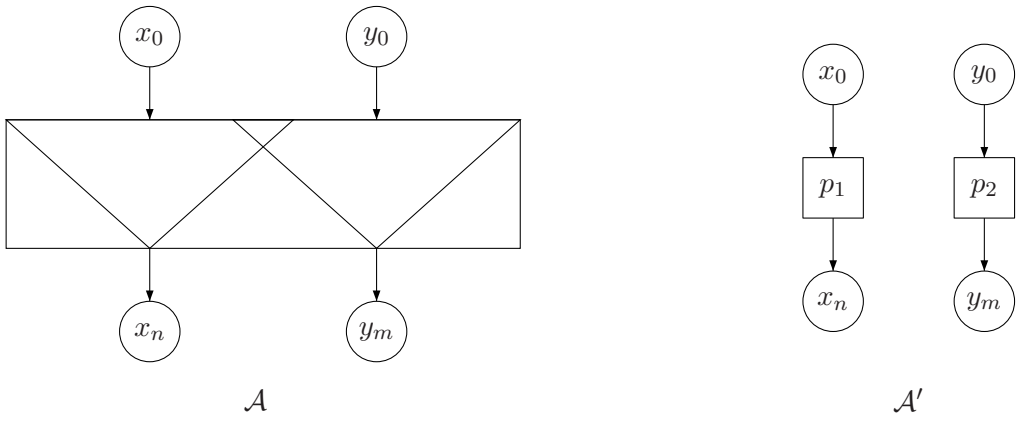


FIG. 3.4 – Architectures $\mathcal{A}$ and $\mathcal{A}'$

vérifiant $d(x_0, x_n) = d_{x_1} + \cdots + d_{x_n}$ et $d(y_0, y_m) = d_{y_1} + \cdots + d_{y_m}$. Par ailleurs, les ensembles $\{x_0, \ldots, x_n\}$ et $\{y_0, \ldots, y_m\}$ sont disjoints.

Soit $\mathcal{A}' = (\text{Proc}', V', E', (S'^v)_{v \in V'}, s'_0)$ l'architecture de la figure 3.4, avec ${V_\text{I}}' = \{x_0, y_0\}$ et ${V_\text{O}}' = \{x_n, y_m\}$; avec les mêmes domaines de variables de sortie $S'^{x_n} = S^{x_n}$ et $S'^{y_m} = S^{y_m}$; avec $S'^{x_0} = S'^{y_0} = \{0, 1\}$ pour domaines de variables d'entrée ; et avec $s'_0 = s_0^{V'}$. Les délais des processus $(d'_p)_{p \in \text{Proc}'}$ sont les délais cumulatifs minimaux de transmission d'information de x_0 à x_n et de y_0 à y_m : $d'_{p_1} = d'_{x_n} = d(x_0, x_n)$ et $d'_{p_2} = d'_{y_m} = d(y_0, y_m)$.

On sait que le problème SSD synchrone est indécidable pour $(\mathcal{A}', \varphi)$ avec φ spécification LTL ou CTL (théorème 3.13). On présente maintenant la réduction. Considérons d'abord les spécifications CTL.

Remarquons tout d'abord qu'il est nécessaire de modifier la spécification au cours de la réduction. En effet, si on considère

$$\psi = \mathsf{EG}((x_0 = 0) \wedge (x_n = 0)) \wedge \mathsf{EG}((x_0 = 0) \wedge (x_n = 1)))$$

il existe une stratégie distribuée gagnante pour $(\mathcal{A}, \psi)$ (pourvu que $\text{Vue}(x_n) \setminus \{x_0\} \neq \emptyset$ et que tous les délais soient nuls), mais pas pour $(\mathcal{A}', \psi)$: dans $\mathcal{A}$, la valeur de x_n peut être déterminée par la valeur d'une variable de $\text{Vue}(x_n) \setminus \{x_0\}$, tandis que dans $\mathcal{A}'$, la stratégie contrôlant x_n ne peut dépendre que de x_0.

Afin de définir une stratégie gagnante F' pour $\mathcal{A}'$ à partir d'une stratégie gagnante F pour $\mathcal{A}$, on va simuler sur $\mathcal{A}'$ le comportement de F lorsque toutes les variables d'entrée différentes de ${V_\text{I}}' = \{x_0, y_0\}$ valent 0 de façon constante au long de l'exécution. Pour cela nous allons contraindre le comportement des processus de $\mathcal{A}$ en transformant la spécification au cours de la réduction, par l'utilisation de la formule $\chi = (x_0 \in \{0,1\}) \wedge (y_0 \in \{0,1\}) \wedge \bigwedge_{v \in {V_\text{I}} \setminus {V'_\text{I}}} (v = 0)$, où $x \in I$ est une abbréviation pour $\bigvee_{i \in I}(x = i)$. On définit à présent la réduction d'une formule $\psi \in \mathsf{CTL}(V')$ en une formule $\overline{\psi} \in \mathsf{CTL}(V)$ imposant ψ sur le sous-arbre des exécutions vérifiant χ :

$$\overline{(x = s)} = (x = s) \qquad \overline{\neg\psi} = \neg\overline{\psi}$$
$$\overline{\varphi \vee \psi} = \overline{\varphi} \vee \overline{\psi} \qquad \overline{\mathsf{EX}\,\psi} = \mathsf{EX}(\chi \wedge \overline{\psi})$$
$$\overline{\mathsf{E}\,\varphi\,\mathsf{U}\,\psi} = \mathsf{E}(\chi \wedge \overline{\varphi})\,\mathsf{U}\,(\chi \wedge \overline{\psi}) \qquad \overline{\mathsf{EG}\,\psi} = \mathsf{EG}(\chi \wedge \overline{\psi})$$

On utilise la notation suivante : pour tout $r \in S'^{V_{\mathrm{I}}'}$, on définit $\overline{r} \in S^{V_{\mathrm{I}}}$ par $\overline{r}^{V_{\mathrm{I}}'} = r$ et $\overline{r}^v = 0$ pour tout $v \in V_{\mathrm{I}} \setminus V_{\mathrm{I}}'$. On étend cette notation aux mots, en prenant $\overline{\varepsilon} = \varepsilon$. Ceci nous permet de définir un arbre d'exécutions $\tilde{t} : (S'^{V_{\mathrm{I}}'})^* \to S^{V'}$ sur $\mathcal{A}'$ à partir d'un arbre d'exécutions $t : S^{V_{\mathrm{I}}} \to S^V$ sur $\mathcal{A}$, en posant, pour tout $\rho \in (S'^{V_{\mathrm{I}}'})^*$, $\tilde{t}(\rho) = t(\overline{\rho})^{V'}$. L'arbre $\tilde{t}$ représente donc les exécutions sur $\mathcal{A}$ dans lesquelles tous les $v \in V_{\mathrm{I}} \setminus V_{\mathrm{I}}'$ valent 0. La réduction de la formule est correcte dans le sens suivant :

Lemme 3.15. *Pour toute formule $\psi \in \mathsf{CTL}(V')$, pour tout arbre $t : (S^{V_I})^* \to S^V$ et pour tout $\rho \in (S'^{V_I'})^*$, $t, \overline{\rho} \models \overline{\psi}$ si et seulement si $\tilde{t}, \rho \models \psi$.*

Démonstration. Par récurrence sur la structure de la formule ψ. Soit $t : (S^{V_{\mathrm{I}}})^* \to S^V$, et soit $\rho \in (S'^{V_{\mathrm{I}}'})^*$.

Si $\psi = (x = s)$, avec $x \in V'$ et $s \in S^x$, alors le résultat se déduit du fait que $\tilde{t}(\rho)^x = t(\overline{\rho})^x$.

Les cas où $\psi = \neg\varphi$ ou $\psi = \psi_1 \vee \psi_2$ sont triviaux.

Soit maintenant $\psi = \mathsf{E}\,\psi_1\,\mathsf{U}\,\psi_2$ et supposons que $t, \overline{\rho} \models \overline{\psi}$. Alors il existe $s_1 \cdots s_n \in (S^{V_{\mathrm{I}}})^*$ tels que $t, \overline{\rho} \cdot s_1 \cdots s_n \models \chi \wedge \overline{\psi_2}$ et $t, \overline{\rho} \cdot s_1 \cdots s_i \models \chi \wedge \overline{\psi_1}$, pour tout $0 \leq i < n$. Comme $t, \overline{\rho} \cdot s_1 \cdots s_i \models \chi$, alors $s_i = \overline{r_i}$ avec $r_i = s_i^{V_{\mathrm{I}}'} \in S'^{V_{\mathrm{I}}'}$. En appliquant l'hypothèse de récurrence, on obtient que $\tilde{t}, \rho \cdot r_1 \cdots r_i \models \psi_1$ pour tout $0 \leq i < n$, et $\tilde{t}, \rho \cdot r_1 \cdots r_n \models \psi_2$, donc que $\tilde{t}, \rho \models \psi$. Réciproquement, supposons que $\tilde{t}, \rho \models \psi$. Alors il existe $r_1 \cdots r_n \in (S'^{V_{\mathrm{I}}'})^*$ tels que $\tilde{t}, \rho \cdot r_1 \cdots r_n \models \psi_2$ et $\tilde{t}, \rho \cdot r_1 \cdots r_i \models \psi_1$, pour tout $0 \leq i < n$. Par hypothèse de récurrence, on en déduit que $t, \overline{\rho \cdot r_1 \cdots r_n} \models \overline{\psi_2}$, et que $t, \overline{\rho \cdot r_1 \cdots r_i} \models \overline{\psi_1}$, pour tout $0 \leq i < n$. De plus, par construction, pour tout $0 \leq i \leq n$, $t(\overline{\rho \cdot r_1 \cdots r_i})^{V_{\mathrm{I}}'} \in \{0,1\}$ et pour tout $v \in V_{\mathrm{I}} \setminus V_{\mathrm{I}}'$, $t(\overline{\rho \cdot r_1 \cdots r_i})^v = 0$. Donc en posant $s_i = \overline{r_i}$ pour tout $1 \leq i \leq n$, on obtient qu'il existe $s_1, \cdots, s_n \in S^{V_{\mathrm{I}}}$ tels que $t, \overline{\rho} \cdot s_1 \cdots s_n \models \chi \wedge \overline{\psi_2}$ et $t, \overline{\rho} \cdot s_1 \cdots s_i \models \chi \wedge \overline{\psi_1}$, pour tout $0 \leq i < n$, et donc que $t, \overline{\rho} \models \overline{\psi_1\,\mathsf{U}\,\psi_2}$.

Les cas EX et EG s'obtiennent de la même façon. □

On prouve à présent la réduction proprement dite :

Lemme 3.16. *S'il existe un programme distribué F' gagnant pour $(\mathcal{A}', \psi)$, alors il existe un programme distribué F gagnant pour $(\mathcal{A}, \overline{\psi})$.*

Démonstration. Fixons $F' = (f'^{x_n}, f'^{y_m})$ une stratégie distribuée gagnante pour $(\mathcal{A}', \psi)$. On va définir une stratégie distribuée $F = (f^v)_{v \in V \setminus V_{\mathrm{I}}}$ gagnante pour $(\mathcal{A}, \overline{\psi})$ telle que la projection sur V' d'une exécution F-compatible soit une exécution F'-compatible. Plus précisément, si $\sigma \in (S^V)^+$ est un préfixe d'une exécution selon F vérifiant $\sigma^{x_0} \in \{0,1\}^+$ et $\sigma^{y_0} \in \{0,1\}^+$, alors on veut que les égalités suivantes soient vérifiées :

$$f^{x_n}(\sigma^{R(x_n)}) = f'^{x_n}(\sigma^{x_0}) \tag{3.5}$$

$$f^{y_m}(\sigma^{R(y_m)}) = f'^{y_m}(\sigma^{y_0}) \tag{3.6}$$

Pour cela, on va utiliser les variables $x_1, \ldots, x_{n-1}$ pour transmettre les valeurs de x_0 au processus écrivant sur x_n. Formellement, la stratégie f^{x_k} recopie sur x_k la dernière valeur de x_{k-1} dont elle dépend : celle écrite d_{x_k} instants auparavant. Pour $0 < k < n$ et $\tau \in (S^{R(x_k)})^+$, on définit donc

$$f^{x^k}(\tau) = \begin{cases} s^{x_{k-1}} & \text{si } \tau = \tau_1 s \tau_2 \text{ avec } |\tau_2| = d_{x_k} \text{ et } s^{x_{k-1}} \in \{0,1\} \\ 0 & \text{sinon.} \end{cases}$$

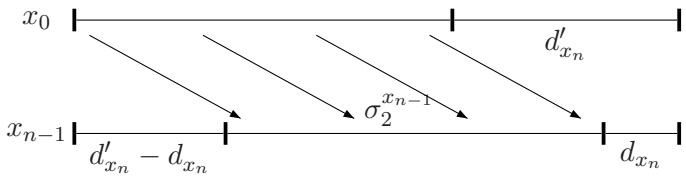


FIG. 3.5 – Simulation de f'^{x_n} par f^{x_n}.

Par définition, f^{x_k} est bien d-compatible. On peut vérifier facilement que si la séquence de valeurs prises par x_0 est $\rho \in \{0,1\}^\omega$, en respectant les stratégies f^{x_k} définies ci-dessus, les valeurs prises par x_{n-1} sont $0^{d(x_0,x_{n-1})}\rho$ où la séquence initiale de 0 correspond au décalage induit par le délai de transmission $d(x_0, x_{n-1}) = d'_{x_n} - d_{x_n}$.

Afin de satisfaire (3.5), la dernière stratégie f^{x_n} simule f'^{x_n} en tenant compte du décalage de $d'_{x_n} - d_{x_n}$ unités de temps de leurs délais respectifs. Considérons une séquence $\sigma \in (S^V)^+$ compatible avec les stratégies $(f^{x_k})_{0<k<n}$ telle que $\sigma^{x_0} \in \{0,1\}^+$. Si $|\sigma| \leq d'_{x_n}$, alors $f'^{x_n}(\sigma^{x_0})$ ne dépend pas de σ, et on définit $f^{x_n}(\sigma^{R(x_n)}) = f'^{x_n}(0^{|\sigma|}) = f'^{x_n}(\sigma^{x_0})$, et l'égalité (3.5) est vérifiée. Si maintenant $\sigma = \sigma_1\sigma_2\sigma_3$, avec $|\sigma_1| = d'_{x_n} - d_{x_n}$, $|\sigma_3| = d_{x_n}$, alors $f'^{x_n}(\sigma^{x_0})$ ne dépend que de son préfixe de taille $|\sigma_2| = i$. On remarque que $\sigma[i]^{x_0} = \sigma_2^{x_{n-1}}$ (voir la figure 3.5), donc on pose dans ce cas $f^{x_n}(\sigma^{R(x_n)}) = f'^{x_n}(\sigma_2^{x_{n-1}} 0^{d'_{x_n}}) = f'^{x_n}(\sigma^{x_0})$ et (3.5) est vérifiée. On définit donc f^{x_n} formellement, pour tout $\tau \in (S^{R(x_n)})^+$, par

$$f^{x_n}(\tau) = \begin{cases} f'^{x_n}(0^{|\tau|}) & \text{si } |\tau| \leq d'_{x_n}, \\ f'^{x_n}(\tau_2^{x_{n-1}} 0^{d'_{x_n}}) & \text{si } \tau = \tau_1\tau_2\tau_3 \text{ tels que } |\tau_1| = d'_{x_n} - d_{x_n},\ |\tau_3| = d_{x_n} \\ & \text{et } \tau_2^{x_{n-1}} \in \{0,1\}^+, \\ 0 & \text{sinon.} \end{cases}$$

Il est clair que f^{x_n} est compatible avec le délai d_{x_n}, et que l'égalité (3.5) est vérifiée avec cette définition des stratégies $(f^{x_k})_{0<k\leq n}$. On définit de façon similaire les stratégies f^{y_k}, pour $0 < k \leq m$, et pour toutes les autres variables v, on pose $f^v = 0$. On obtient une stratégie distribuée F compatible avec les délais, pour laquelle il reste à montrer qu'elle est bien gagnante pour $(\mathcal{A}, \overline{\psi})$.

Soit $t : (S^{V_\mathrm{I}})^* \to S^V$ l'arbre d'exécutions respectant F sur $\mathcal{A}$. On va montrer que $\tilde{t} : (S'^{V_\mathrm{I}'})^* \to S^{V'}$ est l'arbre d'exécutions respectant F' sur $\mathcal{A}'$. Ainsi, comme F' est une stratégie distribuée gagnante pour $(\mathcal{A}', \psi)$, $\tilde{t}, \varepsilon \models \psi$ donc, par le lemme 3.15, $t, \varepsilon \models \overline{\psi}$, et F est une stratégie distribuée gagnante pour $(\mathcal{A}, \overline{\psi})$.

Montrons à présent que $\tilde{t}$ est bien l'arbre d'exécutions respectant F'. Tout d'abord, il est facile de voir que $\tilde{t}$ est un arbre d'exécutions sur $\mathcal{A}'$: $\tilde{t}(\varepsilon) = t(\overline{\varepsilon})^{V'} = s_0^{V'} = s'_0$, et pour tout $\rho \in (S^{V_\mathrm{I}'})^*$, tout $r \in S^{V_\mathrm{I}'}$, $\tilde{t}(\rho \cdot r)^{V_\mathrm{I}'} = t(\overline{\rho} \cdot \overline{r})^{V_\mathrm{I}'} = \overline{r}^{V_\mathrm{I}'} = r$. Pour montrer que c'est bien l'arbre des exécutions respectant F', on doit encore montrer que, pour tout $\rho = r_1 \cdots r_i \in (S'^{V_\mathrm{I}'})^+$, $\tilde{t}(\rho)^{x_n} = f'^{x_n}(\rho^{x_0})$ et $\tilde{t}(\rho)^{y_m} = f'^{y_m}(\rho^{x_0})$. Soit $\sigma \in (S^V)^+$ l'exécution respectant F induite par la séquence d'entrée $\overline{\rho} : \sigma = t(\varepsilon)t(\overline{r_1})t(\overline{r_1 r_2}) \cdots t(\overline{\rho})$. En utilisant le fait que (3.5) est vérifié, on obtient $\tilde{t}(\rho)^{x_n} = t(\overline{\rho})^{x_n} = f^{x_n}(\sigma^{R(x_n)}) = f'^{x_n}(\sigma^{x_0}) = f'^{x_n}(\rho^{x_0})$. En utilisant les mêmes arguments, on obtient que $\tilde{t}(\rho)^{y_m} = f'^{y_m}(\rho^{y_0})$, et donc que $\tilde{t}$ est l'arbre des exécutions respectant F'. □

Lemme 3.17. *S'il existe un programme distribué F gagnant pour $(\mathcal{A}, \overline{\psi})$, alors il existe un programme distribué F' gagnant pour $(\mathcal{A}', \psi)$.*

Démonstration. Fixons $F = (f^v)_{v \in V \setminus V_I}$ une stratégie distribuée gagnante pour $(\mathcal{A}, \overline{\psi})$. On va définir $f'^{x_n} : (S'^{x_0})^+ \to S'^{x_n}$ et $f'^{y_m} : (S'^{y_0})^+ \to S'^{y_m}$ les stratégies pour les variables de $\mathcal{A}'$. La difficulté supplémentaire réside ici dans le fait que f'^{x_n} a en général moins de variables d'entrée que $\hat{f}^{x_n}$, donc on ne peut pas simuler directement les stratégies de $\mathcal{A}$. Pour résoudre ce problème, on va utiliser le fait que, de par la forme particulière de $\overline{\psi}$, l'arbre des exécutions respectant F satisfait $\overline{\psi}$ si et seulement si le sous-arbre restreint aux branches pour lesquelles la valeur de toutes les variables d'entrée $v \in V \setminus \{x_0, y_0\}$ est toujours 0 satisfait aussi $\overline{\psi}$ (lemme 3.15). Les processus de $\mathcal{A}'$ vont donc se comporter comme ceux écrivant sur x_n et y_m dans $\mathcal{A}$ dans les exécutions particulières où toutes les variables d'entrée différentes de x_0 et y_0 valent constamment 0.

Formellement, pour $\rho \in (S'^{V_I'})^+$, on pose $f'^{x_n}(\rho^{x_0}) = \hat{f}^{x_n}(\overline{\rho}^{\mathrm{Vue}(x_n)})$. Remarquons que, comme $\mathcal{A}$ est à information incomparable, $y_0 \notin \mathrm{Vue}(x_n)$ et donc $\hat{f}^{x_n}$ ne dépend pas de $\overline{\rho}^{y_0}$. Donc f'^{x_n} ne dépend que de ρ^{x_0} et f'^{x_n} est une stratégie à mémoire locale pour x_n dans l'architecture $\mathcal{A}'$. De plus, f^{x_n} est d-compatible, donc, comme $d'_{x_n} = d(x_0, x_n)$, f'^{x_n} est également d-compatible. On définit f'^{y_m} de façon similaire et on vérifie que $F' = (f'^{x_n}, f'^{y_m})$ est bien une stratégie gagnante pour $(\mathcal{A}', \psi)$. Soit t l'arbre des exécutions de F sur $\mathcal{A}$ et t' l'arbre des exécutions de F' sur $\mathcal{A}'$. Alors $t'(\rho)^{x_n} = f'^{x_n}(\rho^{x_0}) = \hat{f}^{x_n}(\overline{\rho}^{\mathrm{Vue}(x_n)}) = t(\overline{\rho})^{x_n} = \tilde{t}(\rho)^{x_n}$ et, de la même façon, $t'(\rho)^{y_m} = \tilde{t}(\rho)^{y_m}$. Donc $t' = \tilde{t}$. De plus, $t, \varepsilon \models \overline{\psi}$, donc, par le lemme 3.15, $t', \varepsilon \models \psi$ et F' est bien une stratégie distribuée gagnante pour $(\mathcal{A}', \psi)$. □

On prouve maintenant le résultat pour les spécifications LTL. Dans ce cas, la spécification sur $\mathcal{A}$ doit juste assurer que les valeurs des variables x_0 et y_0 restent dans le domaine autorisé dans $\mathcal{A}'$ au cours de l'exécution. On va utiliser la réduction

$$\overline{\psi} = (\mathsf{G}\,\xi) \to \psi$$

où la formule ξ est définie par $\xi = (x_0 \in \{0,1\}) \wedge (y_0 \in \{0,1\})$.

On utilise les mêmes constructions que celles décrites dans les lemmes 3.16 et 3.17 pour obtenir la réduction. En effet, soit F' une stratégie distribuée gagnante pour $(\mathcal{A}', \psi)$ et soit F la stratégie distribuée définie dans la preuve du lemme 3.16. Soit $\sigma = s_0 s_1 \cdots \in (S^V)^\omega$ une exécution respectant la stratégie F et soit $\rho = \sigma^{V_I}$ sa projection sur les variables d'entrée. Si $\rho^{x_0} \notin \{0,1\}^\omega$ ou $\rho^{y_0} \notin \{0,1\}^\omega$, alors $\sigma \not\models \mathsf{G}\,\xi$. Sinon, par les égalités (3.5) et (3.6), on obtient pour $i > 0$, $s_i^{x_n} = f^{x_n}(\sigma^{R(x_n)}[i]) = f'^{x_n}(\sigma^{x_0}[i])$ et $s_i^{y_m} = f^{y_m}(\sigma^{R(y_m)}[i]) = f'^{y_m}(\sigma^{y_0}[i])$. On en déduit que $\sigma^{V'}$ est une exécution respectant F', et donc $\sigma^{V'} \models \psi$. Comme $\psi \in \mathsf{LTL}(V')$, on a $\sigma \models \psi$. On en conclut que toute exécution σ respectant F est telle que $\sigma \models \mathsf{G}\,\xi \to \psi$ et donc que F est une stratégie distribuée gagnante pour $(\mathcal{A}, \overline{\psi})$.

Réciproquement, pour F stratégie distribuée gagnante pour $(\mathcal{A}, \overline{\psi})$, on définit F' comme dans la preuve du lemme 3.17. Soit $\rho \in (S'^{V_I'})^\omega$ une séquence de valeurs d'entrée et $\sigma' \in (S'^{V'})^\omega$ l'exécution F'-compatible induite. Soit $\sigma = s_0 s_1 \cdots \in (S^V)^\omega$ l'exécution respectant F telle que $\sigma^{V_I} = \overline{\rho}$. Par définition, $\sigma \models \mathsf{G}\,\xi$, et donc, comme F est une stratégie distribuée gagnante pour $(\mathcal{A}, \overline{\psi})$, alors $\sigma \models \psi$. À nouveau, comme $\psi \in \mathsf{LTL}(V')$, cela implique que $\sigma^{V'} \models \psi$. On obtient immédiatement des définitions de f'^{x_n} et f'^{y_m} que $\sigma^{V'}$ est une exécution respectant F'. Par ailleurs $\sigma^{V_I'} = \rho$ donc $\sigma^{V'} = \sigma'$. On en conclut que F' est une stratégie distribuée gagnante pour $(\mathcal{A}', \psi)$.

On a donc défini une réduction du problème de SSD synchrone avec spécifications de LTL ou CTL sur l'architecture $\mathcal{A}'$ au même problème sur une architecture avec information incomparable. Comme le problème de synthèse est indécidable à la fois pour des spécifications LTL et CTL sur $\mathcal{A}'$, on obtient son indécidabilité pour les architectures à information incomparable.

3 Architectures uniformément bien connectées

Dans cette section, on introduit une nouvelle classe d'architectures : les architectures uniformément bien connectées (UWC), et on fournit un critère de décidabilité pour le problème de SSD synchrone pour cette classe. On montre également qu'il est décidable de déterminer si une architecture donnée est UWC, et on donne la complexité de ce problème. Enfin on introduit la notion de spécification *robuste* et on montre que les architectures UWC sont toutes décidables pour ces spécifications. Informellement, une architecture est UWC si on peut transmettre à chaque instant aux processus modifiant les variables de sortie toutes les valeurs de variables dans leur vue. Si le domaine des variables internes est arbitrairement grand, on se convainc facilement que toutes les architectures sont UWC. Ceci sera prouvé à la section suivante.

3.1 Définition

Un *routage* pour une architecture distribuée $\mathcal{A} = (\text{Proc}, V, E, (S^v)_{v\in V}, s_0, (d_p)_{p\in\text{Proc}})$ est une famille de stratégies locales *sans mémoire* $\Phi = (f^v)_{v\in V\setminus(V_\text{I}\cup V_\text{O})}$. On remarque qu'un routage ne comprend pas de stratégies pour les variables de sortie du système. On dit qu'une architecture est uniformément bien connectée (UWC) s'il existe un routage Φ qui permet de transmettre avec un *délai minimal* au processus modifiant une variable $v \in V_\text{O}$ toutes les valeurs des variables de $\text{Vue}(v)$. Formellement,

Définition 3.18. *Une architecture* $\mathcal{A} = (\text{Proc}, V, E, (S^v)_{v\in V}, s_0, (d_p)_{p\in\text{Proc}})$ *est* uniformément bien connectée *(UWC), s'il existe un routage* Φ *et, pour chaque variable de sortie* $v \in V_O$ *et chaque variable d'entrée* $u \in Vue(v)$*, une fonction de décodage* $g^{u,v} : (S^{R(v)})^+ \to S^u$ *qui permet de reconstruire la valeur de* u*, c'est-à-dire telle que, pour toute séquence* $\sigma = s_1 \cdots \in (S^{V\setminus V_O})^+$ Φ*-compatible, on a, pour* $i \geq 1$*,*

$$s_i^u = g^{u,v}(\sigma^{R(v)}[i + d(u, v) - d_v]) \tag{3.7}$$

Remarque 3.19. On n'inclut pas l'état initial dans la séquence. En effet, les processus n'ont pas besoin de le décoder pour le connaître.

Dans le cas d'une architecture sans délai, la notion d'architecture uniformément bien connectée raffine la notion de connectivité adéquate introduite par Pnueli et Rosner [PR90], puisqu'on n'impose plus de transmettre à chaque variable de sortie la valeur de *toutes* les variables d'entrée, mais uniquement de celles appartenant à sa vue. (Si les variables ont toutes le même domaine, les architectures représentées sur la sous-figure (3.3(b)) de la figure 3.3 sont adéquatement connectées).

On note que, si les fonctions de routage sont sans mémoire, on autorise les fonctions de décodage à tenir compte de leur passé ; en fait, la mémoire peut même être nécessaire pour parvenir à reconstituer les valeurs des variables en entrée. En effet, considérons l'architecture représentée dans la figure 3.6. Les délais sont indiqués à côté des processus, et le domaine de toutes les variables est $\{0, 1\}$. Cette architecture est bien UWC : le processus p écrit sur la variable t le résultat de l'addition modulo 2 (ou XOR) de u_1 et u_2 avec un délai de 1. On va l'écrire $t = \mathsf{Y}\, u_1 \oplus \mathsf{Y}\, u_2$, avec $\mathsf{Y}\, x$ représentant la précédente valeur de la variable x. Afin de reconstruire la valeur de $\mathsf{Y}\, u_2$ à partir de $\mathsf{Y}\, u_1 \oplus \mathsf{Y}\, u_2$, le processus q_1 doit mémoriser la précédente valeur de u_1 et calculer le XOR de cette valeur avec la valeur de la variable t :

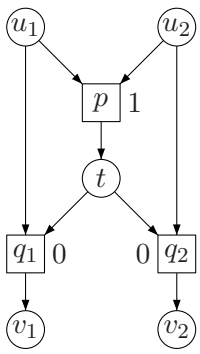


FIG. 3.6 – Une architecture uniformément bien connectée

$\mathsf{Y}\, u_2 = \mathsf{Y}\, u_1 \oplus t$. On montre que si on restreint les fonctions de décodage aux stratégies sans mémoire, alors le processus q_1 ne peut plus retrouver la valeur de u_2 :

Soit $\Phi = f^t$ un routage, avec f^t fonction sans mémoire, et soient $(g^{u,v})_{v\in V_O, u\in \mathrm{Vue}(v)}$ des fonctions de décodage satisfaisant l'égalité (3.7). Remarquons tout d'abord que si $s_1 s_2 s_3 \in (S^V)^3$ est une Φ-séquence, alors s_3^t ne dépend que de $s_2^{\{u_1,u_2\}}$, puisque f^t est sans mémoire et compatible avec son délai. Par ailleurs, $f^t : S^{u_1,u_2} \to S^t$ ne peut être injective, donc on peut trouver une autre Φ-séquence $s_1 s'_2 s'_3 \in (S^V)^3$ telle que $s_3^t = {s'_3}^t$ et $s_2^{\{u_1,u_2\}} \neq {s'_2}^{\{u_1,u_2\}}$ et $s_3^{\{u_1,u_2\}} = {s'_3}^{\{u_1,u_2\}}$. Si par exemple $s_2^{u_2} \neq {s'}_2^{u_2}$, on obtient

$$g^{u_2,v_1}((s_1 s_2 s_3)^{\{u_1,t\}}) = s_2^{u_2} \neq {s'}_2^{u_2} = g^{\{u_2,v_1\}}((s_1 s'_2 s'_3)^{\{u_1,t\}}).$$

Donc g^{u_2,v_1} ne peut pas être sans mémoire.

Cependant, il est intéressant de voir que si une architecture est uniformément bien connectée, les fonctions de décodage n'ont besoin que d'une mémoire *finie*, ce qui nous permet de prouver qu'il est décidable de vérifier qu'une architecture possède cette propriété, qu'on va appeler de *connexion uniforme*. En effet, il est important de pouvoir décider si une architecture vérifie cette condition car on peut montrer que le fait d'être à information linéairement préordonnée devient un critère nécessaire et suffisant de décidabilité pour le problème de SSD synchrone lorsqu'on se restreint aux architectures UWC. De plus, on peut obtenir la décidabilité de toute la classe des architectures UWC si l'on se restreint à certaines spécifications dites *robustes*.

Nous commençons par établir la décidabilité et la complexité de tester si une architecture est UWC, avant d'aborder les résultats obtenus sur cette classe pour la synthèse de systèmes synchrones.

3.2 Décider la connexion uniforme

On donne dans cette section une procédure permettant de vérifier si une architecture est UWC. On évalue sa complexité, et on donne une borne inférieure au problème.

3.2.1 Décidabilité

On montre tout d'abord que, comme annoncé, si une architecture est UWC avec un routage $\Phi = (f^v)_{v \in V \setminus (V_\mathrm{I} \cup V_\mathrm{O})}$ et des fonctions de décodage $(g^{u,v})_{v \in V_\mathrm{O}, u \in \mathrm{Vue}(v)}$, alors les fonctions de décodage sont en réalité des fonctions à mémoire finie. On dit qu'une fonction $f : X^* \to Y$ est *à mémoire finie* si elle peut être calculée par un automate déterministe avec sortie à nombre d'états fini. Dans un tel automate, toutes les exécutions sont acceptantes, et on y associe une fonction donnant une valeur dans Y en fonction de la transition courante de l'automate. Par la suite, on va noter pour tout $v \in V_\mathrm{O}$, g^v le tuple $(g^{u,v})_{u \in \mathrm{Vue}(v)}$ de fonctions de décodages associées à v. On observe que, puisque les fonctions de routage sont sans mémoire et compatibles avec les délais, la valeur d'une variable dans une séquence Φ-compatible n'est influencée que par un nombre limité de valeurs de variables d'entrée. Ceci est formalisé dans le lemme suivant.

Afin de définir précisément la fenêtre de valeurs influençant une variable donnée, on introduit des notations supplémentaires. En effet, on ne va pas seulement utiliser la notion de délai minimal de transmission de u à v – $d(u, v)$– mais aussi celle de délai maximal de transmission $D(u, v)$ définie par

$$
\begin{aligned}
&D(u,u) = 0\\
&D(u,v) = +\infty \text{ si } v \notin (E^2)^+(u)\\
&D(u,v) = d_v + \max\{D(u,w) \mid w \in R(v) \text{ et } w \in (E^2)^*(u)\} \text{ pour tout } u \neq v \in (E^2)^+(u).
\end{aligned}
$$

On rappelle que pour un mot $\sigma = s_1 s_2 \cdots \in \Sigma^\infty$, pour tous i, j entiers, on note $\sigma[i \cdots j]$ le facteur formé des lettres $s_i \cdots s_j$, qui est vide si $i > j$. On utilise aussi cette notation si $i \leq 0$ ou $j \leq 0$: si $j \leq 0$, alors $\sigma[i \cdots j] = \varepsilon$, et si $i \leq 0 < j$ alors $\sigma[i \cdots j] = \sigma[1 \cdots j]$.

Lemme 3.20. *Pour tout $v \in V \setminus V_O$, pour tout routage Φ et toute séquence Φ-compatible $\sigma = s_1 \cdots s_i \in (S^V)^+$, la valeur s_i^v ne dépend que de $(\sigma^u[i - D(u,v) \cdots i - d(u,v)])_{u \in Vue(v)}$.*

Démonstration. On rappelle que l'architecture est acyclique. On va donc procéder par induction sur les variables : les variables minimales sont les variables modifiées par l'environnement (V_I). Soit donc $v \in V_\mathrm{I}$. Alors $\mathrm{Vue}(v) = \{v\}$, $D(v,v) = d(v,v) = 0$ et trivialement, s_i^v ne dépend que de s_i^v (les variables de V_I étant modifiées de façon incontrôlable, elles ne dépendent que d'elles mêmes). Supposons à présent que $v \in V \setminus (V_\mathrm{I} \cup V_\mathrm{O})$. La séquence σ étant Φ-compatible, $s_i^v = f^v(\sigma^{R(v)})$. Les fonctions de routage étant sans mémoire, $f^v(\sigma^{R(v)})$ ne dépend que de $s_{i-d_v}^{R(v)}$. Soit $w \in R(v)$. Par hypothèse de récurrence, $s_{i-d_v}^w$ ne dépend que de $(\sigma^u[i - d_v - D(u,w) \cdots i - d_v - d(u,w)])_{u \in \mathrm{Vue}(w)}$. De plus $\mathrm{Vue}(v) = \bigcup_{w \in R(v)} \mathrm{Vue}(w)$. Donc, en utilisant les définitions de délais minimaux et maximaux de transmission, d et D, on déduit que s_i^v ne dépend que de $(\sigma^u[i - D(u,v) \cdots i - d(u,v)])_{u \in \mathrm{Vue}(v)}$. □

Notation 3.21. Pour tout $v \in V_\mathrm{O}$ et pour toute séquence Φ-compatible finie $\sigma = s_1 \cdots s_{|\sigma|} \in (S^{V \setminus V_\mathrm{O}})^*$, on définit

$$
\begin{aligned}
\Psi_v(\sigma) &= \left(\sigma^u[|\sigma| + 1 - D(u,v) + d_v \cdots |\sigma| - d(u,v) + d_v]\right)_{u \in \mathrm{Vue}(v)},\\
\overline{\Psi}_v(\sigma) &= \left(\sigma^u[1 \cdots |\sigma| - d(u,v) + d_v]\right)_{u \in \mathrm{Vue}(v)}.
\end{aligned}
$$

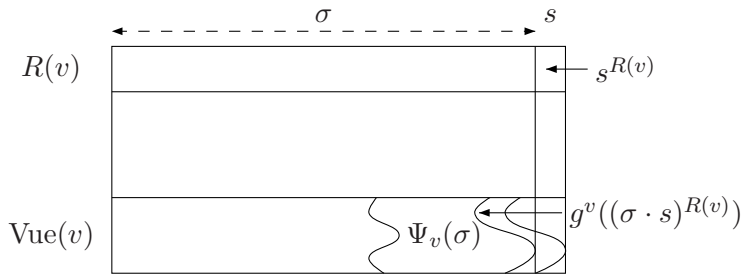


FIG. 3.7 – Valeurs à mémoriser pour décoder $s^{\mathrm{Vue}(v)}$

La proposition suivante montre que, pour tout $v \in V_{\mathrm{O}}$, $\Psi_v(\sigma)$ contient toute l'information nécessaire pour décoder les valeurs des variables dans $\mathrm{Vue}(v)$.

On fixe une variable de sortie $v \in V_{\mathrm{O}}$.

Proposition 3.22. *Soient $\sigma \cdot s$ et $\sigma' \cdot s'$ deux séquences Φ-compatibles telles que $\sigma, \sigma' \in (S^{V \setminus V_O})^*$ et $s, s' \in S^{V \setminus V_O}$ vérifient $\Psi_v(\sigma) = \Psi_v(\sigma')$ et $s^{R(v)} = s'^{R(v)}$. Alors $g^v((\sigma \cdot s)^{R(v)}) = g^v((\sigma' \cdot s')^{R(v)})$ et $\Psi_v(\sigma \cdot s) = \Psi_v(\sigma' \cdot s')$.*

Démonstration. On écrit $\sigma \cdot s = s_1 \cdots s_{|\sigma|} s_{|\sigma|+1}$ et $\sigma' \cdot s' = s'_1 \cdots s'_{|\sigma'|} s'_{|\sigma'|+1}$. Il est clair que $\sigma \cdot s$ et $\sigma' \cdot s'$ ne sont pas nécessairement de la même longueur. On a donc en général $(\sigma \cdot s)^{R(v)} \neq (\sigma' \cdot s')^{R(v)}$. On va donc construire une autre séquence Φ-compatible, $\sigma'' \cdot s''$ telle que, d'une part, $(\sigma \cdot s)^{R(v)} = (\sigma'' \cdot s'')^{R(v)}$ et, d'autre part, $g^v((\sigma' \cdot s')^{R(v)}) = g^v((\sigma'' \cdot s'')^{R(v)})$. Cette séquence $\sigma'' \cdot s''$ est telle que, pour tout $u \in \mathrm{Vue}(v)$,

$$\sigma''^u = \sigma^u[1 \cdots |\sigma| - d(u,v) + d_v] \cdot \sigma'^u[|\sigma'| + 1 - d(u,v) + d_v \cdots |\sigma'|]$$

et $s''^u = s'^u$. On fixe arbitrairement les valeurs des variables d'entrée qui ne sont pas dans la vue de v. On affirme qu'une telle séquence vérifie $(\sigma'' \cdot s'')^{R(v)} = (\sigma \cdot s)^{R(v)}$. Avant de le démontrer formellement, on montre comment cette égalité permet de prouver la proposition.

Comme $\sigma' \cdot s'$ et $\sigma'' \cdot s''$ sont deux séquences Φ-compatibles, pour chaque $u \in \mathrm{Vue}(v)$, on a

$$g^{u,v}((\sigma' \cdot s')^{R(v)}) = s'^u_{|\sigma'|+1-d(u,v)+d_v}$$
$$g^{u,v}((\sigma'' \cdot s'')^{R(v)}) = s''^u_{|\sigma''|+1-d(u,v)+d_v}$$

et, par définition de σ'', $s''^u_{|\sigma''|+1-d(u,v)+d_v} = s'^u_{|\sigma'|+1-d(u,v)+d_v}$. En utilisant l'affirmation faite ci-dessus que $(\sigma'' \cdot s'')^{R(v)} = (\sigma \cdot s)^{R(v)}$, on en déduit que $g^{u,v}((\sigma' \cdot s')^{R(v)}) = g^{u,v}((\sigma \cdot s)^{R(v)})$. Donc, on a bien $g^v((\sigma \cdot s)^{R(v)}) = g^v((\sigma' \cdot s')^{R(v)})$. En utilisant de plus le fait que $\Psi_v(\sigma) = \Psi_v(\sigma')$, on obtient que $\Psi_v(\sigma \cdot s) = \Psi_v(\sigma' \cdot s')$, ce qui prouve la proposition.

On démontre à présent que $(\sigma'' \cdot s'')^{R(v)} = (\sigma \cdot s)^{R(v)}$. Observons en premier lieu que $|\sigma''| = |\sigma|$. Par définition de σ'', on a $\overline{\Psi}_v(\sigma'') = \overline{\Psi}_v(\sigma)$. Soit $w \in R(v)$ et $i \leq |\sigma| = |\sigma''|$. Pour tout $u \in \mathrm{Vue}(w) \subseteq \mathrm{Vue}(v)$, on a $i - d(u,w) \leq |\sigma| - d(u,v) + d_v$. En utilisant le lemme 3.20, et le fait que $\overline{\Psi}_v(\sigma'') = \overline{\Psi}_v(\sigma)$, on déduit que $s''^w_i = s^w_i$. Ainsi, $\sigma''^{R(v)} = \sigma^{R(v)}$.

Il reste à montrer que $s''^{R(v)} = s^{R(v)}$. En utilisant l'égalité $\Psi_v(\sigma) = \Psi_v(\sigma')$ et la définition de $\sigma'' \cdot s''$, on obtient, pour chaque $u \in \mathrm{Vue}(v)$,

$$\begin{aligned}(\sigma'' \cdot s'')^u[|\sigma'' \cdot s''| - D(u,v) + d_v \cdots |\sigma'' \cdot s''| - d(u,v) + d_v] \\ = (\sigma' \cdot s')^u[|\sigma' \cdot s'| - D(u,v) + d_v \cdots |\sigma' \cdot s'| - d(u,v) + d_v].\end{aligned}$$

Soit $w \in R(v)$. Pour tout $u \in \mathrm{Vue}(w)$, on a $D(u,w) \leq D(u,v) - d_v$ et $d(u,w) \geq d(u,v) - d_v$. En utilisant le lemme 3.20, on en déduit que $s''^w = s'^w$. Ceci étant vrai pour tout $w \in R(v)$, $s''^{R(v)} = s'^{R(v)}$. En utilisant l'hypothèse que $s'^{R(v)} = s^{R(v)}$, on obtient que $s''^{R(v)} = s^{R(v)}$, ce qui conclut la démonstration. □

On définit maintenant un automate déterministe avec sortie $\mathcal{B}_v = (Q_v, q_0^v, S^{R(v)}, \delta_v, \alpha^v)$ qui calcule g^v avec une mémoire finie :

- $Q_v = \{\psi_v(\sigma) \mid \sigma \text{ est une séquence finie } \Phi\text{-compatible}\}$, l'ensemble fini d'états, avec $q_0^v = \Psi_v(\varepsilon)$ l'état initial.
- $S^{R(v)}$ l'alphabet d'entrée.
- $\delta_v : Q_v \times S^{R(v)} \to Q_v$ est la fonction de transition définie par

$$\delta_v(\Psi_v(\sigma), s^{R(v)}) = \begin{cases} \Psi_v(\sigma \cdot s) & \text{si } \sigma \cdot s \text{ est une séquence } \Phi\text{-compatible} \\ \text{indéfinie} & \text{sinon.} \end{cases}$$

  Par la proposition 3.22, δ_v est correctement définie. On peut déduire immédiatement par récurrence que $\delta_v(q_0^v, \sigma^{R(v)}) = \Psi_v(\sigma)$ pour tout σ, séquence finie Φ-compatible.
- $\alpha^v : Q_v \times S^{R(v)} \to S^{\mathrm{Vue}(v)}$ est la fonction de sortie définie par

$$\alpha^v(\Psi_v(\sigma), s^{R(v)}) = \begin{cases} g^v((\sigma \cdot s)^{R(v)}) & \text{si } \sigma \cdot s \text{ est une séquence } \Phi\text{-compatible} \\ 0 & \text{sinon.} \end{cases}$$

  Par la proposition 3.22, α est également bien définie. Pour toute séquence Φ-compatible $\sigma \cdot s$, on a

$$g^v((\sigma \cdot s)^{R(v)}) = \alpha^v(\Psi_v(\sigma), s^{R(v)}) = \alpha^v(\delta_v(q_0^v, \sigma^{R(v)}), s^{R(v)}).$$

L'automate fini déterministe $\mathcal{B}_v$ calcule donc bien g^v.

Remarque 3.23. Si l'architecture est sans délai, alors Q_v est un singleton et g^v est sans mémoire.

Par la suite, on va noter $D = \max\{D(u,v) - d_v \mid v \in V_{\mathrm{O}}, u \in \mathrm{Vue}(v)\}$ et on va fixer un routage $\Phi = (f^v)_{v \in V \setminus (V_{\mathrm{I}} \cup V_{\mathrm{O}})}$.

Notation 3.24. Pour toute séquence Φ-compatible finie $\sigma \in (S^{V \setminus V_{\mathrm{O}}})^*$, on note $\mathrm{Suff}_\Phi(\sigma)$ la séquence Φ-compatible de longueur D induite par $\rho = \sigma^{V_{\mathrm{I}}}[|\sigma| + 1 - D \cdots |\sigma|] \in (S^{V_{\mathrm{I}}})^D$.

Remarque 3.25. L'automate $\mathcal{B}_v$ vérifie $\delta_v(q_0^v, \sigma^{R(v)}) = \delta_v(q_0^v, (\mathrm{Suff}_\Phi(\sigma))^{R(v)})$ pour toute séquence Φ-compatible finie. En effet, comme, par définition, $\Psi_v(\sigma) = \Psi_v(\mathrm{Suff}_\Phi(\sigma))$, alors $\delta_v(q_0^v, \sigma^{R(v)}) = \Psi_v(\sigma) = \Psi_v(\mathrm{Suff}_\Phi(\sigma)) = \delta_v(q_0^v, (\mathrm{Suff}_\Phi(\sigma))^{R(v)})$.

Pour prouver qu'il est décidable de vérifier qu'un automate avec sortie calcule bien la fonction de décodage recherchée, on doit d'abord montrer les résultats intermédiaires suivants, assurant qu'on peut vérifier les propriétés recherchées en temps fini.

Lemme 3.26. *Soit $\mathcal{C} = (Q, q_0, S^{R(v)}, \delta, \alpha)$ un automate fini avec sortie, et considérons la propriété $P(\sigma)$ sur les séquences Φ-compatibles finies $\sigma \in (S^{V \setminus V_{\mathrm{O}}})^*$ définie par*

$$\delta(q_0, \sigma^{R(v)}) = \delta(q_0, (\mathrm{Suff}_\Phi(\sigma))^{R(v)}) \qquad P(\sigma)$$

Alors $P(\sigma)$ est vraie pour toute séquence Φ-compatible finie, si et seulement si $P(\sigma)$ est vraie pour toute séquence Φ-compatible de longueur $D+1$.

Démonstration. Supposons que $P(\sigma)$ soit vraie pour toute séquence Φ-compatible de longueur $D+1$. Soit σ une séquence Φ-compatible arbitraire. On montre que $P(\sigma)$ est vraie par récurrence sur la longueur de σ. Si $|\sigma| \leq D$, alors $\mathrm{Suff}_\Phi(\sigma) = \sigma$ et $P(\sigma)$ est vraie. Supposons à présent que $|\sigma| = k+1$ avec $k \geq D$. On écrit $\sigma = s_1 \cdots s_{k+1}$ avec $s_i \in S^{V \setminus V_O}$. On sait que

$$\begin{aligned}\delta(q_0, \sigma^{R(v)}) &= \delta(\delta(q_0, \sigma[k]^{R(v)}), s_{k+1}^{R(v)}) \\ &= \delta(\delta(q_0, \mathrm{Suff}_\Phi(\sigma[k])^{R(v)}), s_{k+1}^{R(v)}) \text{ par hypothèse de récurrence} \\ &= \delta(q_0, (\mathrm{Suff}_\Phi(\sigma[k]) \cdot s_{k+1})^{R(v)}).\end{aligned}$$

Considérons maintenant $\sigma' = s'_1 \cdots s'_{D+1} \in (S^{V \setminus V_O})^{D+1}$ la séquence Φ-compatible telle que $\sigma'^{V_I} = \sigma^{V_I}[|\sigma| - D \ldots |\sigma|]$. On déduit de la notation 3.24 que $\mathrm{Suff}_\Phi(\sigma[k]) = \sigma'[D]$. De plus, pour $w \in R(v)$ et $u \in \mathrm{Vue}(v)$, on sait que $D(u,w) \leq D(u,v) - d_v \leq D$, et le lemme 3.20 implique que $s'^{R(v)}_{D+1} = s^{R(v)}_{k+1}$. Ainsi, $(\mathrm{Suff}_\Phi(\sigma[k]) \cdot s_{k+1})^{R(v)} = \sigma'^{R(v)}$. Comme $|\sigma'| = D+1$, alors $P(\sigma')$ est vraie, ce qui implique que

$$\delta(q_0, (\mathrm{Suff}_\Phi(\sigma[k]) \cdot s_{k+1})^{R(v)}) = \delta(q_0, \sigma'^{R(v)}) = \delta(q_0, \mathrm{Suff}_\Phi(\sigma')^{R(v)}).$$

En observant que $\mathrm{Suff}_\Phi(\sigma') = \mathrm{Suff}_\Phi(\sigma)$, on obtient bien $P(\sigma)$. □

Le lemme suivant montre que l'on peut vérifier en temps fini qu'un automate avec sortie calcule la fonction de décodage correcte.

Lemme 3.27. *Soit $\mathcal{C} = (Q, q_0, S^{R(v)}, \delta, \alpha)$ un automate fini avec sortie tel que $P(\sigma)$ est vraie pour toute séquence Φ-compatible finie. Considérons la propriété $P'(\sigma)$ des séquences Φ-compatibles finies $\sigma = s_1 \cdots s_{k+1} \in (S^{V \setminus V_O})^*$ définie par*

$$\alpha(\delta(q_0, \sigma^{R(v)}[k]), s_{k+1}^{R(v)}) = (s^u_{k+1-d(u,v)+d_v})_{u \in Vue(v)} \qquad P'(\sigma)$$

avec la convention $s_i^u = 0$, pour $i \leq 0$.

Alors $P'(\sigma)$ est vraie pour toutes les séquences Φ-compatibles finies si et seulement si $P'(\sigma)$ est vraie pour toutes les séquences Φ-compatibles de taille inférieure ou égale à $D+1$.

Démonstration. Supposons $P'(\sigma)$ vraie pour toute séquence Φ-compatible de taille inférieure ou égale à $D+1$. Soit $\sigma = s_1 \cdots s_{k+1} \in (S^{V \setminus V_O})^*$ une séquence Φ-compatible avec $k > D$. Comme $P(\sigma[k])$ est vraie, on obtient

$$\alpha(\delta(q_0, \sigma^{R(v)}[k]), s_{k+1}^{R(v)}) = \alpha(\delta(q_0, \mathrm{Suff}_\Phi(\sigma[k])^{R(v)}), s_{k+1}^{R(v)}).$$

Comme dans la preuve du lemme 3.26, on pose $\sigma' = s'_1 \cdots s'_{D+1} \in (S^{V \setminus V_O})^{D+1}$ comme étant la séquence Φ-compatible telle que $\sigma'^{V_I} = \sigma^{V_I}[|\sigma| - D \cdots |\sigma|]$. On a vu que $s^{R(v)}_{k+1} = s'^{R(v)}_{D+1}$ et $\mathrm{Suff}_\Phi(\sigma[k]) = \sigma'[D]$. Donc

$$\alpha(\delta(q_0, \mathrm{Suff}_\Phi(\sigma[k])^{R(v)}), s_{k+1}^{R(v)}) = \alpha(\delta(q_0, \sigma'[D]), s'^{R(v)}_{D+1}).$$

Comme $|\sigma'| = D+1$, on sait que $P'(\sigma')$ est vraie et on obtient que

$$\alpha(\delta(q_0, \sigma'[D]), s'^{R(v)}_{D+1}) = (s'^u_{D+1-d(u,v)+d_v})_{u \in \mathrm{Vue}(v)}.$$

De plus, par définition de σ', on a pour tout $u \in \mathrm{Vue}(v)$, $s'^u_{D+1-d(u,v)+d_v} = s^u_{k+1-d(u,v)+d_v}$. On en conclut que $P'(\sigma)$ est vraie. □

3.2.2 Complexité

On démontre à présent que vérifier qu'une architecture est UWC est décidable et on établit la complexité de la procédure.

Proposition 3.28. *Le problème de vérifier si une architecture donnée est UWC est décidable. De plus, ce problème est*

1. *dans NP si on se restreint aux architectures*
   - *sans délai,*
   - *pour lesquelles le domaine des variables est borné : $|S^v| \leq c_s$, pour tout $v \in V$, où c_s est une constante indépendante des données du problème,*
   - *pour lesquelles le nombre de variables lues par un processus est borné : $|R(v)| \leq c_r$, pour tout $v \in V$, où c_r est une constante indépendante des données du problème.*
2. *dans NEXPTIME si les délais sont bornés par une constante, i.e., $d_v \leq c_d$ pour tout $v \in V \setminus V_I$, où c_d est une constante qui ne dépend pas des données du problème.*
3. *dans 2-NEXPTIME si on ne pose pas d'hypothèse sur l'architecture donnée*

Démonstration. Considérons une architecture $\mathcal{A} = (\text{Proc}, V, E, (S^v)_{v\in V}, s_0, (d_p)_{p\in\text{Proc}})$. Pour vérifier si elle est UWC, on suit la procédure non-déterministe suivante :

- Deviner un routage $\Phi = (f^v)_{v\in V\setminus(V_\text{I}\cup V_\text{O})}$.
- Pour chaque variable de sortie $v \in V_\text{O}$, deviner un automate déterministe avec sortie $\mathcal{C} = (Q, q_0, S^{R(v)}, \delta, \alpha)$ tel que

$$|Q| \leq \Pi_{u\in\text{Vue}(v)}|S^u|^{D(u,v)-d(u,v)}$$

  et, pour toute séquence $\rho \in (S^{V_\text{I}})^+$ de longueur $D+1$, calculer la séquence Φ-compatible $\sigma \in (S^{V\setminus V_\text{O}})^+$ telle que $\sigma^{V_\text{I}} = \rho$ et vérifier que $P(\sigma)$ est vraie et que $P'(\sigma[k+1])$ est vraie pour tout $k \leq D$.

En effet, si l'architecture est UWC par le routage Φ et le n-uplet de fonctions de décodage $(g^{u,v})_{v\in V_\text{O}, u\in\text{Vue}(v)}$, alors pour chaque variable de sortie $v \in V_\text{O}$ l'automate $\mathcal{B}_v$ défini après la proposition 3.22 satisfait les conditions ci-dessus, puisqu'il calcule les fonctions g^v et que, d'après la remarque 3.25, il vérifie $P(\sigma)$ pour toute séquence Φ-compatible.

Réciproquement, si l'on peut trouver un routage Φ et un automate $\mathcal{C}$ pour chaque variable de sortie $v \in V_\text{O}$ qui satisfait les conditions ci-dessus, alors par le lemme 3.27, on déduit que la fonction calculée par $\mathcal{C}$ satisfait bien l'égalité (3.7). L'architecture est donc bien UWC.

On étudie à présent la complexité de cette procédure de décision. Calculons tout d'abord la taille nécessaire à la mémorisation d'un routage Φ et d'un automate $\mathcal{C}$. Pour chaque variable $v \in V \setminus (V_\text{I} \cup V_\text{O})$, écrire une fonction sans mémoire $f^v : S^{R(v)} \to S^v$ nécessite une taille de $|S^{R(v)}| \cdot \log_2 |S^v|$. Donc,

$$|\Phi| \leq \sum_{v\in V\setminus(V_\text{I}\cup V_\text{O})} |S^{R(v)}| \cdot \log_2 |S^v|.$$

On remarque que, étant donnés Φ et $\rho \in (S^{V_\text{I}})^+$, on peut calculer la séquence Φ-compatible induite $\sigma \in (S^{V\setminus V_\text{O}})^+$ (c'est-à-dire vérifiant $\sigma^{V_\text{I}} = \rho$), ainsi que $\text{Suff}_\Phi(\sigma)$ en temps polynomial par rapport à $|\Phi| + |\rho|$.

Par ailleurs, pour chaque variable de sortie $v \in V_\text{O}$, et pour chaque $u \in \text{Vue}(v)$, on a $D(u,v) - d(u,v) \leq D(u,v) - d_v \leq D$. Ainsi, la taille de l'automate déterministe $\mathcal{C}$ est donnée

par $|\mathcal{C}| = |\delta| + |\alpha|$ où

$$|Q| \leq \prod_{u \in \mathrm{Vue}(v)} |S^u|^D = |S^{\mathrm{Vue}(v)}|^D$$
$$|\delta| \leq |Q| \cdot |S^{R(v)}| \cdot \log_2 |Q|$$
$$|\alpha| \leq |Q| \cdot |S^{R(v)}| \cdot \log_2 |S^{\mathrm{Vue}(v)}|$$

Étant donnés $\mathcal{C}$, une séquence σ Φ-compatible et $\mathrm{Suff}_\Phi(\sigma)$, on peut vérifier si $P(\sigma)$ et $P'(\sigma)$ sont vraies en temps polynomial par rapport à $|\mathcal{C}| + |\sigma|$.

Si on considère que le nombre de variables et de processus, la taille de chaque domaine S^v et la valeur de chaque délai d_p est donnée en binaire, la *taille* de l'architecture est donnée par

$$|\mathcal{A}| = \log_2 |V| + \log_2 |P| + |E| + \sum_{v \in V} \log_2 |S^v| + \sum_{p \in P} \log_2(1 + d_p).$$

On peut à présent se tourner vers les trois cas de la proposition 3.28 :

1. Dans ce cas, on a $\Phi| \leq |V| \cdot c_s^{c_r} \cdot \log_2(c_s) = \mathcal{O}(|\mathcal{A}|)$, puisque $|V| \leq |E| \leq |\mathcal{A}|$. De plus, $D = 0$ et $|Q| = 1$, donc il reste uniquement à deviner la fonction de décodage α telle que $|\alpha| = c_s^{c_r} \cdot c_r \cdot \log_2(c_s)$, qui est constante. Enfin, les seules séquences d'entrée $\rho \in (S^{V_\mathrm{I}})^+$ que l'on doit considérer sont celles de taille 1. On en déduit que notre algorithme non-déterministe fonctionne en temps polynomial.
2. Ici, l'hypothèse implique que $D = \mathcal{O}(|V|) = \mathcal{O}(\mathcal{A})$. De plus, comme $\log_2 |S^v| \leq |\mathcal{A}|$ pour tout $v \in V$, alors pour chaque sous-ensemble des variables $U \subseteq V$, on a $|S^U| \leq 2^{|U| \cdot |\mathcal{A}|}$. En utilisant le fait que $|V| \leq |E| \leq |\mathcal{A}|$, on déduit que $|S^U| \leq 2^{|\mathcal{A}|^2}$. De là, $|\Phi| = 2^{\mathcal{O}(|\mathcal{A}|^2)}$ et $|Q| = 2^{\mathcal{O}(|\mathcal{A}|^3)}$, et enfin $|\mathcal{C}| = |\delta| + |\alpha| = 2^{\mathcal{O}(|\mathcal{A}|^3)}$. De plus, le nombre de séquences d'entrée $\rho \in (S^{V_\mathrm{I}})^{D+1}$ à considérer dans notre algorithme est également dans $2^{\mathcal{O}(|\mathcal{A}|^3)}$. On en déduit que notre algorithme non-déterministe fonctionne en temps exponentiel.
3. Dans ce dernier cas, on peut seulement borner D par $2^{|\mathcal{A}|}$ et on obtient que notre algorithme non-déterministe fonctionne en temps doublement exponentiel.

□

On détermine à présent une borne inférieure pour la complexité de ce problème. Pour cela, on établit tout d'abord un lien avec le problème de *flux d'information dans un réseau* (network information flow) introduit dans [ACLY00]. Les instances de ce problème sont des graphes acycliques orientés dans lesquels deux sous-ensembles des nœuds ont été distingués : les *sources* et les *puits*. On se donne en plus d'un tel graphe un certain nombre de *messages*, et chaque puits demande un sous-ensemble de ces messages. Formellement, une instance du problème de flux d'information dans un réseau est un quintuplet $(P, M, E, S, \text{demande})$ dans lequel P est l'ensemble des processus, M est l'ensemble des messages, la relation $E \subseteq (P \times P) \cup (M \times P)$ définit l'ensemble des arcs du graphe et l'ensemble $V = E \cap (P \times P)$ correpond aux variables internes du réseau (données donc de façon implicite). Toutes les variables de $M \cup V$ ont le même domaine S. On dit qu'un processus est une source s'il est relié à un message d'entrée – l'ensemble des sources du réseau est donc $E(M)$. Enfin, la fonction demande : $P \to 2^M$ définit quels messages doivent être transmis à quels processus, On dit donc qu'un processus $p \in P$ est un puits si demande$(p) \neq \emptyset$.

Un problème spécifique a été particulièrement étudié dans ce domaine ; il s'agit du problème de *multicast*, dans lequel la donnée est une instance du problème défini ci-dessus, ayant

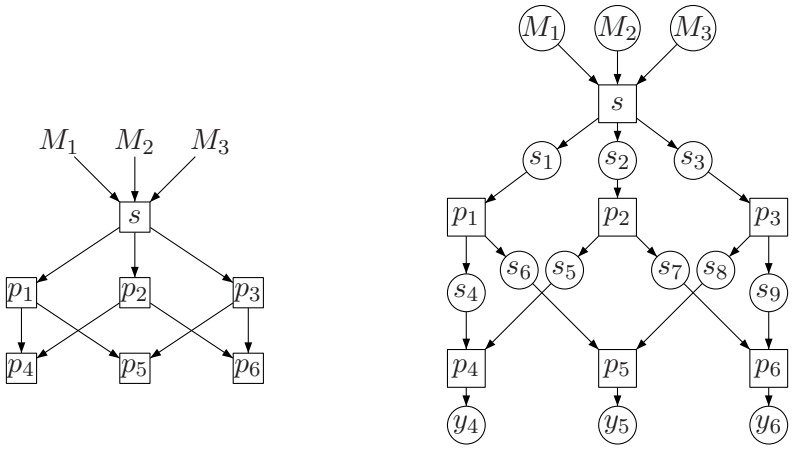


FIG. 3.8 – Une instance de *multicast* et l'architecture distribuée correspondante

une unique source, et dans laquelle tous les puits demandent tous les messages (un exemple est représenté dans la figure 3.8 dans laquelle les puits sont les processus p_4, p_5 et p_6).

Il est clair que les réseaux que l'on vient de décrire sont très proches des architectures que l'on considère ici. Les différences résident principalement dans les aspects suivants. Tout d'abord, une variable d'un réseau est attachée à un arc, ce qui implique qu'une variable ne peut être lue que par un seul processus quand dans notre cas plusieurs processus peuvent lire la même variable. Ensuite, le domaine des variables est uniforme pour toutes les variables d'un réseau dans le problème de flux d'information alors qu'on autorise des domaines de tailles différentes pour les variables de nos architectures. Enfin, les messages transmis dans les problèmes de flux d'information dans un réseau le sont sans délai, alors qu'on autorise des délais arbitraires pour les processus. Donc, les architectures considérées dans le problème de synthèse que l'on étudie sont plus générales, et on obtient le résultat suivant :

Lemme 3.29. *Le problème du multicast se réduit en temps polynomial au problème de connexion uniforme.*

Démonstration. Soit $\mathcal{A} = (P, M, E, S, \text{demande})$ une instance du problème du multicast. Il existe un processus $p_0 \in P$, unique source du réseau, et vérifiant donc $\{p_0\} = E(M)$. On définit une architecture $\mathcal{A} = (\text{Proc}', V', E', (S^v)_{v\in V'}, s_0, (d_p)_{p\in \text{Proc}'})$ par

$$
\begin{aligned}
\text{Proc}' &= P \\
V' &= {V_\text{I}}' \uplus {V_\text{O}}' \uplus V \text{ avec} \\
{V_\text{I}}' &= M \text{ et} \\
{V_\text{O}}' &= \{s_p \mid p \in P \text{ et demande}(p) = M\} \\
E' &= M \times \{p_0\} \cup \{(p, s_p) \mid \text{demande}(p) = M\} \cup \bigcup_{v=(p,q)\in V} \{(p, v), (v, q)\} \\
S^v &= S \text{ pour tout } v \in V', \\
s_0 &= (0)_{v\in V'} \\
d_p &= 0 \text{ pour tout } p \in \text{Proc}'.
\end{aligned}
$$

Une solution au problème de flux d'information dans un réseau est un tuple de fonctions $(f^{p,q})_{(p,q)\in E}$ telles que $f^{p,q} : S^{E^{-1}(p)} \to S$ et des fonctions de décodage $(g^p)_{p\in \text{demande}^{-1}(M)}$ telles que $g^p : S^{E^{-1}(p)} \to S^M$. Si on pose $V = E \cap (P \times P)$, une lettre $s \in S^{V\cup M}$ est compatible avec le routage $(f^{p,q})_{(p,q)\in E}$ si pour tout $(p,q) \in V$, $s^{p,q} = f^{p,q}(s^{R(p,q)})$ où $R(p,q) = \{(p',p) \mid (p',p) \in E\}$. Pour toute lettre $s \in S^{V\cup M}$, compatible avec $(f^v)_{v\in V}$, on veut que $g^p(s^{E^{-1}(p)}) = s^M$.

Pour tout $v \in {V_\text{O}}'$, $\text{Vue}(v) = {V_\text{I}}' = M$, et pour toute variable interne $v \in V$, $R'(v) = R(v)$. Donc la notion de routage sur $\mathcal{A}$ correpond à la notion de routage sur $\mathcal{A}'$.

Par ailleurs, on a vu dans la section 3.2.1 que, s'il existe des fonctions de décodage pour un routage Φ sur $\mathcal{A}'$, alors ces fonctions ont une mémoire finie. On a aussi démontré que, pour $v \in {V_\text{O}}'$, la mémoire $Q_v = \{\Psi_v(\sigma) \mid \sigma \text{ est une séquence } \Phi\text{-compatible}\}$ est suffisante pour les fonctions de décodage $g^v = (g^{u,v})_{u\in \text{Vue}(v)}$. Mais, quand tous les délais valent 0, $|Q_v| = 1$, ce qui signifie que les fonctions g^v sont sans mémoire. Ainsi, on peut réécrire la condition (3.7) de la définition 3.18 en $s_i^{\text{Vue}(v)} = g^v(s_i^{R(v)})$, et la notion de fonction de décodage pour $\mathcal{A}$, fonctions sans mémoire par définition, coïncide avec la notion de fonction de décodage pour $\mathcal{A}'$.

Ceci nous permet de conclure que le problème de multicast pour $\mathcal{A}$ coïncide avec le problème de connexion uniforme pour $\mathcal{A}'$. □

Rasala Lehman et Lehman [RLL04] ont montré que le problème de multicast dans le cas où la taille de S est $q = p^k$ avec p nombre premier, est NP-dur. Par le lemme 3.29, on obtient :

Corollaire 3.30. *Le problème de connexion uniforme est NP-dur.*

En fait, on déduit de [RLL04] que le problème du multicast restreint au cas où la taille de l'alphabet est fixée et égale à 2, le degré entrant est fixé et tel que $E^{-1}(v) \leq 2$ pour tout nœud v du graphe, est aussi NP-dur. En utilisant les mêmes arguments que dans la preuve du lemme 3.29, on obtient donc

Corollaire 3.31. *Le problème de connexion uniforme pour une architecture telle que :*
- *le délai $d_p = 0$ pour tout processus p,*
- *la taille du domaine des variables est fixée, i.e., $|S^v| \leq c_s$ pour toutes les variables v, avec c_s une constante qui ne dépend pas de la donnée du problème,*
- *le degré de lecture est fixé, i.e., $|R(v)| \leq c_r$ pour toutes les variables $v \in V \setminus (V_I \cup V_O)$, avec $c_r \geq 2$ une constante qui ne dépend pas de la donnée du problème*

est NP-complet.

3.3 Le problème de SSD synchrone pour les architectures UWC

On prouve à présent que le problème de SSD synchrone est décidable pour $(\mathcal{A}, \varphi)$, avec $\mathcal{A}$ architecture UWC et φ spécification externe, si et seulement si $\mathcal{A}$ est à information linéairement préordonnée.

Nous allons commencer par montrer qu'il est en quelque sorte plus facile de trouver des programmes distribués sur une architecture UWC. En effet, le routage calculé pour transmettre les valeurs des entrées vers les sorties peut être vu comme une collection de stratégies locales pour les variables internes du système. Ainsi, pour définir une stratégie distribuée, il suffit de définir un tuple de stratégies reliant les variables de sortie aux variables d'entrée, compatibles avec les délais des processus de l'architecture.

Lemme 3.32. *Soit $\mathcal{A} = (\mathrm{Proc}, V, E, (S^v)_{v\in V}, s_0, (d_p)_{p\in\mathrm{Proc}})$ une architecture UWC. Pour chaque variable $v \in V_O$, soit $h^v : (S^{Vue(v)})^+ \to S^v$ une application compatible avec les délais. Alors il existe un programme distribué $F = (f^v)_{v\in V\setminus V_I}$ pour l'architecture $\mathcal{A}$ tel que $h^v = \hat{f}^v$ pour tout $v \in V_O$.*

Démonstration. Soient $\Phi = (f^v)_{v\in V\setminus V_\mathrm{I}\cup V_\mathrm{O}}$ et $(g^{u,v})_{v\in V_\mathrm{O}, u\in\mathrm{Vue}(v)}$ respectivement un routage et les fonctions de décodage permettant de vérifier l'égalité (3.7) de la définition 3.18. Comme annoncé, on utilise les fonctions de routage f^v comme stratégies sans mémoire pour les variables $v \in V \setminus (V_\mathrm{I} \cup V_\mathrm{O})$. Il reste à définir f^v pour $v \in V_\mathrm{O}$. Soit $\rho \in (S^{V_\mathrm{I}})^i$ pour $i > 0$ et soit $\sigma \in (S^{V\setminus V_\mathrm{O}})^i$ la séquence Φ-compatible correspondante. Pour $v \in V_\mathrm{O}$, on va définir f^v par $f^v(\sigma^{R(v)}) = h^v(\rho^{\mathrm{Vue}(v)})$. Montrons que de cette façon, la stratégie est bien définie.

Soient $i > 0$ et $\rho, \rho' \in (S^{V_\mathrm{I}})^i$. Soient $\sigma, \sigma' \in (S^{V\setminus V_\mathrm{O}})^i$ les séquences Φ-compatibles correspondantes, et supposons que $\sigma^{R(v)}[i - d_v] = \sigma'^{R(v)}[i - d_v]$. Alors, pour tout $u \in \mathrm{Vue}(v)$, $\rho^u[i - d(u, v)] = \rho'^u[i - d(u, v)]$. En effet, pour tout $0 \le j \le i - d(u, v)$, l'égalité (3.7) assure que $s^u_j = g^{u,v}(\sigma^{R(v)}[j + d(u, v) - d_v])$ et $s'^u_j = g^{u,v}(\sigma'^{R(v)}[j + d(u, v) - d_v])$. En utilisant le fait que $\sigma^{R(v)}[i - d_v] = \sigma'^{R(v)}[i - d_v]$ et $j + d(u, v) \le i$, on obtient bien $s^u_j = s'^u_j$. Comme h^v est d-compatible, on en déduit que $h^v(\rho^{\mathrm{Vue}(v)}) = h^v(\rho'^{\mathrm{Vue}(v)})$ et f^v est donc bien définie.

Ainsi, pour $\tau \in (S^{R(v)})^i$ avec $i > 0$, on pose

$$f^v(\tau) = \begin{cases} h^v(\sigma^{\mathrm{Vue}(v)}) & \text{s'il existe } \sigma \text{ une séquence } \Phi\text{-compatible telle que} \\ & \tau[i - d_v] = \sigma^{R(v)}[i - d_v] \\ 0 & \text{sinon.} \end{cases}$$

On a montré que f^v était bien définie et elle est de plus d-compatible (il est facile de voir qu'elle ne dépend que de $\tau[i - d_v]$). Soit $\rho \in (S^{V_\mathrm{I}})^+$ et soit σ l'exécution respectant F induite par ρ. Par définition des résumés (donnée page 63), on obtient que $\hat{f}^v(\rho^{\mathrm{Vue}(v)}) = f^v(\sigma^{R(v)})$. Comme $\sigma^{V\setminus V_\mathrm{O}}$ est aussi une séquence Φ-compatible pour ρ, on obtient, par définition de f^v, $\hat{f}^v(\rho^{\mathrm{Vue}(v)}) = f^v(\sigma^{R(v)}) = h^v(\sigma^{\mathrm{Vue}(v)})$. □

On donne maintenant un critère de décidabilité du problème de SSD synchrone pour cette sous-classe particulière d'architectures.

Théorème 3.33. *Le problème de SSD synchrone est décidable pour les instances $(\mathcal{A}, \varphi)$ où $\mathcal{A}$ est une architecture UWC et φ une spécification externe (linéaire ou branchante) si et seulement si $\mathcal{A}$ est à information linéairement pré-ordonnée.*

On a vu dans la proposition 3.14 qu'une architecture à information incomparable est indécidable pour les spécifications externes de LTL et CTL. On prouve à présent que, pour la sous-classe des architectures UWC, c'est une condition *nécessaire* d'indécidabilité.

On fixe jusqu'à la fin de la section une architecture $\mathcal{A} = (\mathrm{Proc}, V, E, (S^v)_{v\in V}, s_0, (d_p)_{p\in\mathrm{Proc}})$ qu'on suppose UWC et à information linéairement préordonnée. On va ordonner les variables de sortie $V_\mathrm{O} = \{v_1, \ldots, v_n\}$ de façon à ce que $\mathrm{Vue}(v_n) \subseteq \cdots \subseteq \mathrm{Vue}(v_1) \subseteq V_\mathrm{I}$.

Pour prouver le théorème, nous allons utiliser des automates d'arbres. On va donc étendre une stratégie locale $f : (S^X)^+ \to S^Y$ en posant $f(\varepsilon) = s^Y_0$, et la considérer comme un (S^X, S^Y)-arbre *complet*. Jusqu'à la fin du chapitre, nous ne considérerons que des arbres complets, que l'on nommera simplement *arbres*. Pour déterminer s'il existe une stratégie distribuée pour une spécification donnée, on procède en deux temps. Tout d'abord, on construit

un automate acceptant *toutes* les stratégies calculant les sorties en fonction des entrées qui satisfont la spécification. On ne se restreint donc tout d'abord pas aux stratégies distribuées, ni aux stratégies compatibles avec les délais. De plus, on ne considère que des stratégies calculant uniquement les valeurs des variables de sortie du système, à l'exclusion des variables internes. Une telle stratégie est vue comme un $(S^{\mathrm{Vue}(v_1)}, S^{V_O})$-arbre h vérifiant $h(\varepsilon) = s_0^{V_O}$. En fait, cette première étape revient à faire abstraction des processus internes de $\mathcal{A}$, vus comme une boîte noire, et à ne considérer que des processus « abstraits » lisant les variables d'entrée et modifiant les variables de sortie. La preuve de ce théorème repose sur le résultat suivant :

Proposition 3.34 ([KV99]). *Étant donnée une spécification externe $\varphi \in \mathsf{CTL}^*(V_I \cup V_O)$, on peut construire un automate d'arbre non-déterministe (AAND) $\mathfrak{A}_1$ sur des $(S^{Vue(v_1)}, S^{V_O})$-arbres tel que $h \in \mathcal{L}(\mathfrak{A}_1)$ si et seulement si l'arbre des exécutions de h, $t_h : (S^{V_I})^* \to S^{V_I \cup V_O}$, satisfait φ.*

Si $\mathcal{L}(\mathfrak{A}_1)$ est vide, on peut conclure qu'il n'existe aucune stratégie gagnante pour $(\mathcal{A}, \varphi)$. Sinon, d'après le lemme 3.32, il reste à vérifier si, pour chaque $v \in V_{\mathrm{O}}$, il existe un $(S^{\mathrm{Vue}(v)}, S^v)$-arbre h^v d-compatible, et tel que la stratégie globale $\bigoplus_{v \in V_{\mathrm{O}}} h^v$ induite par la collection $(h^v)_{v \in V_{\mathrm{O}}}$ est acceptée par $\mathfrak{A}_1$. Formellement, si $X = X_1 \cup X_2 \subseteq V_{\mathrm{I}}$ et $Y = Y_1 \uplus Y_2 \subseteq V_{\mathrm{O}}$, pour $i = 1, 2$, on a h_i un (S^{X_i}, S^{Y_i})-arbre. Alors $h = h_1 \oplus h_2$ est un (S^X, S^Y)-arbre tel que $h(\sigma) = (h_1(\sigma^{X_1}), h_2(\sigma^{X_2}))$ pour tout $\sigma \in (S^X)^*$. La stratégie h est en fait la stratégie distribuée sur l'architecture composée de deux processus, écrivant respectivement sur Y_1 et Y_2 et ayant pour variables d'entrée respectivement X_1 et X_2.

Pour vérifier l'existence de tels arbres $(h^v)_{v \in V_{\mathrm{O}}}$, on va les calculer en considérant les variables de sortie une par une, en suivant l'ordre $v_1, \ldots, v_n$. Il est important de commencer par la variable ayant la plus large vue des variables d'entrée, même si, de par les délais éventuels de l'architecture, elle reçoit réellement l'information plus tard que d'autres variables de sortie.

Pour tout $k \geq 1$, on note $V_k = \{v_k, \cdots, v_n\}$, et on procède récursivement. Le raisonnement repose sur la proposition suivante.

Proposition 3.35. *Soit $1 \leq k < n$. Soit un AAND $\mathfrak{A}_k$ acceptant des $(S^{Vue(v_k)}, S^{V_k})$-arbres. On peut construire un AAND $\mathfrak{A}_{k+1}$ acceptant des $(S^{Vue(v_{k+1})}, S^{V_{k+1}})$-arbres tel qu'un arbre t est accepté par $\mathfrak{A}_{k+1}$ si et seulement si il existe h^{v_k}, $(S^{Vue(v_k)}, S^{v_k})$-arbre d-compatible tel que $h^{v_k} \oplus t \in \mathcal{L}(\mathfrak{A}_k)$.*

La preuve de la proposition 3.35 se fait en deux étapes. Comme $V_k = \{v_k\} \uplus V_{k+1}$, tout $(S^{\mathrm{Vue}(v_k)}, S^{V_k})$-arbre t est de la forme $t = t^{v_k} \oplus t^{V_{k+1}}$ (avec t^U la projection de t sur U). Donc on peut d'abord transformer l'automate $\mathfrak{A}_k$ en un automate $\mathfrak{A}'_k$ acceptant les arbres $t \in \mathcal{L}(\mathfrak{A}_k)$ tels que t^{v_k} est d-compatible. Ceci est fait dans le lemme 3.36. Ensuite, il reste à construire un automate qui restreint le domaine des directions ainsi que l'étiquetage des arbres acceptés par $\mathfrak{A}'_k$ afin d'obtenir des $(S^{\mathrm{Vue}(v_{k+1})}, S^{V_{k+1}})$-arbres.

Lemme 3.36. *Soit $v \in U \subseteq V_O$ et soit un AAND $\mathfrak{A}$ acceptant des $(S^{Vue(v)}, S^U)$-arbres. On peut construire un AAND $\mathfrak{A}' = \mathrm{compat}_v(\mathfrak{A})$ acceptant des $(S^{Vue(v)}, S^U)$-arbres et tel que $\mathcal{L}(\mathfrak{A}') = \{t \in \mathcal{L}(\mathfrak{A}) \mid t^v$ est d-compatible$\}$.*

Démonstration. Dans un arbre compatible avec les délais, certains sous-ensembles de nœuds doivent être étiquetés de la même façon. Par exemple, l'arbre représenté par la figure 3.9

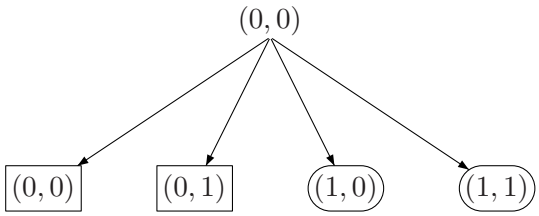


FIG. 3.9 – Contraintes sur l'étiquetage d'un arbre d-compatible

dans lequel les directions, indiquées entre parenthèses, sont formées par les valeurs de deux variables, x et y, de délais respectifs $d(x, v) = 0$, et $d(y, v) = 1$, est d-compatible si les deux nœuds encadrés sont étiquetés par la même valeur, et les deux nœuds encerclés sont également étiquetés par une valeur identique. Intuitivement, pour s'assurer que la fonction t^v est bien d-compatible, l'automate $\mathfrak{A}'$ doit deviner à l'avance les valeurs de t^v, puis vérifier que sa prédiction est correcte. Ceci doit être fait $K = \max\{d(u, v), u \in \mathrm{Vue}(v)\}$ étapes à l'avance et consiste en une fonction d-compatible $g : (S^{\mathrm{Vue}(v)})^K \to S^v$ qui détermine quelles devront être les valeurs de la variable v K instants plus tard. Pendant une transition de l'automate, cette prédiction est envoyée dans chaque direction $r \in S^{\mathrm{Vue}(v)}$ sous la forme d'une fonction $r^{-1}g$ définie par $(r^{-1}g)(\sigma) = g(r\sigma)$, et mémorisée dans l'état de l'automate. Les prédictions précédentes sont raffinées de façon similaire, et également mémorisée dans l'état de l'automate. Ainsi l'ensemble des états de $\mathfrak{A}'$ est $Q' = Q \times \mathcal{F}$, où $\mathcal{F}$ est l'ensemble des fonctions *d-compatibles* $f : (S^{\mathrm{Vue}(v)})^{<K} \to S^v$, avec $Z^{<K} = \bigcup_{i<K} Z^i$. La valeur $f(\varepsilon)$ est la prédiction ayant été faite K instants plus tôt, et doit être comparée avec la valeur courante de v dans l'arbre lu. Par exemple, en reprenant l'arbre dessiné figure 3.9, si l'automate est dans l'état (q, c, δ) en visitant le nœud père, où $c \in S^v$ est une constante, et $\delta : S^{x,y} \to S^v$ une fonction d-compatible, si l'étiquette du nœud courant est bien égale à c, l'automate, en se déplaçant dans la direction $(0, 0)$, va passer dans un état $(q', \delta(0, 0), \delta')$ où q' est dicté par l'étiquette qu'il a lu, et δ' est une nouvelle fonction d-compatible correspondant aux valeurs qu'il s'attend à trouver dans les fils du nœud courant. La fonction δ étant d-compatible, on s'assure ainsi que $\delta(0, 0) = \delta(0, 1)$ et l'automate va donc s'assurer ainsi que les nœuds fils dans les directions $(0, 0)$ et $(0, 1)$ sont bien étiquetés par la même valeur.

Pour formaliser cette intuition, on définit la fonction $\Delta : \mathcal{F} \times S^{\mathrm{Vue}(v)} \to 2^{\mathcal{F}}$ par

$$\Delta(f, r) = \{f' \mid f'(\sigma) = f(r\sigma) \text{ pour tout } |\sigma| < K - 1\}.$$

Cette fonction de transition signifie que, lorsque l'automate se trouve dans un état $(q, f) \in Q \times \mathcal{F}$ à un nœud τ de l'arbre, en allant dans la direction $r \in S^{\mathrm{Vue}(v)}$, $\Delta(f, r)$ calcule l'ensemble des fonctions de $\mathcal{F}$ qui pourraient étiqueter le nœud $\tau \cdot r$. En fait f' est déterminé par f et r pour toutes les valeurs σ telles que $|\sigma| < K - 1$, et correspond alors au raffinement de f par rapport à la direction r. Les fonctions $f' \in \Delta(f, r)$ diffèrent donc seulement sur les valeurs de σ telles que $|\sigma| = K - 1$ qui correspondent aux nouvelles prédictions.

On définit maintenant la fonction de transition de $\mathfrak{A}'$. Tout d'abord, elle n'est définie que pour les états $(q, f) \in Q'$ et $s \in S^U$ tels que $s^v = f(\varepsilon)$ – si c'est le cas, cela signifie que la prédiction faite K instants plus tôt était correcte. Sinon, l'exécution en cours n'est

pas acceptante et peut être stoppée. Lors d'une transition, on envoie dans chaque direction $r \in S^{\mathrm{Vue}(v)}$ de l'arbre une copie de l'automate dans l'état (q_r, g_r) où q_r correspond à la simulation d'une exécution de $\mathfrak{A}$ et $g_r \in \Delta(f, r)$. Formellement, si $s^v = f(\varepsilon)$,

$$\delta'\big((q,f),s\big) = \left\{ (q_r,g_r)_{r\in S^{\mathrm{Vue}(v)}} \,\middle|\, \begin{array}{c} (q_r)_{r\in S^{\mathrm{Vue}(v)}} \in \delta(q,s) \text{ et} \\ g_r \in \Delta(f,r) \text{ pour tout } r \in S^{\mathrm{Vue}(v)} \end{array} \right\}.$$

Enfin, l'ensemble des états initiaux de $\mathfrak{A}'$ est donné par $I' = \{q_0\} \times \mathcal{F}$ et $\alpha' = \pi^{-1}(\alpha)$, où $\pi : (Q \times \mathcal{F})^\omega \to Q$ est la projection sur la composante Q, i.e., une exécution de $\mathfrak{A}'$ est acceptante si et seulement si sa projection sur Q est une exécution acceptante de $\mathfrak{A}$.

On montre à présent que l'automate $\mathfrak{A}'$ est bien celui demandé par le lemme 3.36.

Soit t un $(S^{\mathrm{Vue}(v)}, S^U)$-arbre accepté par $\mathfrak{A}$ et supposons que t^v est d-compatible. Soit $\rho : (S^{\mathrm{Vue}(v)})^* \to Q$ une exécution acceptante de $\mathfrak{A}$ sur t. Il y a une façon unique d'étendre ρ en une exécution $\rho' : (S^{\mathrm{Vue}(v)})^* \to Q \times \mathcal{F}$ de $\mathfrak{A}'$ sur t. La seule possibilité est d'étiqueter un nœud $\sigma \in (S^{\mathrm{Vue}(v)})^*$ par l'application $f_\sigma : (S^{\mathrm{Vue}(v)})^{<K} \to S^v$ définie pour tout $\tau \in (S^{\mathrm{Vue}(v)})^{<K}$ par $f_\sigma(\tau) = t(\sigma\tau)^v$, afin que les prédictions soient correctes. Comme t^v est d-compatible, on en déduit que f_σ est aussi d-compatible, et donc appartient bien à $\mathcal{F}$. On définit donc l'exécution ρ' par $\rho'(\sigma) = (\rho(\sigma), f_\sigma)$ pour $\sigma \in (S^{\mathrm{Vue}(v)})^*$. On peut montrer que ρ' est une exécution acceptante de $\mathfrak{A}'$ sur t. En effet, prouvons d'abord que pour chaque nœud $\sigma \in (S^{\mathrm{Vue}(v)})^*$, la fonction de transition δ' est satisfaite. Soient $(q, f_\sigma) = \rho'(\sigma)$ et, pour tout $r \in S^{\mathrm{Vue}(v)}$, $(q_r, f_{\sigma r}) = \rho'(\sigma r)$. Par définition, $f_\sigma(\varepsilon) = t^v(\sigma)$ et $\delta'((q, f_\sigma), t(\sigma))$ est définie. Comme $\pi_Q(\rho') = \rho$, qui est une exécution de $\mathfrak{A}$ sur t, on sait que $(q_r)_{r\in S^{\mathrm{Vue}(v)}} \in \delta(q, t(\sigma))$. Il nous reste à montrer que $f_{\sigma r} \in \Delta(f_\sigma, r)$ pour tout $r \in S^{\mathrm{Vue}(v)}$. En fait, cela découle directement des définitions : pour tout $\tau \in (S^{\mathrm{Vue}(v)})^{<K-1}$, $f_{\sigma r}(\tau) = t^v(\sigma r\tau) = f_\sigma(r\tau)$. Enfin, l'exécution ρ est acceptante, puisque sa projection sur Q est ρ qui est une exécution acceptante.

Réciproquement, supposons qu'il existe une exécution acceptante de $\mathfrak{A}'$ sur t. On doit montrer que t^v est d-compatible et que $t \in \mathcal{L}(\mathfrak{A})$. On appelle $\rho' : (S^{\mathrm{Vue}(v)})^* \to Q \times \mathcal{F}$ une telle exécution. On décompose $\rho' = (\rho, H)$ en $\rho : (S^{\mathrm{Vue}(v)})^* \to Q$ et $H : (S^{\mathrm{Vue}(v)})^* \to \mathcal{F}$. Par définition de δ', on obtient immédiatement que ρ est une exécution de $\mathfrak{A}$ sur t, et qu'elle est acceptante puisque ρ' est acceptante.

Montrons que t^v est d-compatible. Comme ρ' est une exécution et que la fonction de transition δ' est uniquement définie pour $((q, f), s)$ avec $s^v = f(\varepsilon)$, on déduit que $t^v(\tau) = H(\tau)(\varepsilon)$ pour tout $\tau \in (S^{\mathrm{Vue}(v)})^*$. On doit donc montrer que l'application $\tau \mapsto H(\tau)(\varepsilon)$ est d-compatible.

Soient $\tau, \tau' \in (S^{\mathrm{Vue}(v)})^i$ tels que, pour tout $u \in \mathrm{Vue}(v)$, $\tau^u[i - d(u,v)] = \tau'^u[i - d(u,v)]$. On doit alors avoir $H(\tau)(\varepsilon) = H(\tau')(\varepsilon)$.

Si $|\tau| = |\tau'| > K$, alors on montre que nécessairement τ, τ' partagent un préfixe commun. Plus précisément, comme, pour tout $u \in \mathrm{Vue}(v)$, $K \geq d(u,v)$, on déduit du fait que $\tau^u[i - d(u,v)] = \tau'^u[i - d(u,v)]$ pour tout $u \in \mathrm{Vue}(v)$ que $\tau = \tau_1\tau_2$ et $\tau' = \tau_1\tau_2'$ avec $|\tau_2| = |\tau_2'| = K$ et, pour tout $u \in \mathrm{Vue}(v)$, $\tau_2^u[K - d(u,v)] = \tau'^u_2[K - d(u,v)]$. On peut montrer, par applications successives de la fonction de transition δ', et par définition de Δ, que la valeur de $H(\tau_1\tau_2)(\varepsilon)$ est en fait la prédiction faite au nœud τ_1 pour la direction définie par τ_2 : $H(\tau_1\tau_2)(\varepsilon) = H(\tau_1)(\tau_2)$. De même, on obtient que $H(\tau_1\tau_2')(\varepsilon) = H(\tau_1)(\tau_2')$. Comme $H(\tau_1) \in \mathcal{F}$, c'est une fonction d-compatible. Comme, pour tout $u \in \mathrm{Vue}(v)$, $\tau_2^u[K - d(u,v)] = \tau'^u_2[K - d(u,v)]$, on déduit que $H(\tau_1)(\tau_2) = H(\tau_1)(\tau_2')$. Donc, $H(\tau)(\varepsilon) = H(\tau')(\varepsilon)$.

Si $|\tau| < K$, alors on obtient de façon similaire que, puisque $H(\varepsilon) \in \mathcal{F}$ est d-compatible, $H(\tau)(\varepsilon) = H(\varepsilon)(\tau) = H(\varepsilon)(\tau') = H(\tau')(\varepsilon)$. □

Démonstration de la proposition 3.35. Considérons l'AAND $\text{compat}_{v_k}(\mathfrak{A}_k)$. Afin de conclure la démonstration, on doit supprimer la composante S^{v_k} de l'étiquette des arbres acceptés, et sélectionner les arbres de $\mathcal{L}(\text{compat}_{v_k}(\mathfrak{A}_k))$ tels que la composante $S^{V_{k+1}}$ de l'étiquette ne dépend que de la composante $S^{\text{Vue}(v_{k+1})}$ de la direction. La première opération est la construction classique de l'automate projection sur $S^{V_{k+1}}$, et la seconde correspond à la construction de rétrécissement (appelée *narrow*) introduite dans [KV99]. On décrit intuitivement cette construction. Soit $t : (S^{\text{Vue}(v_{k+1})})^* \to S^{V_{k+1}}$ un arbre. On peut construire l'arbre $\text{wide}_{\text{Vue}(v_k)}(t) : (S^{\text{Vue}(v_k)})^* \to S^{V_{k+1}}$ défini pour tout $\sigma \in (S^{\text{Vue}(v_k)})^*$ par $\text{wide}_{\text{Vue}(v_k)}(t)(\sigma) = t(\sigma^{\text{Vue}(v_{k+1})})$. Ainsi, les nœuds de $\text{wide}_{\text{Vue}(v_k)}(t)$ sont étiquetés d'une façon ne dépendant pas de la composante $S^{\text{Vue}(v_k)\setminus\text{Vue}(v_{k+1})}$ de leur direction. Pour tout $\mathfrak{A} = (S^{\text{Vue}(v_k)}, S^{V_{k+1}}, Q, q_0, \delta, \alpha)$ un automate alternant acceptant des arbres $t : (S^{\text{Vue}(v_k)})^* \to S^{V_{k+1}}$, on note $\text{narrow}_{\text{Vue}(v_{k+1})}(\mathfrak{A}) = (S^{\text{Vue}(v_{k+1})}, S^{V_{k+1}}, Q, q_0, \delta', \alpha)$ l'automate acceptant tous les arbres $t : (S^{\text{Vue}(v_{k+1})})^* \to S^{V_{k+1}}$ tels que $\text{wide}_{\text{Vue}(v_k)}(t) \in \mathcal{L}(\mathfrak{A})$. Pour $q \in Q$ et $s \in S^{V_{k+1}}$, la fonction de transition $\delta'(q, s)$ de l'automate $\text{narrow}_{\text{Vue}(v_{k+1})}(\mathfrak{A})$ est définie comme étant la fonction de transition $\delta(q, s)$ de l'automate $\mathfrak{A}$ dans laquelle on remplace chaque élément de la forme (r, q_r) avec $r \in S^{\text{Vue}(v_k)}$, et $q_r \in Q$ par l'élément $(r^{\text{Vue}(v_{k+1})}, q_r)$. Ainsi, l'arbre d'exécution de $\text{narrow}_{\text{Vue}(v_{k+1})}(\mathfrak{A})$ sur un arbre t est l'arbre d'exécution de $\mathfrak{A}$ sur un arbre t' dont l'étiquetage ne tient pas compte de la composante $S^{\text{Vue}(v_k)\setminus\text{Vue}(v_{k+1})}$, et tel que $t' = \text{wide}_{\text{Vue}(v_k)}(t)$. On remarque que même si l'automate $\mathfrak{A}$ est non-déterministe, l'automate $\text{narrow}_{\text{Vue}(v_{k+1})}(\mathfrak{A})$ construit est *alternant* : dans chaque direction $r \in S^{\text{Vue}(v_{k+1})}$ de l'arbre, $\text{narrow}_{\text{Vue}(v_{k+1})}(\mathfrak{A})$ envoie au moins une copie de $\mathfrak{A}$ par direction $\overline{r} \in S^{\text{Vue}(v_k)}$ tel que $\overline{r}^{\text{Vue}(v_{k+1})} = r$.

L'automate $\mathfrak{A}_{k+1}$ annoncé par la proposition 3.35 est donc donné par

$$\mathfrak{A}_{k+1} = \text{narrow}_{\text{Vue}(v_{k+1})}(\text{proj}_{V_{k+1}}(\text{compat}_{v_k}(\mathfrak{A}_k)))$$

. Comme on l'a remarqué ci-dessus, bien que l'automate $\mathfrak{A}_k$ soit un AAND, la construction narrow décrite dans [KV99] construit un automate d'arbres *alternant*. L'automate $\mathfrak{A}_{k+1}$ annoncé est donc l'automate $\text{narrow}_{\text{Vue}(v_{k+1})}(\text{proj}_{V_{k+1}}(\text{compat}_{v_k}(\mathfrak{A}_k)))$ transformé en AAND, en utilisant la construction classique de [MS95] (voir le théorème 2.9). L'inconvénient est que cette transformation induit une augmentation exponentielle de la taille de l'automate. Malheureusement, la construction décrite dans le lemme 3.36 nécessite de prendre un AAND en entrée. □

On conclut à présent la preuve du théorème 3.33. On commence par construire l'automate $\mathfrak{A}_1$ de la proposition 3.34, puis on applique récursivement la construction de la proposition 3.35 jusqu'à obtenir un AAND $\mathfrak{A}_n$ acceptant un $(S^{\text{Vue}(v_n)}, S^{v_n})$-arbre h^{v_n} si et seulement si pour tout $1 \leq i \leq n$, il existe un $(S^{\text{Vue}(v_i)}, S^{v_i})$-arbre h^{v_i} d-compatible et tel que $h^{v_1} \oplus \cdots \oplus h^{v_n} \in \mathcal{L}(\mathfrak{A}_1)$. Ainsi, par le lemme 3.32, il existe une stratégie distribuée gagnante pour la spécification sur $\mathcal{A}$ si et seulement si $\text{compat}_{v_n}(\mathfrak{A}_n)$ est non-vide. La complexité de la procédure est non-élémentaire, par application successive de la construction de la proposition 3.35 induisant chaque fois une augmentation exponentielle de la taille de l'automate. On ne connaît pas pour le moment de borne inférieure pour la complexité de ce problème. □

3.4 Architectures UWC et spécifications robustes

Dans cette section, on montre que si l'on restreint le type de spécifications que l'on s'autorise, on obtient la décidabilité du problème de SSD synchrone pour toute la classe des architectures UWC.

Définition 3.37. *Une spécification $\varphi \in \mathcal{L}$ avec $\mathcal{L} \in \{\mathsf{LTL}, \mathsf{CTL}, \mathsf{CTL}^*\}$ est* robuste *si elle s'écrit comme une disjonction finie de formules de la forme $\bigwedge_{v \in V_O} \varphi_v$ où $\varphi_v \in \mathcal{L}(\mathit{Vue}(v) \cup \{v\})$.*

Remarque 3.38. Une spécification robuste est toujours externe.

Proposition 3.39. *Le problème de SSD synchrone est décidable pour les instances $(\mathcal{A}, \varphi)$ où $\mathcal{A}$ est une architecture UWC et φ une spécification robuste de CTL^*.*

Démonstration. Soit $\mathcal{A} = (\mathrm{Proc}, V, E, (S^v)_{v \in V}, s_0, (d_p)_{p \in \mathrm{Proc}})$ une architecture UWC. Soit φ une spécification robuste de CTL^*. Sans perte de généralité, on suppose que $\varphi = \bigwedge_{v \in V_O} \varphi_v$, avec $\varphi_v \in \mathsf{CTL}^*(\mathrm{Vue}(v) \cup \{v\})$. En effet, considérons une formule $\overline{\varphi} = \bigwedge_{v \in V_O} \varphi_v \vee \bigwedge_{v \in V_O} \varphi'_v$. S'il existe une stratégie distribuée gagnante pour $\bigwedge_{v \in V_O} \varphi_v$ ou pour $\bigwedge_{v \in V_O} \varphi'_v$, alors elle est évidemment gagnante pour φ. Réciproquement, soit F une stratégie distribuée gagnante pour $(\mathcal{A}, \overline{\varphi})$. Alors $t_F : (S^{V_I})^* \to S^V$, l'arbre des exécutions respectant F, vérifie $t_F \models \overline{\varphi}$, et par définition, $t_F \models \bigwedge_{v \in V_O} \varphi_v$ ou $t_F \models \bigwedge_{v \in V_O} \varphi'_v$. Donc F est une stratégie distribuée gagnante pour $(\mathcal{A}, \bigwedge_{v \in V_O} \varphi_v)$ ou $(\mathcal{A}, \bigwedge_{v \in V_O} \varphi'_v)$.

Soit donc $\varphi = \bigwedge_{v \in V_O} \varphi_v$, avec $\varphi_v \in \mathsf{CTL}^*(\mathrm{Vue}(v) \cup \{v\})$. Par la proposition 3.34, on peut construire, pour chaque $v \in V_O$ un AAND $\mathcal{A}_v$ acceptant une stratégie $h : (S^{\mathrm{Vue}(v)})^* \to S^v$ si et seulement si $t : (S^{\mathrm{Vue}(v)})^* \to S^{\mathrm{Vue}(v) \cup \{v\}}$, l'arbre des exécutions respectant h, satisfait φ_v. On affirme qu'il existe une stratégie distribuée gagnante pour $(\mathcal{A}, \varphi)$ si et seulement si, pour chaque $v \in V_O$, l'automate $\mathrm{compat}_v(\mathfrak{A}_v)$ est non-vide.

En effet, soit F une stratégie distribuée gagnante pour $(\mathcal{A}, \varphi)$ et soit $t : (S^{V_I})^* \to S^V$ l'arbre des exécutions respectant F. Fixons une variable $v \in V_O$. L'application $\hat{f}^v : (S^{\mathrm{Vue}(v)})^* \to S^v$ est d-compatible. Considérons $t' : (S^{\mathrm{Vue}(v)})^* \to S^{\mathrm{Vue}(v) \cup \{v\}}$ l'arbre des exécutions respectant $\hat{f}^v$. Alors, pour chaque $\sigma \in (S^{V_I})^*$, $t(\sigma)^{\mathrm{Vue}(v) \cup \{v\}} = t'(\sigma^{\mathrm{Vue}(v)})$. Comme F est une stratégie gagnante, $t \models \varphi$ et donc $t \models \varphi_v$. En procédant par récurrence sur la structure de la formule, il est immédiat que pour toute formule $\psi \in \mathsf{CTL}^*(\mathrm{Vue}(v) \cup \{v\})$, pour toute branche $\sigma \in (S^{V_I})^\omega$, et toute position i, on a $t, \sigma, i \models \psi$ si et seulement si $t', \sigma^{\mathrm{Vue}(v)}, i \models \psi$. On en déduit, puisque $\varphi_v \in \mathsf{CTL}^*(\mathrm{Vue}(v) \cup \{v\})$, que $t' \models \varphi_v$. Ainsi, $\hat{f}^v \in \mathcal{L}(\mathfrak{A}_v)$ et donc $\hat{f}^v \in \mathcal{L}(\mathrm{compat}_v(\mathfrak{A}_v))$.

Réciproquement, pour tout $v \in V_O$, soit $h^v : (S^{\mathrm{Vue}(v)})^* \to S^v$ une stratégie acceptée par $\mathrm{compat}_v(\mathfrak{A}_v)$. D'après le lemme 3.36, h^v est d-compatible. Soit $t_v : (S^{\mathrm{Vue}(v)})^* \to S^{\mathrm{Vue}(v) \cup \{v\}}$ l'arbre des exécutions respectant h^v. Par définition de $\mathfrak{A}_v$, $t_v \models \varphi_v$. D'autre part, par le lemme 3.32, il existe une stratégie distribuée $F = (f^v)_{v \in V \setminus V_I}$ telle que $\hat{f}^v = h^v$ pour tout $v \in V_O$. Appelons $t : (S^{V_I})^* \to S^V$ l'arbre des exécutions respectant F. Pour tout $\sigma \in (S^{V_I})^*$, $t(\sigma)^{\mathrm{Vue}(v) \cup \{v\}} = t_v(\sigma^{\mathrm{Vue}(v)})$ et on obtient comme précédemment que $t \models \varphi_v$. Ainsi, $t \models \varphi$ et F est une stratégie distribuée gagnante pour $(\mathcal{A}, \varphi)$. □

4 Architectures bien connectées

Une question naturelle est de se demander si le critère de décidabilité que nous venons d'établir pour les architectures UWC peut être étendu à une classe plus large d'architectures. Dans cette section nous relâchons légèrement la définition de cette propriété et nous montrons que malheureusement, dans ce cas, être à information linéairement préordonnée n'est plus une condition suffisante pour obtenir la décidabilité du problème.

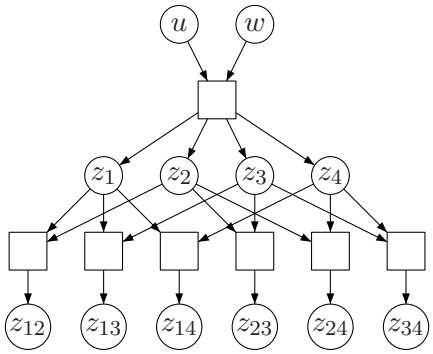


FIG. 3.10 – Une architecture bien connectée

4.1 Définition

Définition 3.40. *Une architecture est* bien connectée *si, pour chaque variable de sortie $v \in V_O$, la sous-architecture constituée des variables de $(E^{-1})^*(v)$ est uniformément bien connectée.*

Intuitivement, une architecture est bien connectée si, pour chaque variable de sortie v il y a un routage permettant de transmettre les valeurs des variables de Vue(v) au processus écrivant sur v, mais ce routage n'a pas besoin d'être identique pour toutes les variables de sortie, contrairement à une architecture uniformément bien connectée pour laquelle on demande l'existence d'un *unique* routage pour toutes les variables.

Exemple 3.41. L'architecture de la figure 3.10 est bien connectée. En effet, pour transmettre les valeurs de u et w à z_{ij}, il suffit de recopier la valeur de u sur z_i et la valeur de w sur z_j. On note que ce routage n'est pas uniforme. En fait, si le domaine des variables est $\{0, 1\}$, cette architecture n'est pas uniformément bien connectée (voir proposition 3.43).

Nous montrons donc tout d'abord que les architectures UWC forment une sous-classe stricte des architectures bien connectées. Dans la preuve de la proposition 3.43, on utilise le lemme suivant, établi dans [RLL04] afin de résoudre le problème de transmission d'information dans un réseau (présenté dans la section 3.2.2).

On dit que deux fonctions f et g de S^2 dans S sont indépendantes si $(f, g) : S^2 \to S^2$ est bijective.

Lemme 3.42 ([RLL04])**.** *Si $f^1, \ldots, f^n$ sont des fonctions de S^2 dans S indépendantes deux à deux, alors $n \leq |S| + 1$.*

Ce lemme établit que sur un petit domaine, on ne peut construire un grand ensemble de fonctions deux à deux indépendantes. Pour nous, il a la conséquence suivante :

Proposition 3.43. *L'architecture représentée figure 3.10, dans laquelle $S^v = \{0, 1\}$ pour tout $v \in V$, et $d_p = 0$ pour tout $p \in$ Proc est bien connectée, mais pas uniformément bien connectée.*

Démonstration. Il est facile de montrer que l'architecture $\mathcal{A}$ de la figure 3.10 est bien connectée (voir l'exemple 3.41). Supposons qu'elle est aussi UWC. Alors il existe un routage

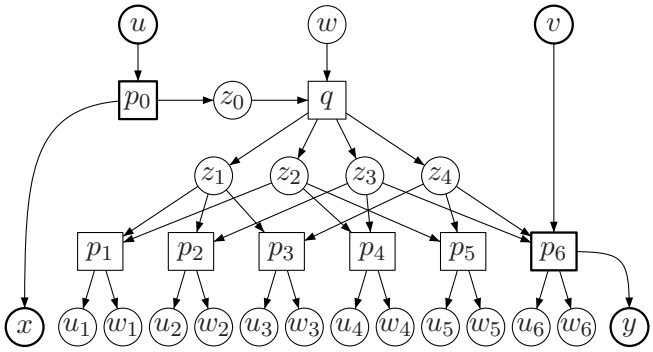


FIG. 3.11 – Architecture bien connectée, à information linéairement préordonnée, et indécidable

$\Phi = (f^{z_1}, f^{z_2}, f^{z_3}, f^{z_4})$ et, pour toute variable de sortie $v \in V_O$, des fonctions de décodage $g^{u,v} : \{0,1\}^2 \to \{0,1\}$ et $g^{w,v} : \{0,1\}^2 \to \{0,1\}$. On rappelle que lorsque l'architecture est sans délai, les fonctions de décodage sont sans mémoire. On notera $g^v : \{0,1\}^2 \to \{0,1\}^2$ la fonction qui, à $x \in \{0,1\}^2$ associe $g^v(x) = (g^{u,v}(x), g^{w,v}(x))$. Comme $\mathcal{A}$ est uniformément bien connectée, chaque paire (f^{z_i}, f^{z_j}) est inversible, d'inverse $g^{z_{ij}}$. Or, ceci est en contradiction avec le lemme 3.42, qui implique que pour des domaines booléens, il y a au plus trois fonctions indépendantes deux à deux. On en conclut que l'architecture n'est pas uniformément bien connectée. □

En fait, comme nous l'avons informellement remarqué au début de la section 3, la taille du domaine des variables a une influence sur la possibilité d'avoir un routage *uniforme* ou non, et le lemme 3.42 aide à comprendre pourquoi. Pour le problème de synthèse de système distribué synchrone, cela signifie qu'en augmentant suffisamment le domaine des variables internes, on peut transformer une architecture bien connectée en architecture uniformément bien connectée.

4.2 Architecture à information linéairement préordonnée indécidable

Le théorème suivant établit que le critère de décidabilité présenté dans les sections précédentes ne s'étend pas aux architectures bien connectées.

Théorème 3.44. *Le problème de SSD synchrone est indécidable pour les instances $(\mathcal{A}, \varphi)$ où $\mathcal{A}$ est une architecture bien connectée à information linéairement préordonnée et φ est une spécification externe de* LTL.

La fin de la section est dédiée à la démonstration de ce théorème. Soit $\mathcal{A}$ l'architecture de la figure 3.11, dans laquelle tous les délais valent 0. Elle est clairement bien connectée, et à information linéairement préordonnée : pour toutes les variables u_i et w_i, $1 \leq i \leq 5$, on peut transmettre les valeurs des variables u et w : la valeur de u est copiée sur z_0, puis il existe i_1, i_2 tels que $\{z_{i_1}, z_{i_2}\} = R(u_i) = R(w_i)$. Il suffit alors de copier z_0 et w respectivement sur z_{i_1} et z_{i_2}. Pour les variables u_6, w_6 et y il faut également transmettre la valeur de v qui est directement dans leur domaine de lecture. De même, on demande de transmettre u à la variable x, ce qui se fait directement. Par ailleurs, elle est à information linéairement préordonnée : $\mathrm{Vue}(x) \subseteq \mathrm{Vue}(u_i) = \mathrm{Vue}(w_i)$ pour $1 \leq i \leq 5$, et $\mathrm{Vue}(u_i) \subseteq \mathrm{Vue}(u_6) = \mathrm{Vue}(w_6) = \mathrm{Vue}(y)$.

Pour une machine de Turing M, on définit une spécification LTL φ_M et on va réduire le problème du non-arrêt de M sur bande vide au problème de synthèse de système distribué synchrone pour $(\mathcal{A}, \varphi_M)$. Soit $S^z = \{0,1\}$ pour $z \in V \setminus \{x,y\}$ et $S^x = S^y = \Gamma \uplus Q \uplus \{\#\}$ avec $\#$ un nouveau symbole. Une configuration de M définie par un état q et un contenu de bande $\gamma_1\gamma_2$, avec la tête de lecture située sur le premier symbole de γ_2 est représentée par le mot $\gamma_1 q \gamma_2 \in \Gamma^* Q \Gamma^+$ (on demande que γ_2 soit non vide pour des raisons techniques, en ajoutant des symboles vides si nécessaire). Une séquence de valeurs prises par la variable $u \in 0^*1^p0\{0,1\}^\omega$ code l'entier $n(u) = p$. On prend le même codage pour la variable v. On construit donc une spécification LTL φ_M forçant toute stratégie distribuée gagnante à écrire sur la variable x la $n(u)$-ième configuration de M lorsqu'elle commence sur la bande vide. Ainsi, les processus p_0 et p_6 vont jouer le rôle des deux processus de l'architecture indécidable de [PR90] (voir la démonstration du théorème 3.13). Bien sûr, notre cas est différent puisque le processus p_6 a la possibilité matérielle de recevoir de l'information sur la valeur de u. La difficulté est donc de masquer les informations utiles sur la valeur de u au processus p_6.

4.2.1 Description de la spécification

La spécification que nous allons construire est très proche de celle écrite dans la démonstration du théorème 3.13. On rajoute essentiellement une sous-formule ayant pour but d'empêcher la transmission d'information au processus p_6. Bien sûr, comme la spécification est externe, on ne peut pas directement empêcher les processus de communiquer l'information, mais on utilise le fait qu'il est impossible de transmettre les valeurs en entrée de p_0 et q simultanément à tous les processus. Formellement, soit $\varphi_M = \alpha \wedge \beta \wedge \gamma_M \wedge \delta \wedge \psi_M$ conjonction des cinq propriétés décrites ci-dessous.

1. Les processus p_i, pour $1 \leq i \leq 5$ doivent copier les valeurs courantes de (u,w) sur les variables (u_i, w_i) jusqu'à ce que w prenne la valeur 1 (valeur incluse). Puis, ils ne sont plus restreints par la spécification. Le processus p_6 doit toujours copier la valeur courante de w sur w_6. De plus, après l'occurrence du premier 1 sur w, il doit copier aussi la valeur courante de u sur u_6. Ceci est formalisé par la formule $\alpha \in \mathsf{LTL}(V_\mathrm{I} \cup V_\mathrm{O})$ suivante :

$$\alpha \stackrel{\text{def}}{=} \mathsf{G}(w_6 = w) \wedge \Big[\big((w=0) \wedge \alpha'\big)\ \mathsf{W}\ \big((w=1) \wedge \alpha' \wedge \mathsf{X}\,\mathsf{G}(u_6 = u)\big)\Big], \text{ avec}$$
$$\alpha' \stackrel{\text{def}}{=} \bigwedge_{1 \leq k \leq 5} (u_k = u) \wedge (w_k = w)$$

2. Si la séquence des valeurs prises par u (respectivement v) est dans $0^q1^p0\{0,1\}^\omega$, alors la séquence des valeurs prises par x (respectivement y) est dans $\#^{q+p}\Gamma^* Q \Gamma^+ \#^\omega$.
Ceci est exprimé par la formule $\beta = \beta_{u,x} \wedge \beta_{v,y}$ avec

$$\beta_{z,t} \stackrel{\text{def}}{=} ((z=0) \wedge (t=\#))\ \mathsf{W}$$
$$\Big((z=1) \wedge \big(((z=1) \wedge (t=\#))\ \mathsf{W}\ ((z=0) \wedge (t \in \Gamma^* Q \Gamma^+ \#^\omega))\big)\Big)$$

où

$$(t \in \Gamma^* Q \Gamma^+ \#^\omega) \stackrel{\text{def}}{=} (t \in \Gamma)\ \mathsf{U}\ ((t \in Q) \wedge \mathsf{X}(t \in \Gamma)\ \mathsf{U}\ ((t \in \Gamma) \wedge \mathsf{X}\,\mathsf{G}(t = \#)))$$

3. La formule γ_M que nous décrivons à présent impose que si $n(u) = 1$, alors la séquence de valeurs prises par x doit correspondre au codage de $\mathcal{C}_1$, la première configuration de la machine de Turing M commençant sur bande vide. Plus précisément, si la séquence de valeurs prises par u est dans $0^q 10\{0,1\}^\omega$, alors la séquence de valeurs prises par x au cours de l'exécution est $\#^{q+1}\mathcal{C}_1\#^\omega$. La formule $\gamma_M \in \mathsf{LTL}(V_\mathrm{I} \cup V_\mathrm{O})$ est

$$\gamma_M \stackrel{\text{def}}{=} (u = 0)\,\mathsf{W}\,((u = 1) \wedge \mathsf{X}((u = 0) \rightarrow (x \in \mathcal{C}_1\#^\omega)))$$

où $(x \in \mathcal{C}_1\#^\omega)$ peut s'exprimer facilement.

4. On dit que les mots d'entrée sont *synchronisés* si, soit $u, v \in 0^q 1^p 0\{0,1\}^\omega$ soit $u \in 0^q 1^{p+1} 0\{0,1\}^\omega$ et $v \in 0^{q+1} 1^p 0\{0,1\}^\omega$. La formule δ exprime le fait que si u et v sont synchronisés, et que $n(u) = n(v)$, alors x et y prennent la même séquence de valeurs au cours de l'exécution. Pour exprimer le fait que u et v sont synchronisés et que $n(u) = n(v)$, on définit la formule de $\mathsf{LTL}(V_\mathrm{I} \cup V_\mathrm{O})$

$$(n(u) = n(v)) \stackrel{\text{def}}{=} (u = v = 0)\,\mathsf{U}\,((u = v = 1) \wedge (u = v = 1)\,\mathsf{U}\,(u = v = 0))$$

La formule δ est donc définie par

$$\delta \stackrel{\text{def}}{=} (n(u) = n(v)) \rightarrow \mathsf{G}(x = y)$$

5. Enfin, on exprime avec la formule ψ_M que si les séquences d'entrée sont synchronisées et telles que $n(u) = n(v) + 1$, alors la configuration encodée sur la variable x est la configuration successeur de la configuration encodée sur la variable y dans l'exécution de la machine de Turing M. Pour exprimer le fait que $n(u) = n(v) + 1$ on utilise la formule $(n(u) = n(v) + 1) \in \mathsf{LTL}(V_\mathrm{I} \cup V_\mathrm{O})$ définie par :

$$(u = v = 0)\,\mathsf{U}\,\Big((u = 1) \wedge (v = 0) \wedge \mathsf{X}((u = v = 1) \wedge (u = v = 1)\,\mathsf{U}\,(u = v = 0))\Big)$$

La formule ψ_M est donc :

$$\psi_M = (n(u) = n(v) + 1) \rightarrow \Big((x = y)\,\mathsf{U}\,\big(\mathrm{Trans}(y, x) \wedge \mathsf{X}^3\,\mathsf{G}(x = y)\big)\Big)$$

où $\mathrm{Trans}(y, x)$ exprime le fait que le facteur de trois lettres de x est obtenu à partir de celui de y en effectuant une transition de la machine de Turing M. On a

$$\begin{aligned}\mathrm{Trans}(y, x) = &\bigvee_{(p,a,q,b,\leftarrow)\in T, c\in\Gamma} (y = cpa) \wedge (x = qcb)\\ \vee &\bigvee_{(p,a,q,b,\rightarrow)\in T, c\in\Gamma} (y = pac) \wedge (x = bqc)\\ \vee &\bigvee_{(p,a,q,b,\rightarrow)\in T} (y = pa\#) \wedge (x = bq\Box)\end{aligned}$$

On utilise l'abréviation $(x = abc)$ pour $(x = a) \wedge \mathsf{X}(x = b) \wedge \mathsf{X}^2(x = c)$. De plus, $\Box$ est le symbole vide de la bande et T est l'ensemble des transitions de M (la transition (p, a, q, b, dir), prise quand M est dans l'état p avec la tête de lecture sur le symbole a, change l'état de la machine en q, écrit sur la bande le symbole b et déplace la tête de lecture dans la direction $dir \in \{\leftarrow, \rightarrow\}$).

On montre tout d'abord qu'il existe une stratégie distribuée gagnante pour $(\mathcal{A}, \varphi_M)$. Comme précédemment, on va noter $\oplus$ l'addition modulo 2 (ou XOR). Le processus p_0 va copier la valeur de u sur z_0. Le processus q copie z_0 (donc la valeur de u) sur z_1, $u \oplus w$ sur z_2 et w sur z_3. Sa stratégie pour modifier z_4 nécessite elle de la mémoire : le processus q va copier sur z_4 la valeur de w jusqu'à ce que w prenne la valeur 1 pour la première fois (valeur incluse). Puis, il écrit $u \oplus w$ sur z_4. Formellement, pour tout $u \in \{0,1\}^*$, tout $b \in \{0,1\}$, $f^{z_4}(u, 0^q b) = b$ et pour tout $u, w \in \{0,1\}^*$, et tout $a, b \in \{0,1\}$, $f^{z_4}(ua, 0^q 1 wb) = a \oplus b$. Les stratégies pour les variables u_i, w_i, pour $1 \leq i \leq 6$ sont données par, pour $a, b \in \{0,1\}$

$$
\begin{aligned}
f^{u_1}(a,b) &= a & \qquad f^{w_1}(a,b) &= a \oplus b \\
f^{u_2}(a,b) &= a & \qquad f^{w_2}(a,b) &= b \\
f^{u_3}(a,b) &= a & \qquad f^{w_3}(a,b) &= b \\
f^{u_4}(a,b) &= a \oplus b & \qquad f^{w_4}(a,b) &= b \\
f^{u_5}(a,b) &= a \oplus b & \qquad f^{w_5}(a,b) &= b \\
f^{u_6}(a,b) &= a \oplus b & \qquad f^{w_6}(a,b) &= a
\end{aligned}
$$

Il est facile de se convaincre que toute exécution respectant ces stratégies locales satisfait la formule α. On remarque que tant que w n'a pas pris la valeur 1, la variable u_6 vaut toujours 0, et qu'après cette première occurence de 1 sur w, la variable u_5 est toujours mise à 0 (mais la spécification ne contraint plus le processus p_5 à copier les variables u et w).

La stratégie f^x (respectivement f^y) est d'écrire la p-ième configuration de M commençant sur la bande vide lorsque la séquence de valeurs prises par u (respectivement v) code p. On rappelle qu'avec l'encodage choisi, on peut déterminer quelle configuration est codée après un temps fini. Ainsi, on est assuré que le reste de la spécification $\beta \wedge \gamma_M \wedge \delta \wedge \psi_M$ est satisfaite.

Remarque 3.45. En fait, on pourrait décrire une autre stratégie gagnante en modifiant la stratégie de z_4 : à chaque instant, le processus q transmet au processus p_6 la valeur de u *à l'instant précédent* comme étant la différence modulo 2 des valeurs de z_3 et z_4, tant que le premier 1 n'est pas apparu sur w. Formellement, pour tout $a, a_1, a_2, b \in \{0,1\}$, pour tout $u, w \in \{0,1\}^*$, $f^{z_4}(a,b) = b$, $f^{z_4}(u \cdot a_1 \cdot a_2, 0^q b) = a_1 \oplus b$ et $f^{z_4}(ua, 0^q 1 wb) = a \oplus b$. On adapte également les stratégies des processus p_3 et p_5 afin de satisfaire α : elles ne peuvent plus être sans mémoire, elles doivent mémoriser la dernière valeur prise par u. Par exemple, le processus p_5 reçoit à l'instant initial les valeurs de w et de $u \oplus w$. Il peut donc retrouver u en effectuant le XOR de ses deux entrées. À l'instant suivant par contre, il reçoit la valeur de w additionnée de la valeur de u précédente (ce qu'on va noter $\mathsf{Y}\, u \oplus w$), ainsi que la valeur courante de $u \oplus w$. Pour décoder w il doit effectuer la somme modulo 2 de $\mathsf{Y}\, u \oplus w$ avec la valeur précédente de u, soit avec la somme modulo 2 de ses deux entrées précédentes : $f^{w_5}(a_1 a_2, b_1 b_2) = a_2 \oplus a_1 \oplus b_1$. À présent, en effectuant la somme modulo 2 de ses deux valeurs en entrée, le processus p_6 peut reconstituer toute l'histoire des valeurs prises par u, à l'exception d'une seule, celle écrite sur u au même instant que le premier 1 sur w. Ceci signifie que nous nous trouvons presque dans le cas de l'architecture décidable de la figure 2.3, mais que, de façon surprenante, *perdre un bit d'information* suffit à entraîner l'indécidabilité.

On prouve à présent que s'il existe une stratégie gagnante $F = (f^v)_{v \in V \setminus V_{\mathrm{I}}}$ pour $(\mathcal{A}, \varphi_M)$, alors nécessairement f^x simule le comportement de la machine de Turing M commençant sur bande vide. La difficulté supplémentaire par rapport au théorème 3.13 réside dans le fait qu'on doit s'assurer qu'aucune stratégie gagnante ne peut transmettre suffisamment d'information

à p_6 pour lui permettre de « tricher » et d'écrire sur sa variable de sortie une configuration différente de celle dictée par u.

4.2.2 Relations entre les stratégies de z_3 et z_4

Lemme 3.46. *Soient $g_1, g_2, g_3 : \{0,1\}^2 \to \{0,1\}$ des fonctions indépendantes deux à deux. Alors il existe $\varepsilon \in \{0,1\}$ tel que pour tout $a, b \in \{0,1\}$:*

$$g_3(a,b) = \varepsilon \oplus g_1(a,b) \oplus g_2(a,b)$$

Démonstration. On remarque d'abord que chaque fonction g_k est telle que, pour tout $c \in \{0,1\}$, $|g_k^{-1}(c)| = 2$. En effet, dans le cas contraire, il existe $c \in \{0,1\}$ tel que $|g_k^{-1}(c)| \geq 3$ et, pour $l \neq k$, l'application (g_k, g_l) ne peut pas être injective (et donc g_k et g_l ne sont pas indépendantes).

Pour la même raison, s'il existe $a, b, a', b' \in \{0,1\}$ tels que $g_k(a,b) = g_k(a',b')$, alors il existe $l \neq k$ tel que $g_l(a,b) \neq g_l(a',b')$. Donc, à permutation d'indices près, on suppose que $g_1(0,0) = g_1(0,1)$, $g_2(0,0) = g_2(1,0)$ et $g_3(0,0) = g_3(1,1)$. Ainsi, chaque g_k est entièrement déterminé par sa valeur sur $(0,0)$. Un simple calcul montre alors que $g_1 \oplus g_2 \oplus g_3$ est constant. Par exemple $g_1 \oplus g_2 \oplus g_3(0,0) = g_1 \oplus g_3(0,0) \oplus g_2(1,0)$. Si $g_1(0,0) = g_3(0,0) = 0$, alors $g_1(1,0) = g_3(1,0) = 1$ et $g_1 \oplus g_3(0,0) = g_1 \oplus g_3(1,0)$. Donc $g_1 \oplus g_2 \oplus g_3(0,0) = g_1 \oplus g_2 \oplus g_3(1,0)$. Si $g_1(0,0) = g_3(0,0) = 1$ alors $g_1(1,0) = g_3(1,0) = 0$ et on a à nouveau $g_1(0,0) \oplus g_3(0,0) = g_1(1,0) \oplus g_3(1,0)$. De même, si $g_1(0,0) = 0$ et $g_3(0,0) = 1$ alors $g_1(1,0) = 1$ et $g_3(1,0) = 0$ et $g_1 \oplus g_3(0,0) = g_1 \oplus g_3(1,0)$. Le cas où $g_1(0,0) = 1$ et $g_3(0,0) = 0$ est symétrique. Donc on a $g_1 \oplus g_2 \oplus g_3(0,0) = g_1 \oplus g_2 \oplus g_3(1,0)$. On peut effectuer le même raisonnement pour chaque couple d'entrées. □

En appliquant le lemme 3.46 à la fois à $(\hat{f}^{z_1}, \hat{f}^{z_2}, \hat{f}^{z_3})$ et $(\hat{f}^{z_1}, \hat{f}^{z_2}, \hat{f}^{z_4})$ après des séquences σ telles que $\sigma^u = 0^q$ et $\sigma^w = 0^q$, on obtient le corollaire suivant :

Corollaire 3.47. *Pour tout $q \geq 0$, il existe $\varepsilon \in \{0,1\}$ tel que*

$$\forall a, b \in \{0,1\}, \quad \hat{f}^{z_3}(0^q a, 0^q b) = \varepsilon \oplus \hat{f}^{z_4}(0^q a, 0^q b).$$

Démonstration. Soit $q \geq 0$. Soit $g_i : \{0,1\}^2 \to \{0,1\}$ défini par $g_i(a,b) = \hat{f}^{z_i}(0^q a, 0^q b)$. Le conjoint α de la spécification φ_M impose à p_1, p_2 et p_4 d'écrire la valeur courante du couple (u, w), donc ils doivent être capable de distinguer les quatre valeurs possibles de ces variables. Donc g_1, g_2 et g_3 sont indépendants deux à deux. En appliquant le lemme 3.46, on obtient l'existence de $\varepsilon_3 \in \{0,1\}$ tel que, pour tout $(a,b) \in \{0,1\}^2$, $g_3(a,b) = \varepsilon_3 \oplus g_1(a,b) \oplus g_2(a,b)$. De même, en considérant les valeurs des variables modifiées par p_1, p_3, p_5, on déduit que g_1, g_2 et g_4 sont aussi indépendants deux à deux et que $g_4(a,b) = \varepsilon_4 \oplus g_1(a,b) \oplus g_2(a,b)$.

Ainsi, pour tout $(a,b) \in \{0,1\}^2$, on a $g_3(a,b) \oplus g_4(a,b) = \varepsilon_3 \oplus \varepsilon_4 = \varepsilon$ et on obtient $\hat{f}^{z_3}(0^q a, 0^q b) = \varepsilon \oplus \hat{f}^{z_4}(0^q a, 0^q b)$. □

4.2.3 Perte d'une valeur de u par p_6

Soit $q \geq 0$. Pour $\overline{u} = 0^q 1 u'$ avec $u' \in \{0,1\}^\omega$, on définit $\overline{u}^0 = 0^q 0 u'$. On remarque que si $\overline{u} \in 0^q 1^{p+1} 0 \{0,1\}^\omega$ code $p+1 > 1$, alors $\overline{u}^0 \in 0^{q+1} 1^p 0 \{0,1\}$ code p. Le lemme suivant affirme que la stratégie f^{z_3} (respectivement f^{z_4}) écrit nécessairement la même séquence pour les valeurs d'entrée $\overline{u}$ et $\overline{u}^0$ pourvu que la séquence de valeurs prises par w soit d'une certaine

forme, de façon à ce que le processus p_6 soit incapable de faire la différence entre le codage de p et le codage de $p+1$ donné par la variable d'entrée u.

Lemme 3.48. *Soient $\overline{u}, \overline{w} \in 0^q1\{0,1\}^\omega$. Pour $k \in \{3,4\}$, pour tout $n > 0$, on a*

$$\hat{f}^{z_k}(\overline{u}^0[n], \overline{w}[n]) = \hat{f}^{z_k}(\overline{u}[n], \overline{w}[n]). \tag{3.8}$$

Démonstration. La preuve se fait par récurrence sur n. Si $n \leq q$, $\overline{u}^0[n] = \overline{u}[n]$ et (3.8) est trivialement vérifié.

Supposons maintenant que $n = q+1$, donc $\overline{u}^0[n] = 0^q0$ et $\overline{u}[n] = 0^q1 = \overline{w}[n]$. Si $\hat{f}^{z_3}(0^q0, 0^q0) = \hat{f}^{z_3}(0^q0, 0^q1)$, alors d'après le corollaire 3.47 on a également $\hat{f}^{z_4}(0^q0, 0^q0) = \hat{f}^{z_4}(0^q0, 0^q1)$. Pour un $\overline{v} \in \{0,1\}^n$ donné, cela implique que le processus p_6 a observé exactement la même histoire sur les triplets d'entrée $(0^q0, 0^q0, \overline{v})$ et $(0^q0, 0^q1, \overline{v})$; par conséquent il écrirait à l'instant n la même valeur sur w_6 dans les deux cas, ce qui violerait la condition α. Donc $\hat{f}^{z_3}(0^q0, 0^q0) \neq \hat{f}^{z_3}(0^q0, 0^q1)$. En suivant le même raisonnement, on obtient que $\hat{f}^{z_3}(0^q0, 0^q0) \neq \hat{f}^{z_3}(0^q1, 0^q1)$. Comme l'application $\hat{f}^{z_3}$ ne peut prendre que deux valeurs différentes, on en déduit que $\hat{f}^{z_3}(0^q0, 0^q1) = \hat{f}^{z_3}(0^q1, 0^q1)$. Le corollaire 3.47 nous permet à nouveau de conclure que $\hat{f}^{z_4}(0^q0, 0^q1) = \hat{f}^{z_4}(0^q1, 0^q1)$ et l'égalité (3.8) est vérifiée pour $n = q+1$.

Enfin, supposons que $n \geq q+1$. Par hypothèse de récurrence, pour $k \in \{3,4\}$, et pour tout $i < n$, on a $\hat{f}^{z_k}(\overline{u}^0[i], \overline{w}[i]) = \hat{f}^{z_k}(\overline{u}[i], \overline{w}[i])$. Donc pour tout $\overline{v} \in \{0,1\}^\omega$, pour tout couple $\sigma, \sigma_0 \in (S^V)^\omega$, exécutions respectant F tels que $\sigma^w = \sigma_0^w = \overline{w}$, $\sigma^v = \sigma_0^v = \overline{v}$ et $\sigma^u = \overline{u}$, $\sigma_0^u = \overline{u}^0$, les histoires sur z_3, $\sigma^{z_3}[n-1] = \sigma_0^{z_3}[n-1]$, et les histoires sur z_4, $\sigma^{z_4}[n-1] = \sigma_0^{z_4}[n-1]$, sont identiques, et donc à l'instant $n-1$, le processus p_6 a observé exactement les mêmes séquences de valeurs dans les deux cas : $\sigma^{z_3,z_4,v}[n-1] = \sigma_0^{z_3,z_4,v}[n-1]$.

Considérons à présent trois applications de $\{0,1\}^2$ dans $\{0,1\}^2$ définies par

$$\begin{aligned} h(c,d) &= (f^{u_6}, f^{w_6})(\sigma^{z_3}[n-1]c, \sigma^{z_4}[n-1]d, \sigma^v[n]) \\ h_1(a,b) &= (\hat{f}^{z_3}, \hat{f}^{z_4})(\overline{u}[n-1]a, \overline{w}[n-1]b) \\ h_0(a,b) &= (\hat{f}^{z_3}, \hat{f}^{z_4})(\overline{u}^0[n-1]a, \overline{w}[n-1]b) \end{aligned}$$

On déduit du fait que $\sigma^{z_3}[n-1] = \sigma_0^{z_3}[n-1]$ et $\sigma^{z_4}[n-1] = \sigma_0^{z_4}[n-1]$ et de la condition α que h est une fonction inverse de h_1 et de h_0. Par conséquent, $h_1 = h_0$ et, pour $k = 3, 4$, $\hat{f}^{z_k}(\overline{u}[n], \overline{w}[n]) = \hat{f}^{z_k}(\overline{u}^0[n], \overline{w}[n])$. □

4.2.4 Nécessité d'écrire la $n(u)$-ième configuration de M sur x

Tous ces résultats intermédiaires nous assurent que toute stratégie distribuée gagnante pour $(\mathcal{A}, \varphi_M)$ peut, si l'environnement joue d'une certaine façon, être contrainte de faire manquer une valeur de u au processus p_6. On montre que cela est suffisant pour retrouver l'indécidabilité du théorème 3.13 :

Lemme 3.49. *Soit $\sigma \in (S^V)^\omega$ une exécution respectant F. Alors pour tout $p > 0$ on a*

$$\forall q \geq 0, \quad \sigma^u \in 0^q1^p0\{0,1\}^\omega \Rightarrow \sigma^x = \#^{p+q}\mathcal{C}_p\#^\omega \tag{3.9}$$

avec $\mathcal{C}_p$ la p-ième configuration atteinte par M lors d'un calcul commençant sur la bande vide.

Démonstration. La démonstration se fait par récurrence sur p. Le cas $p = 1$ découle directement de la spécification γ_M. Soit $p \geq 1$, et supposons que $\sigma^u = \overline{u} \in 0^q 1^{p+1} 0\{0,1\}^\omega$, $\sigma^v = \overline{v} = 0^{q+1}1^p0^\omega$ et $\sigma^w = \overline{w} = 0^q1^\omega$. Soit également σ_0 exécution respectant F tel que $\sigma_0^v = \sigma^v$, $\sigma_0^w = \sigma^w$ et $\sigma_0^u = \overline{u}^0 \in 0^{q+1}1^p0\{0,1\}^\omega$. Par hypothèse de récurrence, $\sigma_0^x = \#^{q+1+p}\mathcal{C}_p\#^\omega$. Comme $\sigma_0 \models \varphi_M$, on déduit de la condition δ que $\sigma_0^y = \sigma_0^x = \#^{q+1+p}\mathcal{C}_p\#^\omega$. Par le lemme 3.48, $\sigma^{z_3} = \sigma_0^{z_3}$ et $\sigma^{z_4} = \sigma_0^{z_4}$. Ainsi, nécessairement, $\sigma^y = \sigma_0^y = \#^{q+1+p}\mathcal{C}_p\#^\omega$ comme montré ci-dessus. Comme $\overline{u}$ code la $(p+1)$-ième configuration de M et $\overline{v}$ la p-ième, et qu'ils sont synchronisés, la condition ψ_M de la spécification impose que $\sigma^x = \#^{q+1+p}\mathcal{C}_{p+1}\#^\omega$. Comme la valeur de x ne dépend que de la valeur de u, cela conclut la démonstration. □

En masquant un bit de u au processus p_6 on crée de l'incertitude au sujet de la valeur de $n(u)$, ce qui empêche ce processus de « tricher ». Par ailleurs, le processus p_0, qui n'a lui aucune information sur la valeur des autres variables d'entrée, sait seulement que l'information qu'a le processus p_6 sur la valeur de u peut être brouillée, et que donc ce dernier peut ne pas être capable de tricher. Donc le processus p_0 n'a pas d'autre choix que d'écrire la configuration correcte de la machine de Turing.

Démonstration du théorème 3.44. Pour une machine de Turing M, on a montré que toute stratégie distribuée gagnante pour $(\mathcal{A}, \varphi_M)$ est contrainte d'écrire sur la variable x la $n(u)$-ième configuration de M. Par conséquent, il existe une stratégie distribuée gagnante pour $(\mathcal{A}, \varphi_M \wedge \mathsf{G}(x \neq \text{halt}))$ si et seulement si M ne s'arrête pas lorsque son entrée est la bande vide. On a donc réduit le problème du non-arrêt d'une machine de Turing au problème de synthèse de systèmes distribués pour $(\mathcal{A}, \varphi)$ avec $\mathcal{A}$ architecture bien connectée à information linéairement préordonnée, et $\varphi \in \mathsf{LTL}(V_\mathrm{I} \cup V_\mathrm{O})$, démontrant par là que ce dernier problème est indécidable. □

5 Bilan

On a étudié dans ce chapitre le problème de synthèse de système distribué synchrone, pour lequel on a proposé la légère généralisation par rapport aux résultats existants d'autoriser des délais arbitraires pour les processus. On a choisi de se restreindre à des spécifications *externes*, qui semblent plus naturelles pour décrire des comportements de systèmes distribués ouverts.

Les résultats qu'on a présentés dans ce chapitre sont résumés sur la figure 3.12. On rappelle que [FS05] ont établi pour les spécifications *totales* que le problème de SSD synchrones est décidable si et seulement si l'architecture considérée est *ordonnée*. On a montré ici que, dans le cas de spécifications externes, si l'architecture est à *information incomparable* (ce qui représente une sous-classe des architectures non ordonnées), alors le problème de SSD synchrone est indécidable. Par contre, on a montré que si l'architecture considérée est uniformément bien connectée, alors le problème de SSD synchrone devient décidable pour toutes les architectures à information linéairement préordonnée. Pour continuer la comparaison avec les spécifications totales, on en conclut que, si on considère des domaines de variables booléens pour les architectures représentées sur la figure 3.3, alors le problème de SSD synchrone est indécidable pour l'architecture dessinée sur la sous-figure 3.3(a), que l'on autorise des spécifications externes ou totales, décidable pour l'architecture dessinée sur la sous-figure 3.3(c), que l'on autorise des spécifications externes ou totales, et qu'il est *indécidable* pour les architectures dessinées sur la sous-figure 3.3(b) si on autorise des spécifications totales, alors qu'il devient *décidable* quand on se restreint à des spécifications externes.

Malheureusement, on n'a pas trouvé pour le moment de classe plus générale d'architectures pour lesquelles être à information linéairement préordonnée deviendrait décidable. En particulier, on a mis en lumière que s'il importe peu qu'un processus soit à même de décoder l'information en entrée *plus tard* que les autres, tant que l'architecture est UWC, il est en revanche crucial qu'il obtienne *toute l'information* possible. C'est pourquoi les architectures bien connectées ne constituent pas un candidat satisfaisant. De la même manière, on pourrait envisager une classe d'architecture dans laquelle un routage existe, mais qui n'assure pas que l'information soit transmise *au plus vite* (comme c'est le cas pour les architectures UWC). Cependant, on peut à nouveau écrire une spécification similaire à celle de la preuve du théorème 3.44, forçant le processus p_2 de la figure 3.13 à perdre une valeur de la variable u, et nous permettant à nouveau d'appliquer le lemme 3.49 afin d'aboutir à l'indécidabilité.

Une autre approche permettant d'augmenter le nombre d'architectures pour lesquelles le problème de SSD synchrone est décidable, est de restreindre le langage de spécification. Un point crucial des résultats d'indécidabilité est que les démonstrations reposent fortement sur le fait que la spécification utilisée est *globale* (le synchronisme des exécutions permettant aisément de lier dans une formule de logique temporelle des valeurs de variables attachées à des processus n'ayant pas de possibilité de communiquer), alors que les contrôleurs que l'on cherche à synthétiser sont *locaux*. Une première tentative de limiter ce pouvoir des spécifications a été faite dans [MT01], qui ont défini le concept de spécifications locales. Les spécifications robustes qu'on a définies sont en fait une façon de combiner cette notion de localité, avec la notion de spécification externe. Ceci permet d'obtenir la décidabilité du problème dès que l'architecture considérée est UWC. Cependant, il faut noter que ce résultat n'étend pas à strictement parler les résultats de [MT01]. En effet, si on obtient techniquement plus d'architectures pour lesquelles le problème est décidable, la classe des pipelines à double entrée n'est pas incluse dans la classe des architectures UWC : il n'existe un routage des variables d'entrée vers les sorties dans les pipelines à double entrée que si le domaine des variables est suffisamment grand.

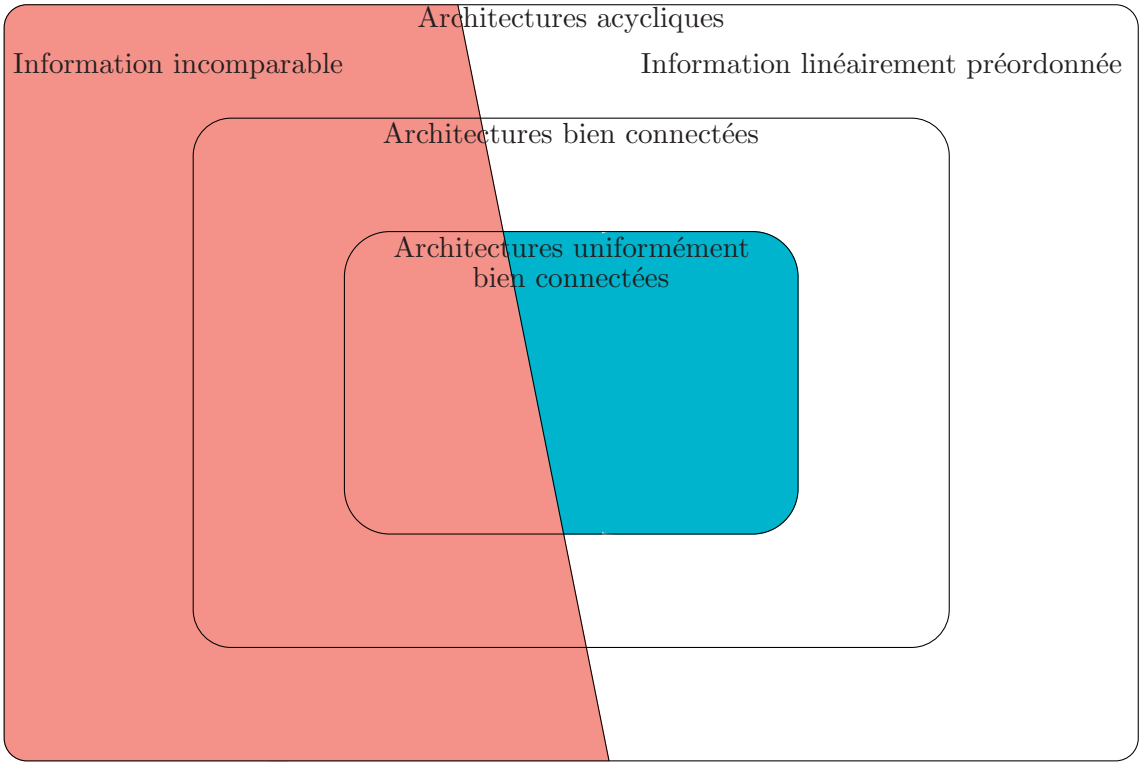


FIG. 3.12 – Le problème de synthèse de système distribué synchrone à délai avec spécifications externes

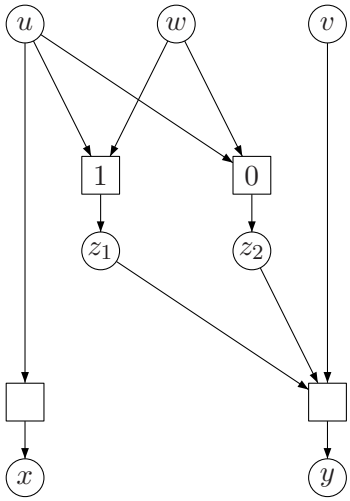


FIG. 3.13 – Une architecture pour laquelle le problème de SSD synchrone est indécidable

Chapitre 4

Synthèse de systèmes asynchrones

Sommaire

Dans ce chapitre, on étudie le problème de synthèse de systèmes distribués dans le cadre asynchrone. Comme on l'a vu dans les chapitres précédents, en particulier dans la preuve d'indécidabilité du problème de synthèse de systèmes distribués synchrones de [PR90], le *synchronisme* est un facteur important d'indécidabilité : c'est le fait que le comportement du système soit synchrone qui permet de lier dans la spécification les comportements de deux processus qui ne peuvent communiquer et qui ont une connaissance incomparable de l'état du système. Une façon de contourner ce problème est de limiter le pouvoir de la spécification, en lui interdisant de mettre en relation le comportement de deux processus qui ont ces caractéristiques : c'était le sens des spécifications robustes définies dans le chapitre précédent (définition 3.37 section 3.4 page 89). Une autre approche est de se placer dans le cadre plus général des systèmes asynchrones.

Ici on introduit un nouveau modèle, avec une modélisation des communications par synchronisation d'actions (voir section 2.3.2 page 41). On va chercher à calculer des contrôleurs distribués à mémoire *locale* (contrairement aux travaux de [GLZ04] et de [MTY05], qui autorisaient une mémoire *causale*). Par ailleurs, le modèle qu'on présente diffère de celui de [MT02b] de deux manières : dans leur modèle, les architectures sont *bipartites* (voir définition page 41), ce qui implique que si les processus évoluent bien de façon asynchrone les uns vis-à-vis des autres, chacun communique localement avec l'environnement de façon synchrone. Ici, on considère que les processus évoluent de façon asynchrone aussi bien vis-à-vis de l'environnement (qui peut effectuer plusieurs actions à la suite, comme ne plus rien faire jusqu'à la fin d'une exécution) que les uns par rapport aux autres. Une seconde différence repose sur les mécanismes de synchronisation entre les processus. Dans [MT02b] ce sont des communications par rendez-vous qui sont envisagées : une action partagée par deux processus est effectuée uniquement si les deux processus sont d'accord pour l'effectuer. Ici, on introduit des actions

qu'on appelle *signaux*, qui sont unidirectionnels : on définit pour chaque signal un processus émetteur qui peut le déclencher, et le signal est immédiatement reçu par le processus récepteur, que ce dernier désire le recevoir ou non. Ce mécanisme est plus pratique, et plus puissant du point de vue de la synthèse que la communication par variables partagées, et plus naturel vis-à-vis des applications pratiques que la communication par rendez-vous classique. Intuitivement, ce type de mécanisme peut correspondre dans un système réel à la possibilité pour le processus émetteur du signal de modifier une variable appartenant au processus récepteur. Ainsi, le processus récepteur est bien *immédiatement* conscient du message.

Comme dans le chapitre précédent, on n'autorisera pas les spécifications à contraindre le comportement interne du système : les communications entre les processus ne sont restreintes que par l'architecture, et non par la spécification. De plus, on considérera les exécutions de nos systèmes comme étant des traces de Mazurkiewicz. Les spécifications seront donc exprimées dans un formalisme logique ayant pour modèles des ordres partiels étiquetés par les actions externes du système. Enfin, on va se restreindre à des spécifications bénéficiant de propriétés de clôture naturelles, afin d'empêcher des contraintes irréalistes : on ne veut pas empêcher de relations de causalité entre deux événements (cela restreindrait les possibilités de communication des processus), et on ne veut pas qu'il soit possible d'imposer des causalités irréalisables a priori. On restreint également l'ensemble des exécutions à confronter avec la spécification aux exécutions équitables vis-à-vis de la stratégie choisie.

Dans la première section de ce chapitre, on présente plus en détail ce modèle et les spécifications que l'on va autoriser. Puis dans la seconde section, on présente les résultats de décidabilité obtenus : dans ce cadre, on obtient la décidabilité du problème de synthèse de systèmes distribués pour toute la sous-classe des architectures ayant un graphe de communication fortement connexe (i.e., dans lesquelles les processus peuvent tous communiquer les uns avec les autres – directement ou non).

1 Le modèle

1.1 Le système

Types d'architectures. On s'intéresse au problème de synthèse de systèmes distribués, dans le cadre de communications asynchrones. Comme on l'a relevé dans la section 2.3.1 (voir en particulier le théorème 2.51 page 41), le type de communication modélisé par des variables partagées rend le travail du contrôleur assez difficile dans une architecture asynchrone. En effet, supposons que le processus p veuille envoyer au processus q une séquence de messages $m_1, m_2 \ldots$. Pour ce faire, le processus commence par écrire sur une de ses variables x lues par le processus q la valeur m_1. Mais, avant d'écrire la valeur m_2, il doit s'assurer que le processus q a bien eu accès à la variable x. Comme le système est asynchrone, p ne peut savoir à quel moment cela sera fait. Il doit donc attendre une sorte d'accusé de réception de la part de q, i.e., que q modifie une de ses variables lues par p. Or, de tels accusés de réception ne sont pas possibles dans toutes les architectures. Dans tous les cas, ces mécanismes rendent la synthèse de programmes distribués plus difficiles.

Nous nous plaçons donc dans un modèle de communication par *signaux*, dans la veine de [LT89].

Un signal est une action partagée entre des processus, mais dont le caractère activable ne dépend que du processus *émetteur*, pas du récepteur. Comme dans le cas de communications par synchronisations d'actions qu'on a présenté dans la section 2.3.2 page 41, on assimile les

registres du système aux processus ($V = \text{Proc}$), et considère que toute action $a \in \Sigma$ peut modifier l'état de tous les processus qu'elle peut lire, i.e., $E^{-1}(a) = E(a)$. On traduit alors la notion de signal par l'ajout d'une application owner : $\Sigma_C \to \text{Proc}$ vérifiant, pour tout $a \in \Sigma_C$, $\text{owner}(a) \in E(a)$, qui associe à chaque action contrôlable son processus émetteur. Les actions incontrôlables Σ_{NC} sont émises par l'environnement, processus abstrait non représenté dans l'architecture. On peut donc étendre owner en une fonction partielle owner : $\Sigma \to \text{Proc}$ avec $dom(\text{owner}) = \Sigma_C$. Comme on l'a dit, l'émission d'un signal ne dépend que d'un seul processus, on impose donc : pour tout $a \in \Sigma_C$, pour tous $s_1, s_2 \in S^{\text{Proc}}$ tels que $s_1^{\text{owner}(a)} = s_2^{\text{owner}(a)}$, alors $a \in en(s_1)$ si et seulement si $a \in en(s_2)$. Par exemple, si on considère deux processus p et q et une action a telle que $E(a) = E^{-1}(a) = \{p, q\}$ avec $\text{owner}(a) = p$, si p est dans un état local dans lequel a est activable, alors quel que soit l'état local dans lequel q se trouve, il est possible de jouer a.

Par la suite, comme dans le chapitre précédent, on va s'intéresser uniquement au problème de *synthèse*. Les architectures que l'on considère sont donc telles que pour tout $p \in \text{Proc}$, $|S^p| = 1$, et pour tout $s \in S^{\text{Proc}}$, $en(s) = \Sigma$. Les domaines des registres ainsi que les fonctions de transitions locales des actions sont alors des données superflues, et on définira une architecture distribuée simplement par le tuple $\mathcal{A} = (\text{Proc}, \Sigma, E, \text{owner})$.

Les architectures vont respecter les contraintes supplémentaires suivantes : pour tout $a \in \Sigma$, $|E(a)| \leq 2$, i.e., on n'autorise pas d'envois de messages de type *broadcast*. De plus, comme on l'a dit, le processus environnement est abstrait, les actions de communication avec l'environnement sont donc considérées comme locales. On note ces actions $\Gamma = \{a \in \Sigma \mid |E(a)| = 1\}$. On les divise en deux catégories, les signaux émis par l'environnement, et les signaux émis par les processus. Pour $p \in \text{Proc}$, on note $\text{In}_p = \{a \in \Sigma_{NC} \mid E(a) = \{p\}\}$ les actions émises par l'environnement et reçues par le processus p, et $\text{Out}_p = \{a \in \Sigma_C \mid E(a) = \{p\}\}$ les signaux émis par le processus p vers l'environnement. On utilisera également les notations $\text{In} = \bigcup_{p \in \text{Proc}} \text{In}_p$, et $\text{Out} = \bigcup_{p \in \text{Proc}} \text{Out}_p$. Ainsi, les signaux *externes* sont $\Gamma = \text{In} \cup \text{Out}$.

Les actions de $\Sigma \backslash \Gamma$ sont les actions de communication entre les processus. Pour $p, q \in \text{Proc}$, on note $\Sigma^{p,q}$ l'ensemble des signaux émis par p et reçus par q. Formellement, $\Sigma^{p,q} = \{a \in \Sigma \mid E(a) = \{p, q\}, \text{owner}(a) = p\}$.

On note Σ^p les actions *visibles au processus* p, soit $\Sigma^p = \{a \in \Sigma \mid p \in E(a)\}$. Ces actions sont donc les actions de communications locales avec l'environnement, et les signaux qu'il émet et reçoit des autres processus : $\Sigma^p = \text{In}_p \cup \text{Out}_p \cup \bigcup_{q \in \text{Proc}} \Sigma^{p,q} \cup \Sigma^{q,p}$. On note enfin Σ_C^p le sous-ensemble des actions de Σ^p que le processus p peut contrôler : $\Sigma_C^p = \{a \in \Sigma^p \mid \text{owner}(a) = p\} = \text{Out}_p \cup \bigcup_{q \in \text{Proc}} \Sigma^{p,q}$.

Exemple 4.1 (Exemple d'architecture)**.** Considérons l'architecture à synthétiser $\mathcal{A} = (\text{Proc}, \Sigma, E, \text{owner})$ représentée figure 4.1. Ici, l'ensemble des processus est $\text{Proc} = \{1, 2, 3\}$. Afin de mettre en évidence le processus émetteur de chaque signal, on a adopté une représentation de type graphe de communication, dans lequel les flèches représentent les différents ensembles de signaux décrits ci-dessus. L'image du haut représente la signature telle que dessinée dans le chapitre 2, dans laquelle on a regroupé toutes les actions de même type en un seul carré. Dans l'image du bas on représente la signature telle qu'on les dessinera dans ce chapitre, avec les processus représentés dans des carrés pour garder une homogénéité avec le chapitre précédent. Les actions de $\Sigma^{1,2}$ par exemple sont bien partagées entre les processus 1 et 2, mais dans l'image du bas le sens de la flèche indique que c'est le processus 1 qui les émet. Les signaux contrôlés par l'environnement sont représentés par des flèches entrantes sur le graphe de communication, et les signaux externes contrôlés par les processus sont représentés

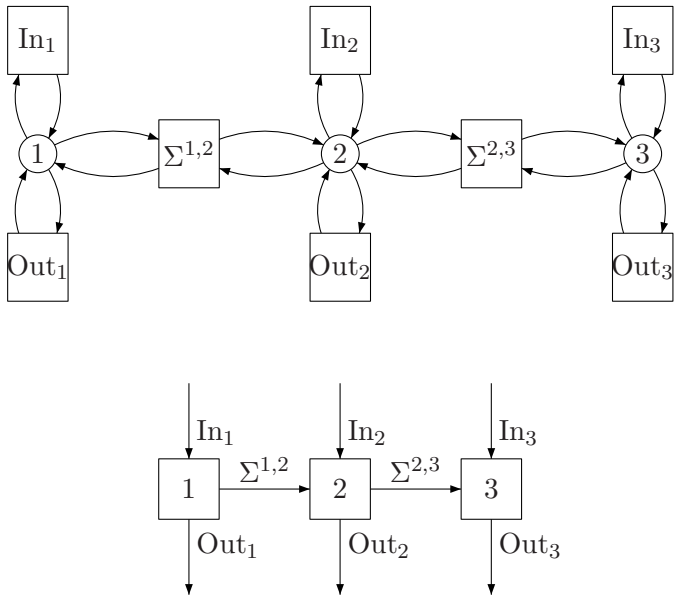


FIG. 4.1 – Un exemple d'architecture

par des flèches sortantes. On a donc par exemple, $\Sigma^2_C = \mathrm{Out}_2 \cup \Sigma^{2,3}$, et $\Sigma^2 = \mathrm{In}_2 \cup \Sigma^{1,2} \cup \Sigma^2_C$.

Exécutions du système. On se place dans un modèle totalement asynchrone dans lequel on assimile Σ' et Σ. On rappelle (voir la remarque 2.12 page 20) que le langage $\mathcal{L}(\mathcal{A})$ est clos par équivalence de traces. De plus, comme nos architectures sont à synthétiser, la séquence d'états visités au cours d'une exécution est toujours la même, et on peut assimiler $\mathcal{L}(\mathcal{A})$ à $\mathrm{Runs}(\mathcal{A})$. On va donc considérer que les exécutions sont des traces de Mazurkiewicz de $\mathbb{R}(\Sigma, D)$. On rappelle (voir la relation (2.1) donnée page 20) que pour tous $a, b \in \Sigma$, $a\ D\ b$ si et seulement si $(E^{-1}(a) \cap E(b)) \cup (E^{-1}(b) \cap E(a)) \cup (E(a) \cap E(b)) \neq \emptyset$. Ici, ceci est équivalent à $E(a) \cap E(b) \neq \emptyset$.

Remarque 4.2. Soit $\alpha = (X, \leq, \lambda) \in \mathbb{R}(\Sigma, D)$ une exécution. Avec ces notations, deux actions a et $b \in \Sigma$ sont dépendantes si et seulement si il existe un processus $p \in \mathrm{Proc}$ tel que $a, b \in \Sigma^p$. Dans α, l'ordre entre deux événements locaux se passant sur deux processus distincts est nécessairement induit par des actions partagées, i.e., des actions de communication. On en déduit que pour tous processus $p \neq q \in \mathrm{Proc}$, et événements $x \in \lambda^{-1}(\Sigma^p)$ et $y \in \lambda^{-1}(\Sigma^q)$, si $x \leq y$, alors il existe $z_p \in \lambda^{-1}(\Sigma^p \setminus \Gamma)$ et $z_q \in \lambda^{-1}(\Sigma^q \setminus \Gamma)$ tels que $x \leq z_p \leq z_q \leq y$. Les événements x et z_p sont ordonnés car ils sont dépendants, ainsi que y et z_q, et il existe entre z_p et z_q une séquence d'événements partagés (par exemple si simplement $z_p = z_q \in \lambda^{-1}(\Sigma^{p,q} \cup \Sigma^{q,p})$).

Stratégies. On cherche à synthétiser des stratégies *déterministes* distribuées parmi les *processus*, *à mémoire locale*. Comme ici, nous assimilons les exécutions de $\mathcal{A}$ à $\mathbb{R}(\Sigma, D)$, on définit les programmes du système sur des préfixes d'exécutions, donc sur des traces de Mazurkiewicz finies : $F : \mathbb{M}(\Sigma, D) \to 2^{\Sigma}$. La stratégie F est distribuée parmi les processus s'il existe un tuple de stratégies $(f^p)_{p \in \mathrm{Proc}}$ avec $f^p : \mathbb{M}(\Sigma, D) \to \Sigma^p_C$ et vérifiant, pour tout $\alpha \in \mathbb{M}(\Sigma, D)$,

$F(\alpha) = \Sigma_{NC} \cup \{f^p(\alpha) \mid p \in \text{Proc}\}$. Ainsi, F est un programme du système non-bloquant, (il satisfait les conditions (2.2), (2.3), et (2.4) (voir page 22).

Une stratégie à mémoire locale pour le processus p est une stratégie qui ne tient compte que des actions visibles pour p, i.e. des actions de Σ^p. On rappelle que pour un ordre partiel $\alpha = (X, \leq, \lambda)$ étiqueté par Σ, on note, pour tout $\Sigma' \subseteq \Sigma$ la *projection* de α sur Σ' comme étant l'ordre partiel $\pi_{\Sigma'}(\alpha) = (X \cap \lambda^{-1}(\Sigma'), \leq \cap(\lambda^{-1}(\Sigma'))^2, \lambda)$.

Définition 4.3 (Stratégie à mémoire locale)**.** *Soit $f^p : \mathbb{M}(\Sigma, D) \to \Sigma^p_C$ une stratégie pour le processus $p \in$ Proc. Elle est* à mémoire locale *si, pour tous $\alpha, \alpha' \in \mathbb{M}(\Sigma, D)$ tels que*

$$\pi_{\Sigma^p}(\alpha) = \pi_{\Sigma^p}(\alpha'),$$

on a

$$f^p(\alpha) = f^p(\alpha').$$

Une stratégie distribuée $F = (f^p)_{p \in \text{Proc}}$ est à mémoire locale si, pour tout $p \in$ Proc, f^p est à mémoire locale.

Les stratégies des processus ne sont pas nécessairement des fonctions totales. Ici, on autorise donc un processus à s'arrêter de jouer s'il estime que la spécification est satisfaite. Par contre, la stratégie du système, F, respectant la condition (2.3), propose toujours les signaux de l'environnement. La stratégie ne peut donc pas imposer qu'une exécution soit finie.

Pour une trace $\alpha = (X, \leq, \lambda) \in \mathbb{R}(\Sigma, D)$, pour $x \in X$, on notera $\alpha_{\downarrow_\alpha x} = (\downarrow_\alpha x, \leq, \lambda)$ et $\alpha_{\Downarrow_\alpha x} = (\Downarrow_\alpha x, \leq, \lambda)$ les traces préfixes de α constituées respectivement des événements dans le passé et dans le passé strict de x.

Exécutions selon une stratégie distribuée. Soit $F : \mathbb{M}(\Sigma, D) \to 2^\Sigma$ une stratégie distribuée du système, à mémoire locale. Une exécution $\alpha = (X, \leq, \lambda) \in \mathbb{R}(\Sigma, D)$ *respecte la stratégie $F = (f^p)_{p \in \text{Proc}}$* (ou est *$F$-compatible*), si toutes les actions contrôlables sont étiquetées en fonction de la stratégie, i.e., pour tout $p \in$ Proc, pour tout $x \in \lambda^{-1}(\Sigma^p_C)$, $\lambda(x) = f^p(\alpha_{\Downarrow_\alpha x})$. On rappelle également qu'une exécution finie $\alpha \in \mathbb{M}(\Sigma, D)$ est F-maximale si $F(\alpha) \cap \Sigma_C = \emptyset$.

Remarque 4.4. Pour toute stratégie F, tout événement d'une exécution étiqueté par un signal de l'environnement respecte la stratégie, par la condition (2.3). Donc cette définition d'une exécution F-compatible rejoint la définition donnée page 22 (définition 2.16). Par ailleurs, contrairement au cas synchrone, pour une stratégie déterministe fixée, même si la séquence d'actions jouée par l'environnement est fixée aussi, il existe plusieurs exécutions F-compatibles et respectant le schéma d'actions de l'environnement, dépendant de l'ordonnancement des actions choisi.

Exemple 4.5 (Exemple d'exécution F-compatible)**.** Considérons l'architecture représentée sur la figure 4.1 pour laquelle on définit l'alphabet d'actions : $\text{In}_1 = \{\texttt{req}_1\}$, $\text{In}_2 = \{\texttt{req}_2\}$, $\Sigma^{1,2} = \{\texttt{msg}_1\}$, $\Sigma^{2,3} = \{\texttt{msg}_{1,3}, \texttt{msg}_2\}$ et $\text{Out}_3 = \{\texttt{rep}_1, \texttt{rep}_2\}$. Informellement, la stratégie du processus 1 est d'envoyer au processus 2 un signal $\texttt{msg}_1$ à chaque fois qu'il reçoit un signal $\texttt{req}_1$ de l'environnement, la stratégie du processus 2 est d'envoyer un signal $\texttt{msg}_2$ au processus 3 si le dernier signal reçu est $\texttt{req}_2$ et $\texttt{msg}_{1,2}$ si c'est $\texttt{msg}_1$, et la stratégie du processus 3 est d'émettre un signal $\texttt{rep}_1$ si le dernier signal reçu est $\texttt{msg}_{1,3}$ et $\texttt{rep}_2$ si le dernier message reçu est $\texttt{msg}_2$. Alors la trace représentée sur la figure 4.2 est compatible avec cette stratégie. Ce n'est en revanche pas une exécution maximale, la stratégie du processus 1 étant d'envoyer un message après la réception du dernier signal de l'environnement. On a représenté à droite une

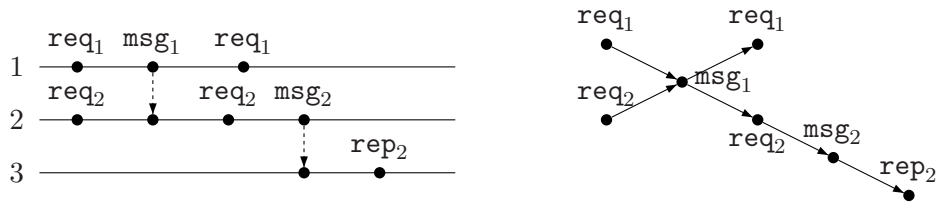


FIG. 4.2 – Une exécution F-compatible

trace comme un ordre partiel classique, et à gauche la représentation faisant explicitement apparaître les processus. Une action partagée apparaît donc sur deux processus à la fois, le sens de la flèche mettant en évidence quel processus est émetteur du signal. La représentation avec processus explicites permet de voir plus facilement les signaux visibles à un processus donné, et donc ce sur quoi la stratégie prend sa décision. En effet, une stratégie à mémoire locale ne dépend en fait que de l'histoire totalement ordonnée des actions visibles au processus considéré.

Équité. On s'intéresse uniquement aux exécutions *équitables* du système. On commence par remarquer que les notions d'équités globales telles que définies dans la définition 2.45 page 40 ne sont pas suffisantes pour éliminer les exécutions « dégénérées ».

Exemple 4.6. On reprend l'architecture de la figure 4.1, dans laquelle on suppose à présent donnés les alphabets d'actions externes $\mathrm{In}_1 = \{\mathtt{req}_1\}$, $\mathrm{Out}_1 = \{\mathtt{rep}_1\}$, $\mathrm{In}_2 = \{\mathtt{req}_2\}$, $\mathrm{Out}_2 = \{\mathtt{rep}_2\}$, et on considère une spécification informelle qui demande : « Toute requête $\mathtt{req}_1$ est suivie d'une réponse $\mathtt{rep}_1$, et toute requête $\mathtt{req}_2$ est suivie d'une réponse $\mathtt{rep}_2$ ». Une telle spécification est en fait une conjonction de restrictions sur des comportements locaux des processus, et ne devrait pas être plus difficile à réaliser que sur une architecture constituée d'un seul processus. En particulier, la stratégie F telle que, pour $i = 1, 2$, f^i propose de jouer $\mathtt{rep}_i$ à chaque fois que le processus i reçoit un signal $\mathtt{req}_i$ devrait être gagnante. Or, l'exécution dans laquelle le processus 2 reçoit un signal $\mathtt{req}_2$ et ne joue jamais, tandis que le processus 1 joue une infinité de signaux $\mathtt{rep}_1$ est compatible avec cette stratégie, et faiblement équitable au sens global de la définition 2.45, mais ne satisfait pas la spécification.

Il est donc naturel de demander que chaque processus ait les mêmes chances d'envoyer un signal s'il le désire. On va considérer par conséquent une notion d'équité plus *locale*.

Pour exprimer la notion d'équité en toute généralité, on la définit en fonction d'une partition des actions de Σ_C. La partition la plus grossière correspond à une équité globale. L'exemple 4.6 indique qu'il est plus intéressant de se restreindre aux partitions comprises entre celle faite en fonction des actions contrôlables des processus : $\mathcal{P} = \{\Sigma_C^p \mid p \in \mathrm{Proc}\}$ et la partition la plus fine des actions : $\mathcal{P} = \{\{a\} \mid a \in \Sigma_C\}$.

Pour $\alpha, \alpha' \in \mathbb{R}(\Sigma, D)$ on note $\alpha \leq \alpha'$ pour « α est une trace préfixe de α' » et on définit formellement les exécutions équitables comme suit :

Définition 4.7 (Exécutions équitables)**.** *On dit qu'une exécution $\alpha = (X, \leq, \lambda) \in \mathbb{R}(\Sigma, D)$ est* F-équitable pour une partition $\mathcal{P}$ *des actions contrôlables Σ_C si, pour tout $\Sigma' \in \mathcal{P}$, s'il existe*

$\alpha' \in \mathbb{M}(\Sigma, D)$ *préfixe de* α *telle que, pour toute trace* α'' *telle que* $\alpha' \leq \alpha'' \leq \alpha$, $F(\alpha'') \cap \Sigma' \neq \emptyset$, *alors* $\lambda^{-1}(\Sigma')$ *est infini.*

Remarque 4.8. Une exécution finie est F-équitable si et seulement si elle est F-maximale. En effet, soit $\alpha \in \mathbb{M}(\Sigma, D)$ une exécution F-compatible finie. Si $F(\alpha) \cap \Sigma_C \neq \emptyset$ (et donc α n'est pas F-maximale), alors il existe Σ' un élément de la partition, et un préfixe fini $\alpha' = \alpha$ de α tel que pour tout $\alpha \leq \alpha'' \leq \alpha$, $F(\alpha'') \cap \Sigma' \neq \emptyset$. Or, α étant fini, on a $\lambda^{-1}(\Sigma')$ nécessairement fini. Donc α n'est pas F-équitable. Si réciproquement, α est F-maximale, alors $F(\alpha) \cap \Sigma_C = \emptyset$, et donc pour tout $\Sigma' \in \mathcal{P}$, pour tout $\alpha' \in \mathbb{M}(\Sigma, D)$, il existe $\alpha'' = \alpha$ tel que $F(\alpha'') \cap \Sigma' = \emptyset$. L'exécution est donc F-équitable car satisfaisant trivialement l'implication.

Remarque 4.9. La définition 4.7 revient à dire que pour toute trace préfixe $\alpha'' = (X'', \leq, \lambda)$ telle que $\alpha' \leq \alpha'' \leq \alpha$, il existe $x \in X \setminus X''$ tel que $\lambda(x) \in \Sigma'$.

Par la suite, sauf mention contraire, on va s'intéresser à la notion la plus forte d'équité locale, qu'on appellera *équité locale forte*[1], c'est-à-dire à la partition des actions $\mathcal{P} = \{\Sigma_C^p \mid p \in \text{Proc}\}$. Les exécutions équitables seront donc celles correspondant à la définition suivante, qui est une reformulation de la définition 4.7 :

Définition 4.10. *On dit qu'une exécution* $\alpha = (X, \leq, \lambda) \in \mathbb{R}(\Sigma, D)$ *est* F-équitable *si, pour tout processus* $p \in \text{Proc}$*, s'il existe* $\alpha' \in \mathbb{M}(\Sigma, D)$ *préfixe de* α *telle que, pour toute trace* α'' *telle que* $\alpha' \leq \alpha'' \leq \alpha$, $F(\alpha'') \cap \Sigma_C^p \neq \emptyset$, *alors* $\lambda^{-1}(\Sigma_C^p)$ *est infini.*

Les spécifications. Comme dans le chapitre 3, on affirme que les spécifications raisonnables à considérer sont de type *externe*. En effet, on veut que les processus soient libres de collaborer sans restriction afin de se conformer aux comportements autorisés. On se restreint donc à des spécifications ne contraignant que les actions de Γ, qui sont les actions de communication avec l'environnement. De plus, nos spécifications seront sur des *ordres partiels*. En effet, dans ce modèle, on définit les exécutions comme étant déjà des traces de Mazurkiewicz, donc des ordres partiels. Il est donc logique que la spécification ait comme modèles des ordres partiels. Par ailleurs, comme on l'a déjà signalé dans la section 2.3.3 page 46, des spécifications sur des linéarisations des exécutions pourraient faire la distinction entre deux linéarisations de la même exécution (en accepter une et en rejeter l'autre), ce qui n'est pas désirable.

Pour répondre à ces deux exigences, les spécifications seront donc données dans un formalisme logique dont les modèles sont des ordres partiels étiquetés par Γ.

Pour une exécution *concrète* du système, $\alpha = (X, \leq, \lambda) \in \mathbb{R}(\Sigma, D)$ on définit sa partie *observable* (dénommée *exécution abstraite*, ou *exécution observable*) par

$$\pi_\Gamma(\alpha) = (\lambda^{-1}(\Gamma), \leq \cap (\lambda^{-1}(\Gamma))^2, \lambda).$$

On dira qu'une exécution concrète α satisfait une spécification φ si sa projection $\pi_\Gamma(\alpha)$ satisfait φ. On remarque que les exécutions observables sont des ordres partiels étiquetés par Γ ayant la caractéristique que, pour tout processus $p \in \text{Proc}$, l'ensemble des événements de $\lambda^{-1}(\Sigma^p)$ est totalement ordonné.

Exemple 4.11. Reprenons l'architecture représentée sur la figure 4.1 mais restreinte aux processus 1 et 2. On considère l'alphabet d'actions $\text{In}_1 = \{\texttt{req}_1\}$, $\text{In}_2 = \{\texttt{cancel}\}$, $\Sigma^{1,2} =$

[1] Cette appellation peut prêter à confusion : en effet, la notion d'équité que l'on définit correspond à ce qui est généralement appelé « équité faible ». On appelle ici équité locale forte, l'équité « faible » la plus forte du point de vue de la partition des actions.

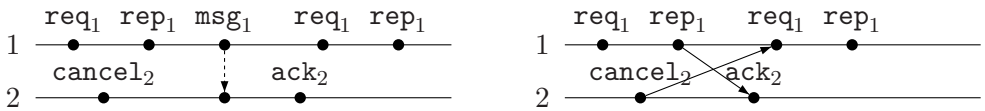


FIG. 4.3 – Une exécution concrète et sa partie observable

$\{\texttt{msg}_1\}$, $\text{Out}_1 = \{\texttt{rep}_1\}$ et $\text{Out}_2 = \{\texttt{rep}_2, \texttt{ack}_2\}$. L'exécution représentée sur la figure 4.3 satisfait la spécification informelle « Toute action de type $\texttt{req}_1$ est suivie par une action de type $\texttt{rep}_1$ et (par une réponse de type $\texttt{rep}_2$ si et seulement si il n'y a pas eu de signal $\texttt{cancel}_2$) et toute action de type $\texttt{cancel}_2$ est suivie par une action $\texttt{ack}_2$ ». On rappelle que le signal $\texttt{msg}_1$ est bien une action *partagée* entre les processus 1 et 2 – le sens de la flèche n'est représenté que pour indiquer le sens du signal. Ceci explique la causalité apparaissant entre $\texttt{cancel}_2$ et $\texttt{req}_1$ dans la partie observable de l'exécution.

Remarque 4.12. Les ordres partiels observables, modèles de nos spécifications, ne sont plus des traces de Mazurkiewicz. On perd en particulier la relation surjective [ ] définie dans la section 1.1.4 page 11 liant les linéarisations des exécutions et les ordres partiels correspondants (i..e., on peut trouver deux ordres partiels distincts ayant une linéarisation commune).

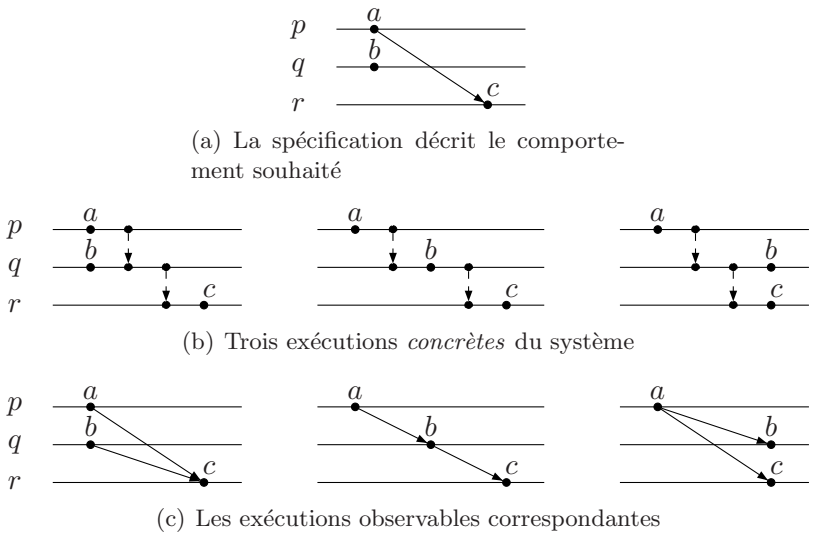
Le problème de synthèse de systèmes distribués. Afin de laisser le plus de latitude possible aux processus, on modifie légèrement la façon dont on donne le problème : on laisse aux processus le loisir de choisir leur alphabet de communication. La donnée du problème est maintenant un tuple $\mathcal{S} = (\text{Proc}, R, (\text{In}_p)_{p\in\text{Proc}}, (\text{Out}_p)_{p\in\text{Proc}})$ (qu'on appellera une *structure*) où (Proc, R) est un *graphe de communication* dont les nœuds sont les processus, et dans lequel il existe un arc (orienté) $(p, q) \in R$ si le processus p peut envoyer un signal au processus q, et les tuples $(\text{In}_p)_{p\in\text{Proc}}$ et $(\text{Out}_p)_{p\in\text{Proc}}$ forment l'alphabet des signaux externes (le graphe représenté figure 4.1 est un graphe de communication pour la structure $\mathcal{S} = (\text{Proc}, R, (\text{In}_p)_{p\in\text{Proc}}, (\text{Out}_p)_{p\in\text{Proc}})$ où $\text{Proc} = \{1, 2, 3\}$ et $R = \{(1, 2), (2, 3)\}$). Pour définir un programme satisfaisant une spécification φ, on commence par définir des alphabets de communication $\Sigma^{p,q}$ pour tous les $(p, q) \in R$. L'architecture distribuée induite est donnée par $\mathcal{A} = (\text{Proc}, \Sigma, E, \text{owner})$ avec $\Sigma = \Gamma \cup \bigcup_{(p,q)\in R} \Sigma^{p,q}$ et pour tout $p \in \text{Proc}$, $a \in \text{In}_p \cup \text{Out}_p$, $E(a) = E^{-1}(a) = \{p\}$ et pour tout $(p, q) \in R$, pour tout $a \in \Sigma^{p,q}$, $E^{-1}(a) = E(a) = \{p, q\}$. La fonction owner est définie par, pour tout $p \in \text{Proc}$, pour tout $a \in \text{Out}_p \cup \bigcup_{(p,q)\in R} \Sigma^{p,q}$, $\text{owner}(a) = p$.

Formellement, le problème considéré ici est :

Définition 4.13 (Le problème de SSD asynchrone équitable)**.** *Étant données une structure* $\mathcal{S} = (\text{Proc}, R, (\text{In}_p)_{p\in\text{Proc}}, (\text{Out}_p)_{p\in\text{Proc}})$*, une spécification* φ *sur les ordres partiels étiquetés par* $\Gamma = \text{In} \cup \text{Out}$*, existe-t-il*

- *des alphabets de communication* $(\Sigma^{p,q})_{(p,q)\in R}$ *induisant une architecture distribuée* $\mathcal{A}$*,*
- *et une stratégie distribuée* $F = (f^p)_{p\in\text{Proc}}$ *pour* $\mathcal{A}$*, à mémoire locale, telle que toute exécution* F*-compatible et* F*-équitable* $\alpha \in \mathbb{R}(\Sigma, D)$ *soit telle que* $\pi_\Gamma(\alpha)$ *satisfait la spécification* φ *?*

On dira alors que le couple $((\Sigma^{p,q})_{(p,q)\in R}, F)$ satisfait $(\mathcal{S}, \varphi)$, ou encore que la stratégie distribuée F est une stratégie gagnante pour $(\mathcal{A}, \varphi)$.



(a) La spécification décrit le comportement souhaité

(b) Trois exécutions *concrètes* du système

(c) Les exécutions observables correspondantes

FIG. 4.4 – Les spécifications doivent être closes par extension d'ordre

Remarque 4.14. La notion de stratégie équitable, sauf mention contraire, fait référence à l'équité locale forte (définition 4.10).

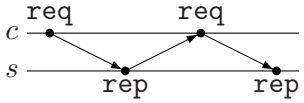
1.2 Les spécifications

1.2.1 Spécifications acceptables

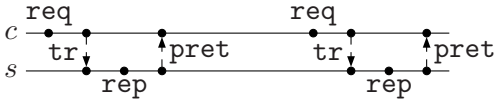
On explique à présent que, bien que l'on se soit déjà restreint aux spécifications sur des ordres partiels, toutes les spécifications ne sont pas pour autant *acceptables* dans ce modèle. On commence par donner un exemple montrant que les spécifications doivent être *closes par extension d'ordre*.

Exemple 4.15. Considérons la structure de la figure 4.1, dans lequel le processus 1 ne peut envoyer de signal directement à 3. Si on suppose que $a \in \mathrm{In}_1$, $b \in \mathrm{Out}_2$ et $c \in \mathrm{Out}_3$, une spécification naturelle peut demander que le processus 2 effectue l'action visible b, et que, *par ailleurs*, si le processus 1 reçoit le signal a de l'environnement, alors le processus 3 doit émettre le signal c en conséquence. L'ordre partiel correspondant est représenté figure 4.4(a) (en effet, la spécification n'imposant aucune relation de causalité entre b et les autres signaux, ceci se traduit dans les modèles d'ordre partiel par une absence d'ordre entre b et les autres signaux). Pour satisfaire cette spécification, le processus 1 doit, après la réception d'un signal a émis par l'environnement, envoyer un signal au processus 3 (signal qui doit nécessairement transiter par 2 de par la structure de l'architecture distribuée) qui, en conséquence, émet le signal observable c. Mais ces signaux *internes* induisent de l'ordre supplémentaire dans les exécutions selon cette stratégie (voir figure 4.4(b)). En conséquence, on obtient les ordres partiels *abstraits* correspondants représentés figure 4.4(c), et on remarque qu'aucun d'eux n'est un modèle de la spécification, bien qu'ils en soient tous des *extensions*. Afin que cette spécification ait un programme distribué la satisfaisant, on doit nécessairement autoriser également les *extensions d'ordre* correspondantes.

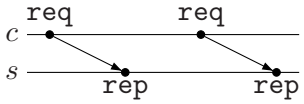
Remarque 4.16. Cet exemple donne une justification *technique* à cette restriction supplémentaires des spécifications. On fait remarquer que, par ailleurs, ne pas se restreindre aux



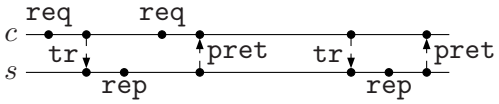
(a) Spécification naïve



(b) Exécution respectant une stratégie correspondante



(c) Spécification affaiblie



(d) Exécution respectant une stratégie correspondante

Fig. 4.5 – Client–Serveur

spécifications closes par extension reviendrait à autoriser la spécification à *empêcher certains processus de communiquer*. En effet, l'exemple ci-dessus met bien ce fait en lumière, pour que deux signaux soient *concurrents* dans une exécution abstraite, il faut qu'il n'y ait eu aucune communication entre les processus concernés entre ces deux événements. Donner un tel pouvoir à la spécification est contraire à la notion de spécification externe que l'on cherche, dans laquelle on ne veut restreindre les possiblités de communication des processus que par la structure donnée en entrée du problème.

On montre à présent qu'une spécification doit également être close par un certain *affaiblissement d'ordre*, i.e., si un ordre partiel α est modèle de la spécification, alors certains ordres partiels dont α est une extension doivent aussi être modèle de la spécification. La raison est que les signaux émis par l'environnement sont des actions *incontrôlables* pour le système. En conséquence, il paraît irréaliste d'essayer d'imposer une relation de causalité *directe* entre une action ayant lieu sur un processus donné, et une action contrôlée par l'environnement sur un autre processus. Par exemple, considérons une structure formée de deux processus, c et s, l'un recevant des requêtes de service d'un client, et l'autre satisfaisant ces requêtes avec les actions externes : $\mathrm{In}_c = \{\texttt{req}\}$ et $\mathrm{Out}_s = \{\texttt{rep}\}$ telle que $R = \{(c,s),(s,c)\}$.

Une spécification naïve pourrait demander une alternance stricte de signaux `req` et `rep`, comme représenté sur la figure 4.5(a). Un exemple d'exécution concrète, utilisant les signaux internes `tr` et `pret`, la satisfaisant est représenté figure 4.5(b). Cependant, les signaux émis par l'environnement étant incontrôlables, on ne peut pas contraindre le second signal `req` à arriver après le signal interne `pret`. Une exécution concrète dans laquelle les processus se comportent de la même façon, mais dans laquelle l'environnement envoie ses signaux à des instants différents est représentée sur la figure 4.5(d), et l'exécution abstraite correspondante sur la figure 4.5(c). On affirme donc qu'une spécification raisonnable ayant pour modèle l'ordre partiel de la figure 4.5(a) doit aussi avoir pour modèle l'ordre partiel de la figure 4.5(c).

Remarque 4.17. La notion d'*affaiblissement* ne concerne que les signaux de l'environnement se passant sur un processus « distant ». On pourrait appliquer le même raisonnement et interdire aux spécifications d'imposer des causalités directes entre un signal et un signal émis par l'environnement, même s'ils arrivent sur le même processus. Cependant, un signal de l'environnement arrivant sur un processus donné est immédiatement connu par ce dernier, et le processus a l'opportunité de changer sa stratégie pour sa prochaine action en fonction du moment où l'action de l'environnement a eu lieu. Dans l'exemple ci-dessus par contre, le processus s ne peut pas changer de stratégie car il ne peut pas immédiatement connaître le

nouveau signal émis par l'environnement.

Il est cependant envisageable de définir une notion plus restrictive des spécifications acceptables, leur interdisant d'imposer qu'un signal incontrôlable soit *causé* (i.e. dans le futur) par un signal contrôlable, même si ces deux derniers mettent en jeu le même processus.

Formellement, si un ordre partiel $t = (X, \leq_t, \lambda)$ satisfait une spécification et si $x \lessdot_t x'$, avec x' un signal émis par l'environnement et x, x' n'étant pas des actions arrivant sur le même processus alors l'*affaiblissement* $s = (X, \leq_s, \lambda)$ défini par $\leq_s = \leq_t \setminus \{(x, x')\}$ doit aussi satisfaire la spécification (on remarque que, comme x' est un successeur de x, $\leq_s$ reste une relation d'ordre).

On définit à présent l'ordre partiel le plus faible induit par t. On rappelle que les actions de Γ sont soit des entrées de l'environnement, soit des sorties vers l'environnement : $\Gamma = \mathrm{In} \cup \mathrm{Out}$, avec $\mathrm{In} = \bigcup_{p \in \mathrm{Proc}} \mathrm{In}_p$ et $\mathrm{Out} = \bigcup_{p \in \mathrm{Proc}} \mathrm{Out}_p$. Soit un ordre partiel $t = (X, \leq, \lambda)$. On définit

$$W_t = \{(x, x') \in X^2 \mid \exists p \in \mathrm{Proc},\ \lambda(x) \notin \Sigma^p \wedge \lambda(x') \in \mathrm{In}_p \wedge x < x' \wedge (\neg \exists y, \lambda(y) \in \mathrm{Out}_p \wedge x < y < x')\}. \tag{4.1}$$

L'ensemble W_t correspond aux paires d'événements (x, x') pour lesquels l'ordre dans t est fortuit. Ceci arrive quand x et x' sont sur des processus différents, et que x' est un signal de l'environnement, sauf s'il existe un événement y entre x et x', correspondant à un signal contrôlable qui se trouve sur le même processus que x'. En effet, l'événement y peut avoir été déclenché par x, donc on doit conserver l'ordre entre x et y (qui peut correspondre à une causalité réelle donc), et on conserve également l'ordre entre y et x' qui ont eu lieu sur le même processus (voir remarque 4.17).

Remarque 4.18. Si $t = (X, \leq_t, \lambda)$ est un ordre partiel, la relation $\leq_s = \leq_t \setminus W_t$ est également une relation d'ordre. En effet, il est facile de voir qu'elle est réflexive et antisymétrique. On peut également montrer que c'est une relation transitive. Soit $x, y, z \in X$, tels que $x \leq_s y$ et $y \leq_s z$. Alors $x \leq_t y$, $y \leq_t z$ et $(x, y) \notin W_t$ et $(y, z) \notin W_t$. De plus, par transitivité de $\leq_t$, $x \leq_t z$. S'il existe $p \in \mathrm{Proc}$ tel que $x, z \in \lambda^{-1}(\Sigma^p)$ alors il est clair que $(x, z) \notin W_t$ et donc $x \leq_s z$. Sinon, soit $p, p' \in \mathrm{Proc}$ tels que $x \in \lambda^{-1}(\Sigma^p)$ et $z \in \lambda^{-1}(\Sigma^{p'})$. Si $\lambda(z) \in \mathrm{Out}_{p'}$, alors on obtient immédiatement $(x, z) \notin W_t$. Si $\lambda(z) \in \mathrm{In}_{p'}$, deux cas sont envisageables. Si $y \in \lambda^{-1}(\Sigma^{p'})$, alors on déduit de l'hypothèse que $x \leq_t y$ et $(x, y) \notin W_t$ qu'il existe $z' \in \lambda^{-1}(\Sigma^{p'})$ tel que $\lambda(z') \in \mathrm{Out}_{p'}$ et $x <_t z' \leq_t y$. Donc il existe $z' \in \lambda^{-1}(\mathrm{Out}_{p'})$ tel que $x <_t z' <_t z$ et on en conclut que $(x, z) \notin W_t$. Sinon, on utilise le fait que $y \leq_s z$, donc il existe à nouveau $z' \in \lambda^{-1}(\mathrm{Out}_{p'})$ tel que $y <_t z' <_t z$, et comme $x \leq_t y$, on a $x <_t z' <_t z$ et donc $(x, z) \notin W_t$. Ainsi, on peut conclure que $x \leq_s z$ et que la relation $\leq_s$ est bien une relation d'ordre.

On définit à présent les spécifications acceptables :

Définition 4.19 (Spécifications acceptables)**.** *Une spécification est* acceptable *si elle est close par extension et affaiblissement d'ordre. Formellement, une spécification φ est acceptable si, pour tout $t = (X, \leq, \lambda)$, ordre partiel étiqueté par Γ tel que, pour tout $p \in \mathrm{Proc}$, $\lambda^{-1}(\Sigma^p)$ est totalement ordonné, si $t \models \varphi$ alors*

- *pour tout $r = (X, \leq_r, \lambda)$ tel que $\leq_t \subseteq \leq_r$, $r \models \varphi$ (clôture par extension),*
- *l'ordre partiel $s = (X, \leq_s, \lambda)$ défini par $\leq_s = \leq_t \setminus W_t$, est tel que $s \models \varphi$ (clôture par affaiblissement).*

On remarque que cette définition de l'affaiblissement supprime toutes les relations d'ordre fortuites dans t à la fois mais, puisque la spécification est également close par extension, tous les ordres partiels intermédiaires sont aussi capturés dans cette définition.

FIG. 4.6 – Un ordre partiel et son extension

1.2.2 AlocTL

Parmi les différentes logiques disponibles pour exprimer des spécifications sur des ordres partiels, on va se concentrer sur les logiques temporelles locales, car elles permettent d'exprimer facilement et intuitivement des spécifications pour des systèmes distribués, et parce qu'elles ont une complexité raisonnable [GK07]. Cependant, elles ne sont pas toutes *acceptables*. Considérons la logique locTL introduite dans [DG01], dont les formules sont évaluées sur les *éléments* d'un ordre partiel, et dont la syntaxe est donnée par la grammaire suivante :

$$\varphi ::= \bot \mid a \in \Sigma \mid \neg\varphi \mid \varphi \vee \varphi \mid \mathsf{EX}\,\varphi \mid \varphi\,\mathsf{U}\,\varphi \mid \varphi\,\mathsf{EU}\,\varphi \mid \mathsf{EY}\,\varphi \mid \varphi\,\mathsf{S}\,\varphi \mid \varphi\,\mathsf{ES}\,\varphi.$$

dans laquelle le symbole $\bot$ signifie *faux*, $\mathsf{EX}\,\varphi$ signifie que φ est vérifiée pour un successeur immédiat de l'élément courant, $\varphi\,\mathsf{U}\,\psi$ signifie que ψ est vérifiée pour un événement plus grand que l'événement courant, et que φ est vérifiée sur *tous* les éléments entre les deux, alors que $\varphi\,\mathsf{EU}\,\psi$ signifie que ψ est vérifiée pour un événement plus grand que l'élément courant et qu'il existe un chemin allant du nœud courant au nœud vérifiant ψ sur lequel φ est toujours vraie. Les modalités EY, S et ES correspondent aux modalités passées duales (pour une définition formelle de la sémantique de locTL, voir [DG01]). Comme un ordre partiel peut avoir plusieurs événements minimaux, se pose la question du point de départ de l'interprétation de la formule. Ce problème est résolu dans [DG01] en ajoutant la notion de *formule initiale* : on vérifie qu'un ordre partiel donné est un modèle d'une formule de locTL *initiale* dont la syntaxe est donnée par

$$\alpha ::= \bot \mid \mathsf{EM}\,\varphi \mid \neg\alpha \mid \alpha \vee \alpha$$

où $t \models \mathsf{EM}\,\varphi$ si et seulement si il existe x élément minimal de t tel que $t, x \models \varphi$.

Remarquons à présent que les formules locTL suivantes : $\mathsf{EM}(a \wedge \neg\,\mathsf{F}\,b)$ ou $\mathsf{EM}(a \wedge \mathsf{EX}\,c)$ ne sont pas closes par extension d'ordre. En effet, pour rester dans la classe des spécifications closes par extension, on doit éliminer les modalités permettant d'*imposer que deux événements soient concurrents*. Si la négation d'une modalité F viole clairement cette contrainte, c'est peut-être moins évident pour le deuxième exemple de spécification. La figure 4.6 met en lumière pourquoi : l'ordre partiel à gauche satisfait la spécification, mais son extension représentée à droite ne la satisfait pas : en fait imposer que l'action c sur le processus 2 soit l'événement successeur de l'action a du processus 1 revient ici à imposer que b soit *concurrent* de a.

Afin d'obtenir la clôture par affaiblisement, on restreint l'usage de la relation d'ordre entre deux événements ayant lieu sur des processus différents aux cas où le plus grand des deux n'est pas un signal émis par l'environnement.

On introduit à présent une restriction *syntaxique* d'une logique temporelle locale basée sur les processus (comme locTL), dans laquelle toutes les formules sont des spécifications acceptables.

La syntaxe de la logique $\mathsf{AlocTL}(\Gamma, \mathrm{Proc})$ (ou simplement AlocTL si Γ et Proc sont évidents par le contexte) est donnée par :

$$\begin{aligned}\varphi ::= a \mid \neg a \mid \neg \mathsf{X}_p \top \mid \neg \mathsf{Y}_p \top \mid \varphi \vee \varphi \mid \varphi \wedge \varphi \\ \mid \mathsf{X}_p \varphi \mid \varphi \,\mathsf{U}_p\, \varphi \mid \mathsf{G}_p \varphi \mid \mathsf{F}_{p,q}(\mathrm{Out} \wedge \varphi) \mid \mathsf{Y}_p \varphi \mid \varphi \,\mathsf{S}_p\, \varphi \mid \mathrm{Out} \wedge \mathsf{H}_{p,q} \varphi\end{aligned}$$

où $a \in \Gamma$ et $p, q \in \mathrm{Proc}$. Les modalités X_p, Y_p, U_p and S_p sont les *next*, *yesterday*, *until* et *since* usuels, seulement restreints aux événements totalement ordonnés du processus p. On peut aussi exprimer dans notre logique *release* (dual de *until*) : $\varphi \mathsf{R}_p \psi = (\mathsf{G}_p\, \psi) \vee (\psi \mathsf{U}_p (\varphi \wedge \psi))$. Lorsqu'on se restreint aux événements du processus p, notre logique a le pouvoir d'expression de LTL ou FO. On restreint uniquement la façon de passer d'un processus à un autre, afin que les clôtures par extension et affaiblissement soient obtenues : pour cela on utilise $\mathsf{F}_{p,q}$ ou $\mathsf{H}_{p,q}$. La première modalité permet de spécifier une propriété réponse déclenchée sur le processus p et dont l'action de réponse est effectuée par le processus q – par exemple $\mathsf{G}_p(\texttt{requete} \rightarrow \mathsf{F}_{p,q}(\mathrm{Out} \wedge \texttt{reponse}))$. La seconde modalité peut être utilisée pour spécifier que certains signaux envoyés à l'environnement doivent avoir une cause – par exemple $\mathsf{G}_p(\texttt{reponse} \rightarrow \mathrm{Out} \wedge \mathsf{H}_{p,q}\, \texttt{requete}))$. On n'inclut pas de modalités de la forme $\mathsf{X}_{p,q}$ car elles ne permettent pas de rester dans la classe des spécifications acceptables (comme illustré par la figure 4.6).

Pour $\varphi \in \mathsf{AlocTL}(\Gamma, \mathrm{Proc})$, la sémantique définit, pour $t = (X, \leq, \lambda)$ un ordre partiel étiqueté par Γ, tel que pour tout $p \in \mathrm{Proc}$ l'ensemble $X^p = \lambda^{-1}(\Sigma^p)$ est totalement ordonné, et pour $x \in X$, à quelle condition $t, x \models \varphi$:

- $t, x \models a \in \Gamma$ si et seulement si $\lambda(x) = a$,
- $t, x \models \neg a$ si et seulement si $\lambda(x) \neq a$,
- $t, x \models \neg \mathsf{X}_p \top$ si et seulement si $x \notin X^p$ ou, pour tout $y \in X^p$, $y \leq x$,
- $t, x \models \neg \mathsf{Y}_p \top$ si et seulement si $x \notin X^p$ ou, pour tout $y \in X^p$, $x \leq y$,
- $t, x \models \mathsf{X}_p \varphi$ si et seulement si $x \in X^p$ et il existe $y \in X^p$ tel que $x < y$, et pour tout $z \in X^p$ $z \leq x$ ou $y \leq z$, et $t, y \models \varphi$,
- $t, x \models \varphi \,\mathsf{U}_p\, \psi$ si et seulement si $x \in X^p$ et il existe $y \in X^p$ tel que $x \leq y$ et $t, y \models \psi$ et pour tout $z \in X^p$, si $x \leq z < y$ alors $t, z \models \varphi$,
- $t, x \models \mathsf{G}_p \varphi$ si et seulement si $x \in X^p$ et, pour tout $y \in X^p$ tel que $x \leq y$, $t, y \models \varphi$,
- $t, x \models \mathsf{F}_{p,q}(\varphi \wedge \mathrm{Out})$ si et seulement si $x \in X^p$ et il existe $y \in X^q$, tel que $x \leq y$, $\lambda(y) \in \mathrm{Out}$ et $t, y \models \varphi$,
- $t, x \models \mathsf{Y}_p \varphi$ si et seulement si $x \in X^p$ et il existe $y \in X^p$, $y < x$ et pour tout $z \in X^p$, $z \leq x$ ou $x \leq z$, et $t, y \models \varphi$,
- $t, x \models \varphi \,\mathsf{S}_p\, \psi$ si et seulement si $x \in X^p$, et il existe $y \in X^p$ tel que $y \leq x$ et $t, y \models \psi$, et pour tout $z \in X^p$, si $y < z \leq x$, alors $t, z \models \varphi$,
- $t, x \models \mathrm{Out} \wedge \mathsf{H}_{p,q} \varphi$ si et seulement si $x \in X^p$, $\lambda(x) \in \mathrm{Out}$ et il existe $y \in X^q$ tel que $y \leq x$ et $t, y \models \varphi$.

Pour déterminer si un ordre partiel donné satisfait la spécification, on doit déterminer où commencer l'évaluation de la formule. Comme mentionné plus haut, contrairement au cas des mots ou des arbres, il peut y avoir plusieurs éléments minimaux à un ordre partiel. Comme dans [DG01], on choisit d'introduire des formules *initiales* pour traiter ce problème.

Dans notre cas, les formules initiales sont données par

$$\alpha ::= \bot \mid \top \mid \neg\, \mathsf{EM}_p \top \mid \mathsf{EM}_p \varphi \mid \alpha \vee \alpha \mid \alpha \wedge \alpha$$

où φ est une formule AlocTL.

Pour $t = (X, \leq, \lambda)$ ordre partiel étiqueté par Γ tel que pour tout $p \in \text{Proc}$ l'ensemble $X^p = \lambda^{-1}(\Sigma^p)$ est totalement ordonné, pour tout α, formule initiale de AlocTL, la sémantique est donnée par

- $t \not\models \bot$,
- $t \models \top$,
- $t \models \mathsf{EM}_p\, \varphi$ si et seulement si il existe $x \in X^p$, tel que pour tout $x' \in X$, si $x' \leq x$ alors $x' \notin X^p$, et $t, x \models \varphi$,

La sémantique des combinaisons booléennes de formules initiales est classique.

Exemple 4.20 (Exemples de formules AlocTL)**.** Considérons la structure représentée sur la figure 4.7.

1. Si on la dote de l'alphabet $\text{In}_p = \{\texttt{req}\}$ et $\text{Out}_q = \{\texttt{rep}\}$, alors la spécification

$$\varphi = \mathsf{EM}_p(\mathsf{G}_p(\texttt{req} \rightarrow \mathsf{F}_{p,q}(\text{Out} \wedge \texttt{rep})))$$

signifie que toute requête reçue par le processus p doit être ultérieurement suivie par une réponse. L'exécution représentée figure 4.7(b) satisfait la spécification. En effet, les deux premières requêtes sur le processus p sont suivies par la deuxième réponse sur le processus q (la causalité étant induite par le signal de p vers q apparaissant entre la deuxième requête et la deuxième réponse), et la troisième requête est suivie par la troisième réponse (la causalité étant induite par le signal de p vers q apparaissant entre la troisième requête et la troisième réponse).

2. Avec le même alphabet externe, on peut écrire également la spécification

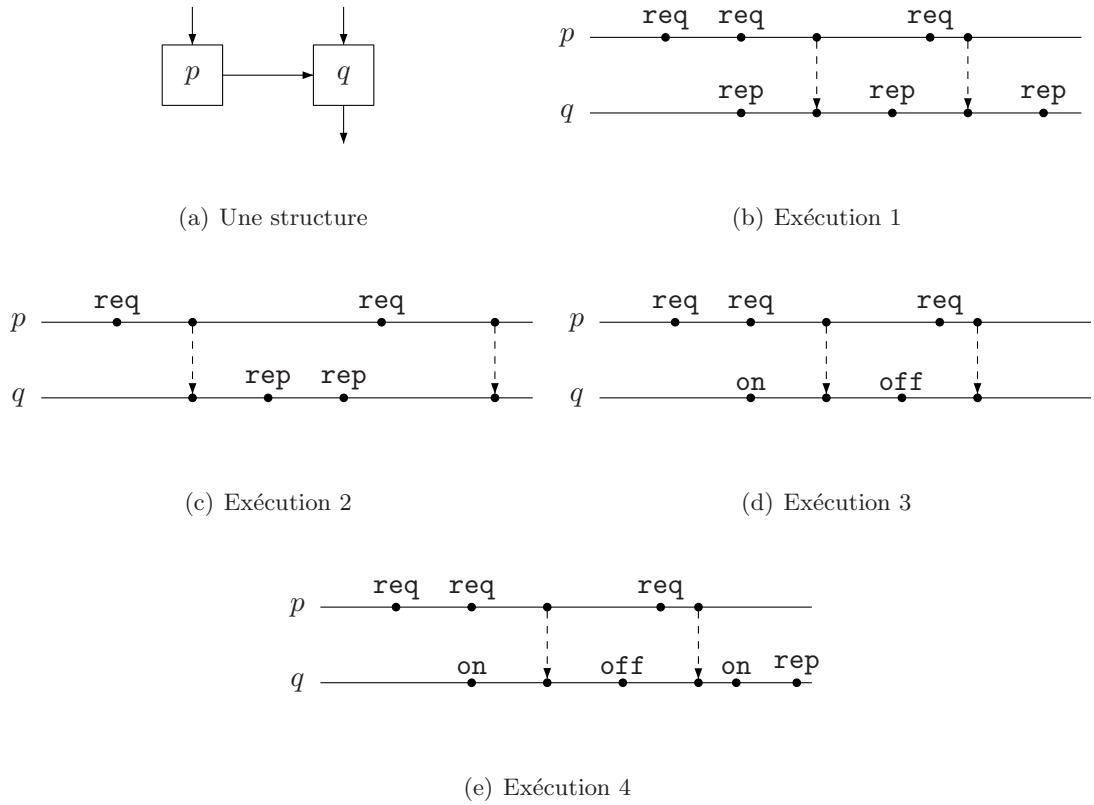
$$\varphi = \mathsf{EM}_q(\mathsf{G}_q(\texttt{rep} \rightarrow (\text{Out} \wedge \mathsf{H}_{q,p}\, \texttt{req}))).$$

Cette spécification signifie que toute réponse donnée par le processus q a été déclenchée par une requête reçue par le processus p. On remarque qu'alors l'exécution observable correspondant à l'exécution concrète représentée figure 4.7(b) ne satisfait pas cette nouvelle spécification. En effet, la première réponse apparaît de façon « spontanée », il n'existe aucun événement dans son passé, donc aucun événement de type req sur le processus p. Par contre, l'exécution abstraite induite par l'exécution représentée figure 4.7(c) satisfait cette spécification, bien qu'elle ne satisfasse plus la spécification précédente (la dernière requête n'étant suivie d'aucune réponse).

3. Si on dote cette structure de l'alphabet $\text{In}_p = \{\texttt{req}\}$, $\text{In}_q = \{\texttt{on}, \texttt{off}\}$ et $\text{Out}_q = \{\texttt{rep}\}$, on peut exprimer la spécification un peu plus complexe suivante :

$$\begin{aligned}\varphi = {} & \mathsf{EM}_q(\neg(\mathsf{G}_q\, \mathsf{F}_q\, \texttt{on})) \vee \mathsf{EM}_p(\mathsf{G}_p(\texttt{req} \rightarrow \mathsf{F}_{p,q}(\text{Out} \wedge \texttt{rep} \wedge (\neg\texttt{off}\, \mathsf{S}_q\, \texttt{on})))) \\ & \wedge \mathsf{EM}_q(\mathsf{G}_q(\texttt{rep} \rightarrow (\text{Out} \wedge \mathsf{H}_{q,p}\, \texttt{req}))).\end{aligned}$$

En fait, on veut à nouveau que toute requête sur le processus p soit suivie par une réponse sur le processus q, mais on ajoute la contrainte que les réponses ne peuvent être envoyées que si le processus q est en configuration on, ce qui arrive lorsque le dernier signal de l'environnement reçu par q est on. Formellement, la sous-formule écrite sur la première ligne est en fait une implication : s'il existe dans le futur de toute action sur le processus q un signal de type on, alors toutes les requêtes sur le processus p sont suivies par une réponse sur le processus q, et cette réponse arrive après un signal on suivi



(a) Une structure

(b) Exécution 1

(c) Exécution 2

(d) Exécution 3

(e) Exécution 4

FIG. 4.7 – Exemple d'architecture et d'exécutions

d'aucun signal `off`. Par ailleurs la deuxième ligne de la spécification demande une fois de plus qu'il n'y ait pas de réponse spontanée, déclenchée par aucune requête. Les deux parties abstraites des exécutions représentées sur les figures 4.7(d) et 4.7(e) satisfont la spécification : la première car $\mathsf{EM}_q(\neg(\mathsf{G}_q\,\mathsf{F}_q\,\texttt{on}))$ est vérifiée : au point `off` il n'y a aucun événement dans le futur sur le processus q vérifiant `on`, la deuxième, car l'unique réponse émise par le processus q est bien dans le futur des trois requêtes reçues par le processus p.

Proposition 4.21. *La logique* AlocTL *est close par extension et affaiblissement.*

Démonstration. Soit $t = (X, \leq_t, \lambda)$ un ordre partiel étiqueté par Γ tel que $X^p = \lambda^{-1}(\Sigma^p)$ est totalement ordonné pour tout $p \in \mathrm{Proc}$, soit $s = (X, \leq_s, \lambda)$ avec $\leq_s = \leq_t \setminus W_t$ l'affaiblissement de t, et soit $r = (X, \leq_r, \lambda)$ tel que $\leq_s \subseteq \leq_r$ une extension de s. On montre par récurrence sur la structure des formules que $t, x \models \varphi$ implique que $r, x \models \varphi$ pour tout $x \in X$.

L'affirmation est claire pour a et $\neg a$, ainsi que pour la disjonction et la conjonction. Comme les restrictions de $\leq_t$ et $\leq_r$ à X^p induisent le même ordre total, l'étape de récurrence est facile pour les modalités X_p, Y_p, G_p, U_p and S_p qui sont restreintes au processus p. Si $t, x \models \mathsf{F}_{p,q}(\mathrm{Out} \wedge \varphi)$ alors $x \in X^p$ et il existe $y \in X^q$ tel que $x \leq_t y$, $\lambda(y) \in \mathrm{Out}$ et $t, y \models \varphi$. Comme y est un événement de type sortie, alors on a $x \leq_s y$ et donc également $x \leq_r y$.

On en déduit immédiatement que $r, x \models \mathsf{F}_{p,q}(\mathrm{Out} \wedge \varphi)$. La preuve est similaire pour le cas $\mathrm{Out} \wedge \mathsf{H}_{i,j}\,\varphi$.

Enfin, t et r ont le même éventuel événement minimal sur le processus p, donc $t \models \mathsf{EM}_p\,\varphi$ implique que $r \models \mathsf{EM}_p\,\varphi$ et $t \models \neg\,\mathsf{EM}_p\,\top$ implique que $r \models \neg\,\mathsf{EM}_p\,\top$. L'implication est donc vraie aussi pour les formules initiales. □

On rappelle que lorsqu'on se restreint aux événements d'un seul processus p, cette logique a le même pouvoir d'expression que LTL ou FO. On restreint uniquement la façon dont on peut passer d'un processus à un autre, afin d'obtenir les clôtures par extension et par affaiblissement. Ainsi, on pourrait également définir une version plus forte d'AlocTL ayant le pouvoir d'expression de MSO sur les formules ne concernant les événements que d'un processus. On a choisi de définir cette logique afin d'avoir un exemple de formalisme pour les spécifications acceptables, suffisamment expressif pour permettre des spécifications intéressantes. Cependant les résultats présentés dans ce chapitre restent vrais pour tout langage de spécification clos par extension et affaiblissement (logiques plus fortes, automates présentant de bonnes propriétés de clôture, etc.)

2 Résultats de décidabilité

Dans cette section, on résoud le problème de SSD asynchrone équitable pour la sous-classe de structures ayant un graphe de communication sous-jacent (Proc, R) fortement connexe : chaque processus peut envoyer des signaux à n'importe quel autre processus (éventuellement en utilisant des processus intermédiaires). Par la suite, nous appellerons plus simplement ces structures des *structures fortement connexes*.

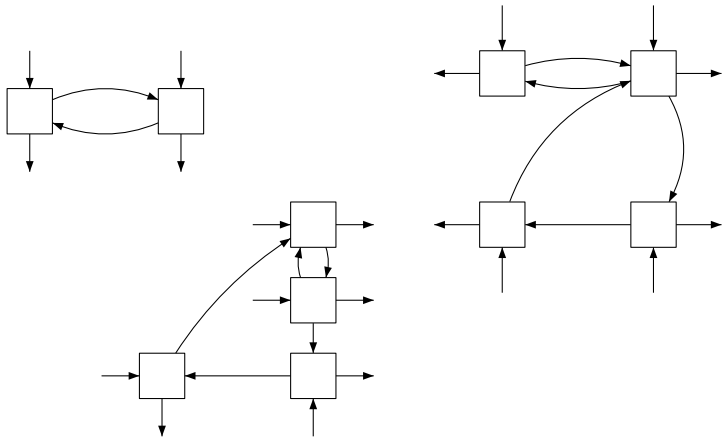


Fig. 4.8 – Des structures fortement connexes

Par la suite, on sera amené à considérer des exécutions observables particulières qui seront des ordres *totaux*.

2.1 Les structures singleton

Une première étape pour résoudre le problème général est de le résoudre pour le cas particulier de structures constituées d'un unique processus : les structures *singleton*. Dans ce

cas précis, il n'y a pas d'action interne, donc $\Sigma = \Gamma = \text{In} \cup \text{Out}$ et toutes les exécutions sont des ordres totaux. De plus les notions de structures et d'architectures induites se confondent. On parlera donc indifféremment d'architectures singleton ou de structures singleton. On assimilera ici les ordres totaux étiquetés par Γ et les mots de Γ^∞. Enfin, puisqu'il n'y a ni extension ni affaiblissement possible, on utilise les logiques classiquement utilisées pour les spécifications.

On va de plus montrer un résultat un peu plus fort que la décidabilité du problème de SSD asynchrone équitable de la définition 4.13 ; on va montrer que, pour toute partition des actions contrôlables, le problème de SSD asynchrone équitable selon la définition d'équité générale 4.7 est décidable. Cette partition des actions contrôlables du processus vis-à-vis de l'équité sera nécessaire pour résoudre le cas général.

Théorème 4.22. *Le problème de SSD asynchrone équitable est décidable pour les architectures singleton, les spécifications ω-régulières, et toute partition des actions contrôlables.*

La fin de cette sous-section est consacrée à la preuve de ce théorème. Soit $\mathcal{A} = (\{p\}, \Gamma, E)$ une architecture singleton. On présente la démonstration pour une formule φ de $\mathsf{LTL}(\Gamma)$, i.e., dont les propositions atomiques sont les actions de Γ.

La démonstration se fait par réduction à la satisfaisabilité de CTL^*. On peut assimiler ici une stratégie du système à une fonction sur les mots $f : \Gamma^* \to 2^\Gamma$ telle que, pour tout $\alpha \in \Gamma^*$, $f(\alpha) \cap \text{In} = \text{In}$ et $|f(\alpha) \cap \text{Out}| \leq 1$. On va construire une formule de $\mathsf{CTL}^*(\Gamma)$ dont on affirme qu'elle est satisfaisable si et seulement si il existe une stratégie gagnante pour $(\mathcal{A}, \varphi)$. Cette formule va s'interpréter sur des arbres d'exécutions selon une stratégie dont l'étiquette aura été enrichie de deux façons : afin de déterminer si une exécution satisfait la spécification, on va faire apparaître les actions (donc la structure de l'arbre) dans l'étiquette. Afin de déterminer si l'arbre est bien un arbre selon une stratégie, et que l'exécution est bien équitable, on doit également faire apparaître la valeur de la stratégie dans l'étiquette.

Pour une stratégie du système $f : \Gamma^* \to 2^\Gamma$, on va définir behavior(f) : $\Gamma^* \to (\{\varepsilon\} \cup \Gamma) \times (\{\#\} \cup \text{Out})$ son arbre d'exécutions enrichi. Moralement, la première composante de l'étiquette va refléter la direction courante de l'arbre, et donc indiquer quelle a été la dernière action jouée, tandis que la deuxième composante de l'étiquette va donner la valeur de la stratégie sur les actions contrôlables après avoir vu l'histoire représentée par le nœud courant. Par exemple, pour un nœud $\alpha \cdot a \in \Gamma^*$ représentant un préfixe d'une exécution f-compatible, behavior(f)$(\alpha \cdot a) = (a, b)$ avec $\{b\} = f(\alpha \cdot a) \cap \text{Out}$.

Formellement, on définit

$$\text{behavior(f)}(\varepsilon) = \begin{cases} (\varepsilon, f(\varepsilon) \cap \text{Out}) & \text{si } f(\varepsilon) \cap \text{Out} \neq \emptyset \\ (\varepsilon, \#) & \text{sinon.} \end{cases}$$

et, pour tout α tel que behavior(f)(α) est déjà défini, pour tout $a \in f(\alpha)$,

$$\text{behavior(f)}(\alpha \cdot a) = \begin{cases} (a, f(\alpha \cdot a) \cap \text{Out}) & \text{si } f(\alpha \cdot a) \cap \text{Out} \neq \emptyset \\ (a, \#) & \text{sinon.} \end{cases}$$

Tel que l'arbre est défini, toutes les branches de behavior(f) sont infinies si $\text{In} \neq \emptyset$. Or, lorsqu'après une histoire $\alpha \in \Gamma^*$, la stratégie ne propose aucune action contrôlable, l'exécution f-compatible finie α est f-maximale, donc f-équitable d'après la remarque 4.8, et doit satisfaire la spécification. On veut donc que la formule de CTL^* que l'on va construire

soit à même de vérifier que la branche *finie* α vérifie φ. Pour cela, il faut matérialiser dans l'arbre cette branche finie, ce que l'on va faire artificiellement en ajoutant une direction à l'arbre behavior(f). Finalement, pour $f : \Gamma^* \to 2^\Gamma$ stratégie du système, on définit behavior(f) : $(\Gamma \cup \{\#\})^* \to (\{\varepsilon, \#\} \cup \Gamma) \times (\{\#\} \cup \mathrm{Out})$ par

$$\mathrm{behavior(f)}(\varepsilon) = \begin{cases} (\varepsilon, f(\varepsilon) \cap \mathrm{Out}) & \text{si } f(\varepsilon) \cap \mathrm{Out} \neq \emptyset \\ (\varepsilon, \#) & \text{sinon.} \end{cases}$$

pour tout $\alpha \in \Gamma^*$ tel que behavior(f)(α) est déjà défini, pour tout $a \in f(\alpha)$,

$$\mathrm{behavior(f)}(\alpha \cdot a) = \begin{cases} (a, f(\alpha \cdot a) \cap \mathrm{Out}) & \text{si } f(\alpha \cdot a) \cap \mathrm{Out} \neq \emptyset \\ (a, \#) & \text{sinon.} \end{cases}$$

et, si $f(\alpha) \cap \mathrm{Out} = \emptyset$,

$$\mathrm{behavior(f)}(\alpha \cdot \#) = (\#, \#)$$

Ainsi les nœuds $\alpha \in \Gamma^* \cdot \{\#\}$ sont des feuilles de l'arbre, matérialisant les exécutions finies f-compatibles et f-équitables.

Exemple 4.23. Considérons une structure singleton ayant comme alphabet d'actions $\mathrm{In} = \{a, b, c, d\}$ et $\mathrm{Out} = \{A, B, C\}$. La stratégie $f : \Gamma^* \to 2^\Gamma$ définie par

$$\begin{aligned} f(\alpha \cdot a) &= \{a, b, c, d, A\} \\ f(\alpha \cdot b) &= \{a, b, c, d, B\} \\ f(\alpha \cdot c) &= \{a, b, c, d, C\} \\ f(\alpha \cdot d) &= \{a, , b, c, d\} \end{aligned}$$

pour tout $\alpha \in \Gamma^*$ est associée à l'arbre behavior(f) dont une partie est représentée figure 4.9.

Remarque 4.24. Quel que soit $\alpha \in \Gamma^\omega$, exécution f-compatible, α est une branche de behavior(f). Quel que soit $\alpha \in \Gamma^*$, exécution f-compatible et f-maximale, $\alpha \cdot \#$ est une branche maximale de behavior(f). Réciproquement, toute branche $\alpha \in \Gamma^\omega$ de behavior(f) est une exécution f-compatible, et toute branche maximale $\alpha \in \Gamma^*$ de behavior(f) est de la forme $\alpha = \alpha' \cdot \{\#\}$ avec α' exécution f-compatible f-maximale.

Soit $\varphi \in \mathsf{LTL}(\Gamma)$. On définit la formule $\overline{\varphi} \in \mathsf{LTL}((\{\varepsilon, \#\} \cup \Gamma) \times (\mathrm{Out} \cup \{\#\}))$ par récurrence sur la structure de φ :

$$\begin{aligned} \overline{a} &= \bigvee_{b \in \mathrm{Out} \cup \{\#\}} (a, b) \text{ pour } a \in \Gamma \\ \overline{\neg\varphi} &= \neg\overline{\varphi} \\ \overline{\varphi \vee \psi} &= \overline{\varphi} \vee \overline{\psi} \\ \overline{\mathsf{X}\,\varphi} &= \mathsf{X}(\overline{\varphi} \wedge \neg(\#, \#)) \\ \overline{\varphi\,\mathsf{U}\,\psi} &= \overline{\varphi}\,\mathsf{U}\,(\overline{\psi} \wedge \neg(\#, \#)) \end{aligned}$$

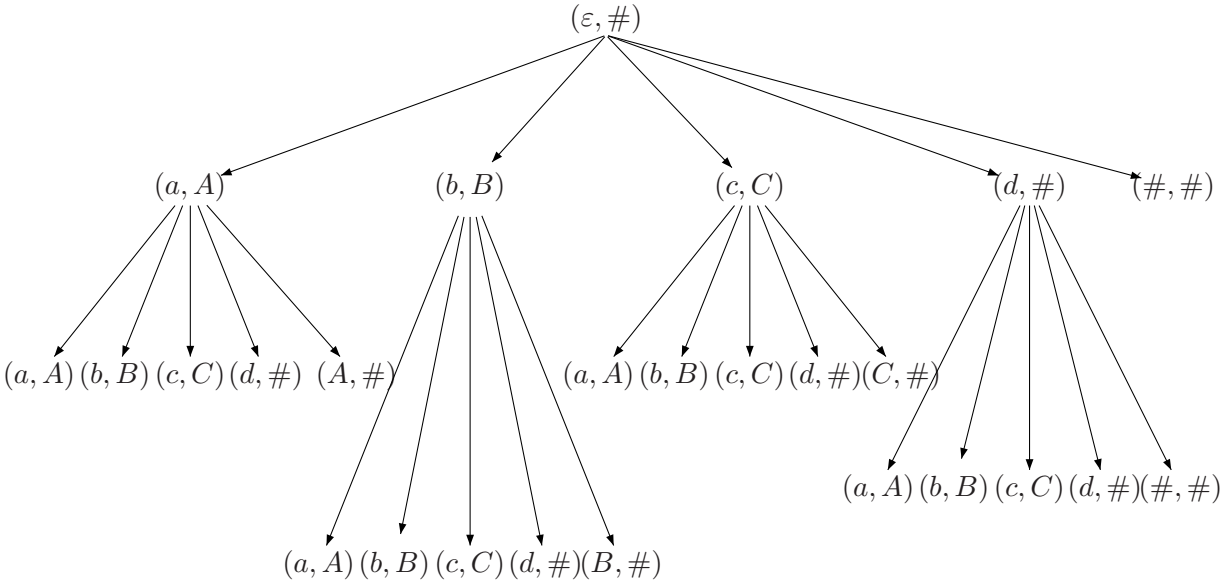


FIG. 4.9 – Un arbre behavior(f)

La formule $\overline{\varphi}$ correspond à la spécification φ interprétée sur les branches de behavior(f). Les transformations de $\mathsf{X}\varphi$ et $\varphi\,\mathsf{U}\,\psi$ correspondent aux exécutions finies qui finissent par $\#$ sur les branches de behavior(f). On veut donc s'assurer que la formule est vérifiée avant d'atteindre la feuille $\#$ artificiellement ajoutée et qui ne correspond à aucun événement de l'exécution représentée.

Par la suite, pour tout élément $(a, b) \in (\{\varepsilon, \#\} \times \Gamma) \times (\{\#\} \cup \mathrm{Out})$, on note $\pi_1(a, b) = a$ la projection sur la première composante, et $\pi_2(a, b) = b$ la projection sur la deuxième composante. Le lemme suivant formalise le fait qu'une exécution f-compatible α satisfait φ si et seulement si la branche correspondante de behavior(f) satisfait $\mathsf{X}\overline{\varphi}$.

Notation 4.25. Pour tout $\alpha \in (\Gamma \cup \{\#\})^\infty$ branche maximale de behavior(f), on définit $\mathrm{run}_f(\alpha)$ de la façon suivante :

$$\mathrm{run}_f(\alpha) = \begin{cases} \pi_1(\text{behavior(f)}(\alpha[1])) \cdot \pi_1(\text{behavior(f)}(\alpha[2])) \cdots & \text{si } \alpha \in \Gamma^\omega \\ \pi_1(\text{behavior(f)}(\alpha[1])) \cdots \pi_1(\text{behavior(f)}(\alpha[n])) & \text{si } \alpha \in \Gamma^n \cdot \{\#\} \end{cases}$$

En fait, $\mathrm{run}_f(\alpha) = \alpha$ si $\alpha \in \Gamma^\omega$ et $\mathrm{run}_f(\alpha \cdot \#) = \alpha$ si $\alpha \in \Gamma^*$.

Lemme 4.26. *Pour tout $f : \Gamma^* \to 2^\Gamma$ stratégie du système, pour tout $\alpha \in (\Gamma \cup \{\#\})^\infty$ branche maximale de* behavior(f)*, on a* behavior(f)$, \alpha, 0 \models \mathsf{X}\overline{\varphi}$ *si et seulement si $run_f(\alpha), 0 \models \varphi$.*

Démonstration. La modalité X apparaît dans la réduction car dans le cas de formules de CTL^*, donc interprétée sur les arbres, l'étiquette de la racine est la lettre initiale, tandis que dans notre réduction, elle ne représente pas la première action d'une exécution. Pour rendre ceci évident, on constate que $\mathsf{X}\overline{\varphi}$ est en fait une formule de $\mathsf{LTL}((\{\varepsilon, \#\} \cup \Gamma) \times (\{\#\} \times \mathrm{Out}))$, et on identifie (behavior(f)$, \alpha$) au modèle de LTL $u : \mathbb{N}_{|\alpha|} \to (\{\varepsilon, \#\} \cup \Gamma) \times (\{\#\} \cup \mathrm{Out})$ tel que, pour tout $i \in \mathbb{N}_{|\alpha|}$, $u(i) =$ behavior(f)$(\alpha[i])$. On rappelle (voir section 1.2.2 page 14) que

$\mathbb{N}_\omega = \mathbb{N}$ et pour tout $n \in \mathbb{N}$, $\mathbb{N}_n = \{0, 1 \dots, n\}$. Il est clair que pour tout $i \in \mathbb{N}_{|\alpha|}$, pour tout $\varphi \in \mathsf{LTL}((\{\varepsilon, \#\} \cup \Gamma) \times (\{\#\} \cup \mathrm{Out}))$, behavior(f), $\alpha, i \models \varphi$ si et seulement si $u, i \models \varphi$.

On identifie également $\mathrm{run}_f(\alpha)$ à l'application $v : \{i \in \mathbb{N} \mid 0 \leq i < |\mathrm{run}_f(\alpha)|\} \to \Gamma$ en posant, pour tout $0 \leq i < |\mathrm{run}_f(\alpha)| = |v|$, $v(i) = \pi_1(\mathrm{behavior(f)}(\alpha[i+1])) = \pi_1(u(i+1))$.

On remarque que si α est fini et tel que $|\alpha| = n + 1$, alors $|u| = n + 2$ et $|v| = n$; si α est infini, $|\alpha| = |u| = |v| = \omega$. Ceci nous permet de montrer par récurrence sur la structure de $\varphi \in \mathsf{LTL}(\Gamma)$, que pour tout $0 \leq i < |\mathrm{run}_f(\alpha)|$, $u, i+1 \models \overline{\varphi}$ si et seulement si $v, i \models \varphi$.

Le cas où $\varphi = a \in \Gamma$ découle directement des définitions. Les cas de combinaisons booléennes sont triviaux.

Supposons maintenant que $u, i+1 \models \overline{\mathsf{X}\,\varphi}$. Alors, $i+2 \in \mathbb{N}_{|\alpha|}$ et $u, i+2 \models \overline{\varphi}$ et $u, i+2 \models \neg(\#, \#)$, donc $i+1 < |\mathrm{run}_f(\alpha)| = |v|$. En appliquant l'hypothèse de récurrence, on obtient donc que $v, i+1 \models \varphi$ donc $v, i \models \mathsf{X}\,\varphi$. Réciproquement, si $v, i \models \mathsf{X}\,\varphi$ alors $i+1 < |v|$ donc $i+3 < |u|$, et $v, i+1 \models \varphi$. Par définition, on en déduit que $u(i+2) \neq (\#, \#)$ (car $(\#, \#)$ étiquette les feuilles de behavior(f)), et par hypothèse de récurrence on a également que $u, i+2 \models \overline{\varphi}$. Donc $u, i+1 \models \overline{\mathsf{X}\,\varphi}$.

Enfin, si $u, i+1 \models \overline{\varphi\,\mathsf{U}\,\psi}$ alors il existe j, tel que $i+1 \leq j+1 < |u|$ et tel que $u, j+1 \models \overline{\psi} \wedge \neg(\#, \#)$ et pour tout $i+1 \leq k+1 < j+1$, $u, k+1 \models \overline{\varphi}$. Comme $u, j+1 \models \neg(\#, \#)$, alors $j+2 < |u|$ et donc $j < |v|$. Par hypothèse de récurrence, on a $v, j \models \psi$ et pour tout $i \leq k < j$, $v, k \models \varphi$. Réciproquement, si $v, i \models \varphi\,\mathsf{U}\,\psi$, alors il existe $i \leq j < |v|$ tel que $v, j \models \psi$ et pour tout $i \leq k < j$, on a $v, k \models \varphi$. Comme $j < |v|$, alors $j+2 < |u|$, et donc $u, j+1 \models \neg(\#, \#)$, et par hypothèse de récurrence, on a $u, j+1 \models \overline{\psi}$ et pour tout $i+1 \leq k+1 < j+1$, on a $u, k+1 \models \overline{\varphi}$, donc $u, i+1 \models \overline{\varphi\,\mathsf{U}\,\psi}$. □

On affirme qu'une stratégie f est gagnante pour $(\mathcal{A}, \varphi)$ et une partition $\mathcal{P}$ des actions contrôlables Out si et seulement si behavior(f) est un modèle de la formule de $\mathsf{CTL}^*((\{\varepsilon, \#\} \cup \Gamma) \times (\mathrm{Out} \cup \{\#\}))$ suivante

$$\widetilde{\varphi} = \mathsf{AG}\,\mathrm{Compat} \wedge \mathsf{AG}\,\mathrm{Complet} \wedge \mathsf{A}(\mathrm{Fair} \to \mathsf{X}\,\overline{\varphi}) \wedge \varepsilon_1$$

avec

$$\mathrm{Compat} = \#_1 \to (\#_2 \wedge \neg\,\mathsf{X}\,\top)$$

$$\mathrm{Complet} = \neg\#_1 \to \Big(\bigwedge_{a \in \mathrm{In}} \mathsf{EX}\,a_1 \wedge \bigvee_{b \in \mathrm{Out} \cup \{\#\}} (b_2 \wedge \mathsf{EX}\,b_1)\Big)$$

$$\mathrm{Fair} = \bigwedge_{\Sigma' \in \mathcal{P}} (\mathsf{FG}(\bigvee_{b \in \Sigma'} b_2) \to (\mathsf{GX}\,\top \wedge \mathsf{GF} \bigvee_{b \in \Sigma'} b_1))$$

où, pour tout $a \in \{\varepsilon, \#\} \cup \Gamma$, la formule $a_1 = \bigvee_{b \in (\mathrm{Out} \cup \{\#\})}(a, b)$ et, pour tout $b \in \{\#\} \cup \mathrm{Out}$, $b_2 = \bigvee_{a \in \{\varepsilon, \#\} \cup \Gamma}(a, b)$.

La sous-formule Compat assure qu'un nœud étiqueté par # sur sa première composante est bien une feuille de l'arbre considéré, alors que Complet vérifie que tout nœud étiqueté par $\{\varepsilon\} \cup \Gamma$ sur sa première composante a au moins $|\mathrm{In}| + 1$ fils, un pour chaque entrée possible, et un pour l'action contrôlable choisie par la stratégie (ou # si la stratégie ne propose aucune action contrôlable). Avec la sous-formule Complet on s'assure donc que toutes les exécutions f-compatibles apparaissent dans l'arbre. Enfin, Fair est une formule sur les chemins vérifiant que l'exécution est équitable. On remarque que les chemins finis f-maximaux satisfont trivialement Fair car violent la prémisse de l'implication : dans l'arbre behavior(f) ces chemins se finissent

par $\# \notin \Sigma$. La clause ε_1 est ajoutée pour des simplifications techniques, et pourrait être supprimée. Elle vise à s'assurer que tous les modèles de $\widetilde{\varphi}$ ont la même valeur sur la première composante de l'étiquette de la racine.

À présent, on fixe la partition $\mathcal{P}$ en fonction de laquelle l'équité est assurée, et on dit qu'une stratégie $f : \Gamma^* \to 2^\Gamma$ est gagnante pour $(\mathcal{A}, \varphi)$ si toutes ses exécutions f-équitables vis-à-vis de $\mathcal{P}$ (voir la définition 4.7 donc) satisfont φ.

Proposition 4.27. *Soit $f : \Gamma^* \to 2^\Gamma$ une stratégie du système gagnante pour $(\mathcal{A}, \varphi)$, avec $\varphi \in \mathsf{LTL}(\Gamma)$. Alors* behavior(f) $\models \widetilde{\varphi}$.

Démonstration.
- Soit $\alpha \in \Gamma^\omega \cup \Gamma^* \cdot \{\#\}$ une branche maximale de behavior(f) et soit $i \in \mathbb{N}_{|\alpha|}$. Supposons que $\pi_1(\text{behavior(f)}(\alpha[i])) = \#$. Alors par définition, $\alpha[i]$ est une feuille de behavior(f) et behavior(f)$(\alpha[i]) = (\#, \#)$. Donc behavior(f) $\models \mathsf{AG}\,\text{Compat}$.
- Soit $\alpha \in \Gamma^\omega \cup \Gamma^* \cdot \{\#\}$ une branche maximale de behavior(f) et soit $i \in \mathbb{N}_{|\alpha|}$. Si $\pi_1(\text{behavior(f)}(\alpha[i])) \neq \#$, alors $\alpha[i] \in \Gamma^* \cap \mathit{dom}(\text{behavior(f)})$ et, par définition, pour tout $a \in f(\alpha[i])$, $\pi_1(\text{behavior(f)}(\alpha[i] \cdot a)) = a$. En particulier, pour tout $a \in \text{In}$, il existe $\alpha' \in \Gamma^\omega \cup \Gamma^* \cdot \{\#\}$ tel que $\alpha'[1] = a$ et $\alpha[i] \cdot \alpha'$ est une branche maximale de behavior(f). Alors behavior(f)$, \alpha[i]\alpha', i+1 \models a_1$. De même, soit $b = \pi_2(\text{behavior(f)}(\alpha[i])) \in \text{Out} \cup \{\#\}$, alors par définition, behavior(f)$, \alpha, i \models \mathsf{EX}\, b_1$. Donc behavior(f) $\models \mathsf{AG}\,\text{Complet}$.
- Soit $\alpha \in \Gamma^\omega \cup \Gamma^* \cdot \{\#\}$ une branche maximale de behavior(f) telle que behavior(f)$, \alpha, 0 \models$ Fair. Si $\alpha \in \Gamma^* \cdot \{\#\}$, alors $\text{run}_f(\alpha)$ est f-maximal, donc équitable. Dans ce cas, $\text{run}_f(\alpha) \models \varphi$, et par le lemme 4.26, behavior(f)$, \alpha \models \mathsf{X}\,\overline{\varphi}$. Supposons maintenant que $\alpha \in \Gamma^\omega$. Soit $\Sigma' \in \mathcal{P}$ et $i \in \mathbb{N}_{|\alpha|}$ vérifiant, pour tout $j \in \mathbb{N}_{|\alpha|}$, si $j \geq i$, alors $\pi_2(\text{behavior(f)}(\alpha[j])) \in \Sigma'$. Pour tout $i \in \mathbb{N}_{|\alpha|}$, il existe alors $j \geq i$ tel que la projection $\pi_1(\text{behavior(f)}(\alpha[j])) \in \Sigma'$. On en déduit donc que l'exécution α est f-équitable pour la partition considérée et, comme f est une stratégie gagnante pour $(\mathcal{A}, \varphi)$, on a $\alpha \models \varphi$. Comme $\text{run}_f(\alpha) = \alpha$, le lemme 4.26 permet de conclure que behavior(f)$, \alpha \models \mathsf{X}\,\overline{\varphi}$. Donc behavior(f) $\models \mathsf{A}(\text{Fair} \to \mathsf{X}\,\overline{\varphi})$.
- Par construction, behavior(f) $\models \varepsilon_1$.

Donc behavior(f) $\models \widetilde{\varphi}$. □

Proposition 4.28. *Si $\widetilde{\varphi}$ a un modèle, alors il existe une stratégie gagnante pour $(\mathcal{A}, \varphi)$.*

Démonstration. Soit D un domaine fini et $t : D^* \to (\{\varepsilon, \#\} \cup \Gamma) \times (\{\#\} \cup \text{Out})$ un modèle de $\widetilde{\varphi}$. On montre qu'on peut alors obtenir un autre modèle de $\widetilde{\varphi}$ dont l'ensemble des directions est exactement $\{\#\} \cup \Gamma$. Pour cela on définit par récurrence une fonction $\Phi : (\{\#\} \cup \Gamma)^* \to D^*$ telle que si $\alpha \in \mathit{dom}(\Phi)$ alors $\Phi(\alpha) \in \mathit{dom}(t)$:

$$\Phi(\varepsilon) = \varepsilon,$$

et, pour tout $\alpha \in \Gamma^*$ tel que $\Phi(\alpha)$ est déjà défini, pour tout $a \in \text{In} \cup \pi_2(t(\Phi(\alpha)))$,

$$\Phi(\alpha \cdot a) = \Phi(\alpha) \cdot d$$

avec $d \in D$ tel que $\pi_1(t(\Phi(\alpha) \cdot d)) = a$ (un tel d existe car $t \models \mathsf{AG}\,\text{Complet}$).

Pour tout $a \in (\text{Out} \cup \{\#\}) \setminus \{\pi_2(t(\Phi(\alpha)))\}$, $\Phi(\alpha \cdot a)$ n'est pas défini. Donc les seuls cas où $\alpha \cdot \# \in \mathit{dom}(\Phi)$ correspondent aux cas où $\pi_2(t(\Phi(\alpha)) = \#$.

On définit maintenant l'arbre $t' : (\Gamma \cup \{\#\})^* \to (\{\varepsilon, \#\} \cup \Gamma) \times (\{\#\} \cup \text{Out})$ de la façon suivante : pour tout $\alpha \in \mathit{dom}(\Phi)$,

$$t'(\alpha) = t(\Phi(\alpha)).$$

.

L'arbre t' est donc isomorphe à un *sous-arbre* de t, et vérifie par construction, pour tout $\alpha \in \Gamma^*$ et tout $a \in \Gamma \cup \{\#\}$ tels que $\alpha \cdot a \in dom(t')$, $\pi_1(t'(\alpha \cdot a)) = a$. On étend Φ aux mots infinis en posant, pour tout $\alpha \in \Gamma^\omega$ branche maximale de t', $\Phi(\alpha) = \bigsqcup_{\alpha' \sqsubseteq \alpha} \Phi(\alpha')$.

Remarque 4.29. Pour tout $\alpha \in (\Gamma \cup \{\#\})^\infty$ branche *maximale* de t', $\Phi(\alpha)$ est une branche *maximale* de t, et de plus, $|\alpha| = |\Phi(\alpha)|$.

Ceci découle du fait que, comme $t \models \mathsf{AG}\,\mathrm{Compat}$, pour tout $\rho \in D^*$ tel que $\pi_1(t(\rho)) = \#$, ρ est une branche maximale de t.

De plus, en appliquant les définitions, on obtient immédiatement que

Remarque 4.30. Pour tout $\alpha \in (\Gamma \cup \{\#\})^\infty$ branche *maximale* de t', pour tout $i \in \mathbb{N}_{|\alpha|}$, pour tout $\psi \in \mathsf{LTL}((\{\varepsilon, \#\} \cup \Gamma) \times (\{\#\} \cup \mathrm{Out}))$, on a $t', \alpha, i \models \psi$ si et seulement si $t, \Phi(\alpha), i \models \psi$.

On définit à présent $f : \Gamma^* \to 2^\Gamma$ stratégie du système en posant, pour tout $\alpha \in \Gamma^*$,

$$f(\alpha) \cap \mathrm{In} = \mathrm{In}$$
$$f(\alpha) \cap \mathrm{Out} = \{\pi_2(t'(\alpha))\} \text{ si } \alpha \in dom(t') \text{ et } \pi_2(t'(\alpha)) \neq \#.$$

Pour montrer que f est une stratégie gagnante, il reste à montrer que $t' \models \widetilde{\varphi}$ et que behavior(f) $= t'$. En effet, si tel est le cas, alors soit $\alpha \in \Gamma^\omega$ une exécution f-compatible et f-équitable. Dans ce cas, comme on l'a vu dans la remarque 4.24, α est une branche de behavior(f). De plus, behavior(f), $\alpha, 0 \models$ Fair. Donc comme behavior(f) $= t' \models \widetilde{\varphi}$, alors behavior(f), $\alpha, 0 \models \mathsf{X}\,\overline{\varphi}$ et, par le lemme 4.26, $\mathrm{run}_f(\alpha) = \alpha \models \varphi$. Soit maintenant $\alpha \in \Gamma^*$ une exécution f-compatible et f-maximale. Elle est donc f-équitable. Comme mentionné dans la remarque 4.24, $\alpha \cdot \{\#\}$ est une branche maximale de behavior(f), et behavior(f), $\alpha \cdot \{\#\}, 0 \models$ Fair. Donc, une fois de plus, en utilisant le fait que $t' =$ behavior(f), que $t' \models \widetilde{\varphi}$ et le lemme 4.26, on obtient que $\mathrm{run}_f(\alpha \cdot \{\#\}) = \alpha \models \varphi$. Donc, toute exécution f-compatible et f-équitable satisfaisant la spécification, on en déduit que f est une stratégie gagnante pour $(\mathcal{A}, \varphi)$.

Montrons à présent que $t' \models \widetilde{\varphi}$:

- Soit $\alpha \in (\{\#\} \cup \Gamma)^\infty$ une branche maximale de t', et soit $i \in \mathbb{N}_{|\alpha|}$. Si $t', \alpha, i \models \#_1$, alors $\pi_1(t'(\alpha[i])) = \pi_1(t(\Phi(\alpha[i]))) = \#$. Soit $\rho \in D^\infty$ une branche maximale de t telle que $\rho[i] = \Phi(\alpha[i])$. Alors $t, \rho, i \models \#_1$ et, comme $t \models \widetilde{\varphi}$, alors $t, \rho, i \models \#_2$. Donc, par définition de t', on a $t', \alpha, i \models \#_2$. De plus, $\alpha[i] \in \Gamma^* \cdot \{\#\}$, donc pour tout $a \in \Gamma \cup \{\#\}$, $\alpha[i] \cdot a \notin dom(\Phi)$, donc $\alpha[i] \cdot a \notin dom(t')$, et $t', \alpha, i \models \neg\,\mathsf{X}\top$. Ainsi, $t' \models \mathsf{AG}\,\mathrm{Compat}$.
- Soit $\alpha \in (\{\#\} \cup \Gamma)^\infty$ une branche maximale de t', et soit $i \in \mathbb{N}_{|\alpha|}$. Si $t', \alpha, i \models \neg\#_1$, on en déduit que $\alpha[i] \in \Gamma^*$ et, par définition de Φ et de t', pour tout $a \in \mathrm{In}$, $\alpha[i] \cdot a \in dom(\Phi)$ et $\pi_1(t'(\alpha[i] \cdot a)) = a$. Donc $t', \alpha, i \models \bigwedge_{a \in \mathrm{In}} \mathsf{EX}\,a$. Soit $b \in \mathrm{Out} \cup \{\#\}$ tel que $\pi_2(t'(\alpha[i])) = b$. Alors par définition, $\pi_2(t(\Phi(\alpha[i]))) = b$, $\alpha[i] \cdot b \in dom(\Phi)$ et $\pi_1(t'(\alpha[i] \cdot b)) = b$. Donc $t', \alpha, i \models \bigvee_{b \in \mathrm{Out} \cup \{\#\}} (b_2 \wedge \mathsf{EX}\,b_1)$, et $t' \models \mathsf{AG}\,\mathrm{Complet}$.
- Soit $\alpha \in (\{\#\} \cup \Gamma)^\infty$ une branche maximale de t' telle que $t', \alpha, 0 \models$ Fair, alors, d'après la remarque 4.30, $t, \Phi(\alpha), 0 \models$ Fair. Comme $t \models \widetilde{\varphi}$, alors $t, \Phi(\alpha), 0 \models \mathsf{X}\,\overline{\varphi}$ et, à nouveau par la remarque 4.30, on a $t', \alpha, 0 \models \mathsf{X}\,\overline{\varphi}$.

Montrons enfin que $t' =$ behavior(f) : on va montrer par récurrence sur la taille de $\alpha \in (\Gamma \cup \{\#\})^*$ que $t'(\alpha) =$ behavior(f)(α).

$$\begin{aligned}\text{behavior(f)}(\varepsilon) &= \begin{cases}(\varepsilon, b) & \text{si } f(\varepsilon) \cap \text{Out} = \{b\}\\ (\varepsilon, \#) & \text{si } f(\varepsilon) \cap \text{Out} = \emptyset.\end{cases}\\ &= t'(\varepsilon).\end{aligned}$$

Soit $\alpha \in dom(\text{behavior(f)})$. Alors, par hypothèse de récurrence, $\alpha \in dom(t')$ et $t'(\alpha) = \text{behavior(f)}(\alpha)$. Par définition de f, $f(\alpha) = \text{In} \cup \{\pi_2(t'(\alpha))\} \setminus \{\#\}$. Soit $a \in f(\alpha)$. Alors

$$\begin{aligned}\text{behavior(f)}(\alpha \cdot a) &= \begin{cases}(a, b) & \text{si } f(\alpha \cdot a) \cap \text{Out} = \{b\}\\ (a, \#) & \text{si } f(\alpha \cdot a) \cap \text{Out} = \emptyset.\end{cases}\\ &= (a, \pi_2(t'(\alpha \cdot a)))\\ &= t'(\alpha \cdot a).\end{aligned}$$

Si $f(\alpha) \cap \text{Out} = \emptyset$, alors $\pi_2(t'(\alpha)) = \pi_2(t(\Phi(\alpha)) = \#$ et

$$\begin{aligned}\text{behavior(f)}(\alpha \cdot \#) &= (\#, \#)\\ &= t'(\alpha \cdot \#).\end{aligned}$$

De même, si $\alpha \in dom(t')$ alors par hypothèse de récurrence, $\alpha \in dom(\text{behavior(f)})$ et $t'(\alpha) = \text{behavior(f)}(\alpha)$. Soit $a \in \{\#\} \cup \Gamma$ tel que $\alpha \cdot a \in dom(t')$. Alors $a \in \text{In} \cup \pi_2(t'(\alpha))$, et il est clair que $\alpha \cdot a \in dom(\text{behavior(f)})$ et que $t'(\alpha \cdot a) = \text{behavior(f)}(\alpha \cdot a)$. □

On a donc donné une réduction polynomiale du problème de SSD asynchrone équitable pour les architectures singleton et les spécifications de $\mathsf{LTL}(\Gamma)$ au problème de satisfaisabilité de CTL^*, ce qui nous permet de conclure que la complexité du problème est 2EXPTIME, car la satisfaisabilité de CTL^* est un problème 2EXPTIME-complet ([VS85, EJ99]). De plus, s'il existe un modèle de $\widetilde{\varphi}$ alors il existe un modèle représentant une stratégie à mémoire finie (i.e., calculable par un automate fini).

On peut montrer que cette complexité est également une borne inférieure :

Proposition 4.31. *Le problème de SSD asynchrone équitable pour les architectures singleton et les spécifications de* $\mathsf{LTL}(\Gamma)$ *est 2EXPTIME-complet.*

Démonstration. On a déjà montré que le problème est dans 2EXPTIME. La démonstration de la 2EXPTIME-difficulté se fait par réduction du problème de synthèse de système centralisé synchrone sans délai de [PR89a], qui est 2EXPTIME-complet pour les spécifications LTL (voir théorème 2.31 page 32).

On a montré au début du chapitre précédent (section 1 page 52) que les données du problème de synthèse synchrone tel que présenté dans la définition 2.29 page 31 pouvaient être présentées de façon simplifiée. Soit donc $\mathcal{A} = (\text{Proc}, V, E, (S^v)_{v \in V}, s_0, (d_p)_{p \in \text{Proc}})$ une architecture à synthétiser, telle que $\text{Proc} = \{p\}$, et $d_p = 0$, et soit $\varphi \in \mathsf{LTL}(V)$ une spécification. On va construire une architecture singleton $\mathcal{A}' = (\text{Proc}, \Gamma, E')$ et une spécification $\varphi' \in \mathsf{LTL}(\Gamma)$ tels qu'il existe une stratégie $f : (S^{V_\text{I}})^+ \to S^{V_\text{O}}$ gagnante pour $(\mathcal{A}, \varphi)$ si et seulement si il existe une stratégie $f' : \Gamma^* \to 2^\Gamma$ gagnante pour $(\mathcal{A}', \varphi')$ et la partition la plus grossière $\mathcal{P} = \text{Out}$.

Afin d'avoir à modifier le moins possible la spécification au cours de la réduction, on va prendre comme actions externes de l'architecture $\mathcal{A}'$ simplement les valeurs des variables. Formellement, on a

$$\begin{aligned} \mathrm{Proc} &= \{p\} \\ \mathrm{In} &= S^{V_\mathrm{I}} \\ \mathrm{Out} &= S^{V_\mathrm{O}} \\ E &= (\Gamma \times \mathrm{Proc}) \cup (\mathrm{Proc} \times \Gamma) \\ \mathrm{owner}(a) &= \{p\} \text{ pour tout } a \in \mathrm{Out}. \end{aligned}$$

Une action de l'environnement est simplement la valeur qu'il décide de donner aux variables d'entrée, et une action du processus correspond à la valeur qu'il décide de donner aux variables de sortie.

Par ailleurs, la spécification φ porte sur les valeurs des états du système de transition $TS_{\mathcal{A}}$ associé à $\mathcal{A}$. Les propositions atomiques sont donc des éléments de S^V. La spécification φ' restreignant les comportements de l'architecture $\mathcal{A}'$ porte sur les mots de Γ^∞. Les propositions atomiques sont donc des éléments de $S^{V_\mathrm{I}} \cup S^{V_\mathrm{O}}$. De plus, les exécutions de $\mathcal{A}'$ que l'on cherche à obtenir doivent simuler les exécutions synchrones de $\mathcal{A}$. Elles doivent donc être une alternance stricte d'actions de l'environnement et d'actions du système. Par ailleurs, elles doivent être infinies ; la stratégie du système $\mathcal{A}'$ doit donc être toujours définie, sous peine de voir une exécution maximale finie, qui ne serait pas une exécution du système synchrone correspondant. On définit donc la spécification $\varphi' \in \mathsf{LTL}(\Gamma)$ suivante

$$\varphi' = \mathsf{G}\mathsf{X}\top \wedge (s_0^{V_\mathrm{I}} \wedge (\mathsf{G}(\mathrm{In} \to \mathsf{X}\,\mathrm{Out})) \wedge (\mathsf{G}(\mathrm{Out} \to \mathsf{X}\,\mathrm{In}))) \to (\mathsf{X}\,s_0^{V_\mathrm{O}} \wedge \overline{\varphi}))$$

où $\overline{\varphi}$ est défini par

$$\begin{aligned} \overline{s} &= s^{V_\mathrm{I}} \wedge \mathsf{X}\,s^{V_\mathrm{O}} \text{ si } s \in S^V \\ \overline{\neg\varphi} &= \neg\overline{\varphi} \\ \overline{\varphi \vee \psi} &= \overline{\varphi} \vee \overline{\psi} \\ \overline{\mathsf{X}\,\varphi} &= \mathsf{X}(\mathsf{X}\,\overline{\varphi}) \\ \overline{\varphi\,\mathsf{U}\,\psi} &= (\mathrm{In} \to \overline{\varphi})\,\mathsf{U}\,(\mathrm{In} \wedge \overline{\psi}) \end{aligned}$$

et $\mathrm{In} \stackrel{\text{def}}{=} \bigvee_{a \in \mathrm{In}} a$, et $\mathrm{Out} \stackrel{\text{def}}{=} \bigvee_{a \in \mathrm{Out}} a$.

Remarque 4.32. Pour toutes séquences $\sigma = s_0 s_1 \cdots \in (S^V)^\omega$ et $\alpha' = a'_0 a'_1 \cdots \in \Gamma^\omega$ vérifiant pour tout $i \geq 0$, $s_i^{V_\mathrm{I}} = a'_{2i}$ et $s_i^{V_\mathrm{O}} = a'_{2i+1}$, alors, pour tout $i \geq 0$, pour tout $\varphi \in \mathsf{LTL}(V)$, on a $\sigma, i \models \varphi$ si et seulement si $\alpha', 2i \models \overline{\varphi}$.

En effet, soit $i \geq 0$, et $s \in S^V$, alors $\sigma, i \models s$ si et seulement si $s_i = s$ si et seulement si $a'_{2i} = s^{V_\mathrm{I}}$ et $a'_{2i+1} = s^{V_\mathrm{O}}$, si et seulement si $\alpha', 2i \models s^{V_\mathrm{I}} \wedge \mathsf{X}\,s^{V_\mathrm{O}}$. Les cas où φ est construit à partir de combinaisons booléennes sont triviaux. Si $\sigma, i \models \mathsf{X}\,\varphi$ alors $\sigma, i+1 \models \varphi$ et par hypothèse de récurrence $\alpha', 2i+2 \models \overline{\varphi}$ et $\alpha', 2i \models \mathsf{X}(\mathsf{X}\,\overline{\varphi})$. Réciproquement, si $\alpha', 2i \models \mathsf{X}(\mathsf{X}\,\varphi)$, alors par définition $\alpha', 2i+2 \models \varphi$ et par hypothèse de récurrence $\sigma, i+1 \models \varphi$ donc $\sigma, i \models \mathsf{X}\,\varphi$. Enfin, si $\sigma, i \models \varphi\,\mathsf{U}\,\psi$, alors il existe $j \geq i$ tel que $\sigma, j \models \psi$ et pour tout $i \leq k < j$ on a $\sigma, k \models \varphi$. Par hypothèse de récurrence, ceci implique que $\alpha', 2j \models \overline{\psi}$ et pour tout $i \leq k < j$ on a $\alpha', 2k \models \overline{\varphi}$. Par construction, $a'_{2j} \in \mathrm{In}$, donc $\alpha', 2j \models \mathrm{In} \wedge \overline{\psi}$, et pour tout $i \leq k < j$, $a'_{2k} \in \mathrm{In}$

et $a'_{2k+1} \notin \text{In}$. Alors, pour tout $2i \leq k < 2j$, on a $\alpha', k \models \text{In} \rightarrow \overline{\varphi}$. Donc $\alpha', 2i \models \overline{\varphi \,\mathsf{U}\, \psi}$. Réciproquement, si $\alpha', 2i \models \overline{\varphi \,\mathsf{U}\, \psi}$, alors il existe $j \geq 2i$ tel que $\alpha', j \models \text{In} \wedge \psi$ et pour tout $2i \leq k < j$, $\alpha', k \models \text{In} \rightarrow \overline{\varphi}$. Par définition de α', si $\alpha', j \models \text{In}$ alors il existe $j' \geq i$ tel que $j = 2j'$ et l'hypothèse de récurrence permet de conclure que $\sigma, j' \models \psi$. De même, pour tout $2i \leq k < j$, si $\alpha', k \models \text{In}$ alors il existe k' tel que $k = 2k'$. Comme par ailleurs, $\alpha', k \models \overline{\varphi}$, une fois de plus, l'hypothèse de récurrence permet de conclure que $\sigma, k' \models \varphi$, et $\alpha, i \models \varphi \,\mathsf{U}\, \psi$.

Soit $f : (S^{V_\text{I}})^+ \rightarrow S^{V_\text{O}}$ une stratégie gagnante pour $(\mathcal{A}, \varphi)$. On construit $f' : \Gamma^* \rightarrow 2^\Gamma$ en posant, pour tout $\alpha \in \Gamma^*$, pour tous $a, a_0, a_1 \in \Gamma$,

$$f'(\varepsilon) = f'(a) = S^{V_\text{I}} \cup \{s_0^{V_\text{O}}\}$$

$$f'(a_0 \cdot a_1 \cdot \alpha) = \begin{cases} S^{V_\text{I}} \cup \{f(\pi_\text{In}(\alpha))\} & \text{si } \alpha \in (\text{In} \cdot \text{Out})^* \cdot \text{In et } a_0 = s_0^{V_\text{I}} \text{ et } a_1 = s_0^{V_\text{O}} \\ S^{V_\text{I}} \cup \{s_0^{V_\text{O}}\} & \text{sinon.} \end{cases}$$

Soit $\alpha' = a'_0 a'_1 \cdots \in \Gamma^\infty$ une exécution f'-compatible et f'-équitable pour $\mathcal{P}$. Alors, comme pour tout $\alpha \in \Gamma^*$ fini, $f'(\alpha) \cap \text{Out} \neq \emptyset$, une exécution finie ne peut pas être f-maximale, donc $\alpha' \in \Gamma^\omega$ et $\alpha' \models \mathsf{G}\,\mathsf{X}\top$. Supposons que $\alpha' \models (s_0^{V_\text{I}} \wedge (\mathsf{G}(\text{In} \rightarrow \mathsf{X}\,\text{Out})) \wedge (\mathsf{G}(\text{Out} \rightarrow \mathsf{X}\,\text{In})))$. Alors $\alpha' \in (\text{In} \cdot \text{Out})^\omega$ et, par définition de f', $a'_1 = s_0^{V_\text{O}}$ et, pour tout $i \geq 1$, $a'_{2i+1} = f(\pi_\text{In}(a'_2 \cdots a'_{2i}))$. On construit $\sigma = s_0 s_1 \cdots \in (S^V)^\omega$ défini par $s_i^{V_\text{I}} = a'_{2i}$ et $s_i^{V_\text{O}} = a'_{2i+1}$, pour tout $i \geq 0$.

Par construction, σ est une exécution f-compatible. Comme f est gagnante pour $(\mathcal{A}, \varphi)$, alors $\sigma \models \varphi$ et, par la remarque 4.32, on a $\alpha' \models \overline{\varphi}$. Donc, pour tout $\alpha' \in \Gamma^\infty$, exécution f'-compatible et f'-équitable, on sait que α' satisfait φ' et on en conclut que f' est une stratégie gagnante pour $(\mathcal{A}', \varphi')$.

Réciproquement, soit $f' : \Gamma^* \rightarrow 2^\Gamma$ une stratégie gagnante pour $(\mathcal{A}', \varphi')$. Alors pour tout $\alpha' \in \Gamma^*$ exécution f'-compatible, $f'(\alpha') \cap \text{Out} \neq \emptyset$ (sinon α' est une exécution f'-maximale, et $\alpha' \not\models \mathsf{G}\,\mathsf{X}\top$, ce qui est impossible car f' est une stratégie gagnante). On définit alors $f : (S^{V_\text{I}})^+ \rightarrow S^{V_\text{O}}$. Pour tout $\rho = r_1 \cdots r_n \in (S^{V_\text{I}})^+$, on pose $\rho' = r'_0 r'_1 \cdots r'_{2n} \in \Gamma^+$, où $r'_0 = s_0^{V_\text{I}}$, $r'_1 = s_0^{V_\text{O}}$, et, pour tout $i > 0$, $r'_{2i} = r_i$ et $r'_{2i+1} = s$ avec $\{s\} = f'(r'_0 r'_1 \cdots r'_{2i}) \cap \text{Out}$. Alors on définit, pour tout $\rho \in (S^{V_\text{I}})^+$,

$$f(\rho) = s \text{ avec } f'(\rho') \cap \text{Out} = \{s\}$$

Soit $\sigma = s_0 s_1 \cdots \in (S^V)^\omega$ une exécution f-compatible. On définit $\alpha' = a'_0 a'_1 \cdots \in (\Gamma)^\omega$ par $a'_{2i} = (s_i)^{V_\text{I}}$ et $a'_{2i+1} = (s_i)^{V_\text{O}}$, pour tout $i \geq 0$. On remarque que si $\rho = (s_1 \cdots s_i)^{V_\text{I}}$ alors $\rho' = a'_0 a'_1 \cdots a'_{2i}$. Par définition de f et d'une exécution synchrone, pour tout $i \geq 1$, $a'_{2i+1} = s_i^{V_\text{O}} = f((s_1 \cdots s_i)^{V_\text{I}}) \in f'(a'_0 \cdots a'_{2i}) \cap \text{Out}$, donc α' est une exécution f'-compatible. Par construction, le nombre d'actions contrôlables de α' est infini, donc cette exécution est trivialement f'-équitable. Comme f' est une stratégie gagnante, on en conclut que $\alpha' \models (s_0^{V_\text{I}} \wedge (\mathsf{G}(\text{In} \rightarrow \mathsf{X}\,\text{Out})) \wedge (\mathsf{G}(\text{Out} \rightarrow \mathsf{X}\,\text{In}))) \rightarrow (\mathsf{X}\, s_0^{V_\text{O}} \wedge \overline{\varphi})$. Par construction, $\alpha' \models s_0^{V_\text{I}} \wedge (\mathsf{G}(\text{In} \rightarrow \mathsf{X}\,\text{Out})) \wedge (\mathsf{G}(\text{Out} \rightarrow \mathsf{X}\,\text{In}))$, donc $\alpha' \models \overline{\varphi}$, et par la remarque 4.32, $\sigma \models \varphi$. Donc f est bien une stratégie gagnante pour $(\mathcal{A}, \varphi)$. □

Remarque 4.33. Afin d'obtenir la décidabilité du problème de SSD asynchrone équitable pour les architectures singleton et les spécifications ω-régulières, on aurait également pu adapter la démonstration de [Var95] à notre modèle. On rappelle que [Var95] propose une démonstration de la décidabilité du problème de synthèse équitable de systèmes centralisés asynchrones à communication par variables partagées basée sur des constructions d'automates (voir théorème 2.50 page 41). Ce problème est également 2EXPTIME-complet lorsque la spécification

est donnée par un automate de Büchi. Par ailleurs, [Var95] fait remarquer que bien qu'on pourrait naïvement penser qu'une spécification LTL entraînerait une complexité dans 3EXPTIME par cette méthode (la transformation d'une formule LTL en un automate de Büchi ayant un coût exponentiel), on peut en fait obtenir une complexité doublement exponentielle même en partant d'une formule LTL, en construisant judicieusement les automates de la démonstration.

2.2 Les architectures fortement connexes

Dans cette section on étend la décidabilité du problème de SSD asynchrone équitable à toute la sous-classe des structures fortement connexes. La démonstration se fait par réduction à la synthèse du singleton. Formellement, on va montrer le résultat suivant :

Théorème 4.34. *Le problème de SSD asynchrone équitable est décidable pour les structures fortement connexes avec des spécifications* AlocTL.

Soient $\mathcal{S} = (\text{Proc}, R, (\text{In}_p)_{p\in\text{Proc}}, (\text{Out}_p)_{p\in\text{Proc}})$ une structure fortement connexe et $\varphi \in \mathsf{AlocTL}(\text{Proc}, \Gamma)$ la spécification.

On définit une structure singleton $\overline{\mathcal{S}} = (\{p\}, \overline{R}, \overline{\text{In}_p}, \overline{\text{Out}_p})$ formée des signaux d'entrée $\overline{\text{In}_p} = \text{In}$ et $\overline{\text{Out}_p} = \text{Out}$. On montre que le problème de SSD asynchrone équitable de la définition 4.13 (donc avec équité locale forte) pour $(\mathcal{S}, \varphi)$ se réduit au problème de SSD asynchrone équitable pour $(\overline{\mathcal{S}}, \varphi)$, *et la partition des actions contrôlables* $\mathcal{P} = \{\text{Out}_p \mid p \in \text{Proc}\}$, problème dont on sait par le théorème 4.22 qu'il est décidable. On rappelle qu'une exécution f-équitable pour l'équité locale forte du singleton $\overline{\mathcal{S}}$ est – en adaptant la définition aux mots – un mot $\alpha \in \Gamma^\infty$ tel que, s'il existe un préfixe $\alpha' \sqsubseteq \alpha$ tel que pour tout préfixe $\alpha'' \in \Gamma^*$ tel que $\alpha' \sqsubseteq \alpha'' \sqsubseteq \alpha$ (avec $\sqsubseteq$ l'ordre préfixe sur les mots), on a $f(\alpha'') \cap \text{Out} \neq \emptyset$, alors $\pi_{\text{Out}}(\alpha) \in \Gamma^\omega$. Or, on va se réduire au problème de synthèse équitable dans lequel une exécution f-équitable du singleton est telle que, *pour tout* $p \in \text{Proc}$, s'il existe un préfixe $\alpha' \in \Gamma^*$ tel que pour tout préfixe $\alpha'' \in \Gamma^*$ vérifiant $\alpha' \sqsubseteq \alpha'' \sqsubseteq \alpha$, $f(\alpha'') \cap \text{Out}_p \neq \emptyset$, alors $\pi_{\text{Out}_p}(\alpha) \in \Gamma^\omega$ (voir la définition 4.7).

Comme toute formule $\varphi \in \mathsf{AlocTL}$ est close par extension d'ordre, si φ est satisfaisable, elle est satisfaisable par des ordres totaux. On remarque que la formule φ dépend de l'ensemble de processus Proc de la structure $\mathcal{S}$. Lorsqu'elle est interprétée sur la structure singleton, les modalités indexées par Proc n'ont a priori pas de sens. Cependant, comme on garde la partition de l'alphabet Γ en fonction des processus de Proc, la sémantique donnée page 113 s'interprète parfaitement sur les ordres totaux étiquetés par Γ qui sont les exécutions du singleton. De plus, il est clair que cette logique est un fragment de FO, les spécifications données sont donc régulières.

On démontre à présent la réduction proprement dite. Tout d'abord on montre comment on peut simuler une stratégie distribuée sur une structure singleton. Afin d'y parvenir, le processus de la structure singleton doit intercaler entre les signaux externes qu'il émet et reçoit les signaux de communication éventuellement échangés entre les processus de la structure distribuée, en respectant la condition d'équité. Ainsi, il peut reconstruire une exécution équitable respectant la stratégie distribuée. On établit donc la proposition suivante :

Proposition 4.35. *S'il existe un ensemble d'alphabets de communication et une stratégie distribuée gagnante pour* $(\mathcal{S}, \varphi)$*, alors il existe une stratégie gagnante pour* $(\overline{\mathcal{S}}, \varphi)$ *et la partition* $\mathcal{P} = \{\text{Out}_p \mid p \in \text{Proc}\}$ *des actions externes de* $\overline{\mathcal{S}}$.

Démonstration. Comme dans la démonstration du théorème 4.22, on remarque que l'alphabet de dépendance $(\Gamma, \overline{D})$ associé à la structure singleton $\overline{\mathcal{S}}$ défini par $a \ \overline{D} \ b$ pour tout $a, b \in \Gamma$ (voir la relation (2.1) page 20) n'induit que des exécutions qui sont des ordres totaux, et on va représenter les exécutions de $\overline{\mathcal{S}}$ par des *mots* de Γ^∞.

Pour une stratégie $f : \Gamma^* \to 2^\Gamma$, une exécution f-compatible du singleton est donc un mot $\sigma = s_0 s_1 \cdots \in \Gamma^\infty$ tel que, pour tout $0 \leq i < |\sigma|$, $s_i \in f(\sigma[i])$ (on rappelle que $\sigma[0] = \varepsilon$ et que pour tout $i \geq |\sigma|$, $\sigma[i] = \sigma[|\sigma|] = \sigma$).

On rappelle par ailleurs qu'on peut associer à tout mot $\sigma \in \Sigma^\infty$ la trace de Mazurkiewicz correspondante $[\sigma] \in \mathbb{R}(\Sigma, D)$.

Soient $(\Sigma^{p,q})_{(p,q)\in R}$ les alphabets de communications internes utilisés induisant l'architecture distribuée $\mathcal{A} = (\text{Proc}, \Sigma, E)$, et $F = (f^p)_{p\in\text{Proc}}$ une stratégie distribuée gagnante pour $(\mathcal{A}, \varphi)$. Par la suite on considérera que les processus de Proc sont tous ordonnés par un ordre $\leq$ quelconque. Afin de définir une stratégie pour le singleton $\overline{\mathcal{S}}$, on doit tout d'abord transformer une histoire sur Γ^* (l'histoire disponible pour le singleton) en une histoire incluant les communications ayant éventuellement eu lieu entre les processus, de telle sorte que le singleton soit à même de simuler le comportement des différents processus de $\mathcal{S}$. De plus le singleton doit simuler un ordonnancement équitable des actions internes des différents processus. Pour cela on va maintenir une file de priorités. On va donc définir $\Phi : \Gamma^* \to \Sigma^*$ une fonction qui enrichit l'histoire sur le singleton en une histoire sur l'alphabet complet, l'application $\text{rg}(f^p) : \Sigma^* \to \mathbb{N}$ pour tout $p \in \text{Proc}$, qui représente la priorité associée au processus p après une histoire sur Σ^*, et enfin la stratégie du singleton $f : \Gamma^* \to 2^\Gamma$.

Pour tout $p \in \text{Proc}$, pour tout $\sigma \in \Sigma^*$ et pour tout $s \in \Sigma$, on définit donc

$$\text{rg}(f^p)(\varepsilon) = 0$$

$$\text{rg}(f^p)(\sigma s) = \begin{cases} \text{rg}(f^p)(\sigma) + 1 & \text{si } s \neq f^p([\sigma]) \text{ et } f^p([\sigma s]) \text{ est définie} \\ 1 & \text{si } s = f^p([\sigma]) \text{ et } f^p([\sigma s]) \text{ est définie} \\ 0 & \text{sinon.} \end{cases}$$

Le rang d'un processus de Proc dans la file de priorités augmente donc strictement tant que sa stratégie est définie sans qu'il puisse jouer. Par ailleurs, un processus dont la stratégie est définie a toujours un rang strictement supérieur dans la file de priorités au rang d'un processus dont la stratégie n'est pas définie.

Notation 4.36. Pour tout $\sigma \in \Sigma^*$, on définit $\text{proc}(\sigma) = \min\{p \in \text{Proc} \mid rg(f^p)(\sigma) = \max_{q\in\text{Proc}}(rg(f^q)(\sigma))\}$, le processus minimal parmi ceux ayant la plus haute priorité. C'est ce processus que le singleton va simuler après avoir vu l'histoire σ. Avec la définition de rg qu'on a donnée, le singleton n'essaiera donc pas de simuler un processus ne désirant pas jouer s'il existe d'autres processus pour lesquels la stratégie est définie.

La stratégie distribuée F étant déterministe, pour tout $\sigma \in \Sigma^*$, il existe une unique séquence maximale $u = u_1 \cdots \in (\Sigma \setminus \Gamma)^\infty$ vérifiant

$$\text{pour tout } i,\ u_i = f^p([\sigma \cdot u[i-1]]) \in (\Sigma_C^p \setminus \Gamma) \tag{4.2}$$

avec $p = \text{proc}(\sigma \cdot u[i-1])$. La séquence u est finie si à un certain point tous les processus ont une priorité 0, et donc si la stratégie du processus minimal n'est pas définie, ou si le processus en tête de la file de priorités veut envoyer un signal externe.

Pour définir Φ on va utiliser la fonction intermédiaire $\mathrm{Com} : \Sigma^* \to (\Sigma \setminus \Gamma)^*$ qui indique quelles actions de communication insérer dans l'histoire vue sur le singleton : pour $\sigma \in \Sigma^*$,

$$\mathrm{Com}(\sigma) = \begin{cases} u & \text{si } u \in (\Sigma \setminus \Gamma)^* \text{ est la séquence maximale vérifiant l'égalité (4.2) pour } \sigma \\ u[1] & \text{si } u \in (\Sigma \setminus \Gamma)^\omega \text{ est la séquence maximale vérifiant l'égalité (4.2) pour } \sigma \end{cases}$$

La fonction Com insère dans l'histoire du singleton autant de messages de communication entre les processus que possible. Si la séquence maximale u vérifiant (4.2) est finie, toutes les communications sont insérées. Sinon, i.e., si les stratégies des processus sont d'envoyer indéfiniment des signaux de communication interne si l'environnement n'émet aucun signal, alors on en insère uniquement un nombre fini (ici, juste le premier signal).

Après une séquence d'actions de Γ, le singleton décide de quelle action dans Out jouer en insérant dans son passé les actions de communication entre les processus que lui dicte la fonction Com. Puis, en fonction de cette histoire sur Σ^* reconstruite, il propose l'action conseillée par la stratégie du processus en tête de la file de priorités après l'exécution de $\mathbb{M}(\Sigma, D)$ correspondant à l'histoire que le singleton a recalculée : on définit donc l'exécution distribuée correspondant à une histoire finie sur le singleton en posant, pour $r \in \Gamma^*$, $a \in \Gamma$,

$$\begin{aligned} \Phi(\varepsilon) &= \mathrm{Com}(\varepsilon) \\ \Phi(r \cdot a) &= \Phi(r) \cdot a \cdot \mathrm{Com}(\Phi(r) \cdot a) \end{aligned}$$

et la stratégie du singleton

$$\begin{aligned} f(r) \cap \mathrm{In} &= \mathrm{In} \\ f(r) \cap \mathrm{Out} &= \{f^p([\Phi(r)])\} \text{ si } f^p([\Phi(r)]) \in \mathrm{Out} \end{aligned}$$

avec $p = \mathrm{proc}(\Phi(r))$. Si $f^p([\Phi(r)])$ n'est pas définie ou si $f^p([\Phi(r)]) \in \Sigma \setminus \Gamma$, la stratégie du singleton est uniquement $f(r) = \mathrm{In}$.

Soit $r = r_0 r_1 \cdots \in \Gamma^\infty$ une exécution f-compatible et f-équitable de $\overline{\mathcal{S}}$. Comme Φ est croissante, la borne supérieure $\bigsqcup_{r' \sqsubseteq r} \Phi(r')$ sur les préfixes finis de r existe et est bien définie. On pose alors, pour tout $r \in \Gamma^\infty$, $\bar{\Phi}(r) = \bigsqcup_{r' \sqsubseteq r} \Phi(r')$ et

$$\sigma = \begin{cases} \Phi(r) & \text{si } \Phi(r) \in \Sigma^\omega \\ \Phi(r) \cdot u & \text{avec } u \in (\Sigma \setminus \Gamma)^\infty \text{ séquence maximale vérifiant (4.2) pour } \Phi(r) \text{ sinon} \end{cases}$$

Lorsque l'environnement joue infiniment souvent, $r \in \Gamma^\omega$ et donc $\Phi(r) \in \Sigma^\omega$. Si par contre, il existe un moment à partir duquel aucune action n'est une action contrôlée par l'environnement, alors il se peut que le singleton ne propose plus non plus d'action à jouer. En revanche, la séquence u représentant les communications entre les processus de $\mathcal{S}$ peut elle être infinie, i.e., les processus peuvent décider de communiquer continûment si l'environnement n'effectue aucune action. Dans ce cas de figure, $\Phi(r)$ se finit par une partie seulement de ces communications des processus, et $\Phi(r)$ peut alors ne pas être une exécution F-compatible F-maximale. La définition de σ complète donc l'exécution en rajoutant la séquence éventuellement infinie des communications entre les processus. On remarque que $\pi_\Gamma(\sigma) = r$, et on va montrer que $[\sigma] \in \mathbb{R}(\Sigma, D)$ est une exécution F-compatible et F-équitable sur $\mathcal{A}$.

Soit $\alpha = [\sigma] = (X, \sqsubseteq, \lambda) \in \mathbb{R}(\Sigma, D)$ l'exécution sur $\mathcal{A}$ correspondant à $\sigma = s_0 s_1 \cdots \in \Sigma^\infty$, avec $X = \{i \in \mathbb{N} \mid 0 \leq i < |\sigma|\}$. On montre que α est F-compatible. Soient $p \in \mathrm{Proc}$ et

$j \in \lambda^{-1}(\Sigma_C^p)$. Alors, par définition de α, $\lambda(j) = s_j$ et, puisque r est f-compatible et par définition de σ, $s_j = f^p([\sigma[j]])$. Pour conclure, on doit montrer que $\pi_{\Sigma^p}([\sigma[j]]) = \pi_{\Sigma^p}(\alpha_{\Downarrow_\alpha j})$. En effet, si tel est le cas, la stratégie f^p étant à mémoire locale, on a $f^p([[\sigma[j]]) = f^p(\alpha_{\Downarrow_\alpha j})$, donc $\lambda(j) = f^p(\alpha_{\Downarrow_\alpha j})$, ce qui correspond à la définition d'une exécution F-compatible.

On note

$$\pi_{\Sigma^p}([\sigma[j]]) = (X_j, \sqsubseteq_j, \lambda) \in \mathbb{M}(\Sigma, D)$$

avec $X_j = \{i \in \mathbb{N} \mid 0 \leq i < j\} \cap \lambda^{-1}(\Sigma^p)$, et $\sqsubseteq_j = \sqsubseteq \cap (X_j \times X_j)$ et

$$\pi_{\Sigma^p}(\alpha_{\Downarrow_\alpha j}) = (X'_j, \sqsubseteq'_j, \lambda) \in \mathbb{M}(\Sigma, D)$$

avec $X'_j = \{i \in \mathbb{N} \mid 0 \sqsubseteq i \sqsubset j\} \cap \lambda^{-1}(\Sigma^p)$ et $\sqsubseteq'_j = \sqsubseteq \cap (X'_j \times X'_j)$.

Remarque 4.37. $< \cap (\lambda^{-1}(\Sigma^p))^2 = \sqsubseteq \cap (\lambda^{-1}(\Sigma^p))^2$ est un ordre total.

Par définition de la relation $\sqsubseteq$, $X'_j \subseteq X_j$. Soit $i \in X_j$. Alors $i < j$ et $i \in \lambda^{-1}(\Sigma^p)$. Or, $j \in \lambda^{-1}(\Sigma^p)$ donc $\lambda(i)\ D\ \lambda(j)$ et par définition de α, $i \sqsubset j$. Ainsi, $X_j = X'_j$, et $\sqsubseteq_j = \sqsubseteq'_j$. Donc $\pi_{\Sigma^p}([\sigma[j]]) = \pi_{\Sigma^p}(\alpha_{\Downarrow_\alpha j})$, et α est une exécution F-compatible.

Supposons à présent que α n'est pas F-équitable. Pour simplifier les explications et éviter des notations fastidieuses sur les indices, on sépare les cas où σ est fini de ceux où σ est infini. Supposons tout d'abord que $\Phi(r) = \sigma$ est un mot fini. Alors si α n'est pas F-équitable, il existe un processus $p \in \text{Proc}$ tel que $f^p([\sigma])$ est défini (voir la remarque 4.8). Par définition, cela signifie que $\text{rg}(f^p)(\sigma) \geq 1$. Si $p \neq \text{proc}(\sigma)$ alors il existe $p' = \text{proc}(\sigma)$ tel que $\text{rg}(f^{p'})(\sigma) \geq 1$, et donc tel que $f^{p'}([\sigma])$ est défini. Si $f^{p'}([\sigma]) \in \Sigma \setminus \Gamma$, alors u n'est pas maximal, ce qui contredit la définition de σ, donc nécessairement, $f^{p'}([\sigma]) \in \text{Out}$ et par définition de f, on a $f(r) \cap \text{Out} = \{f^{p'}([\Phi(r)])\} = \{f^{p'}([\sigma])\}$, ce qui implique que r n'est pas f-maximal, donc pas f-équitable.

Si $\sigma \in \Sigma^\omega$, on commence par démontrer certaines caractéristiques sur la gestion de la file de priorités. Tout d'abord, on montre que s'il existe un processus continuellement activable, mais dont les actions ne sont insérées qu'un nombre fini de fois dans σ, alors il existe un (éventuellement autre) processus continuellement en tête de la file de priorités dont les actions ne sont aussi insérées qu'un nombre fini de fois.

Lemme 4.38. *Supposons que $\sigma \in \Sigma^\omega$. S'il existe un processus $p \in \text{Proc}$ et un indice $i \geq 0$, tel que pour tout $k \geq i$, $rg(f^p)(\sigma[k+1]) > rg(f^p)(\sigma[k])$ alors il existe un processus $p' \in \text{Proc}$ et un indice $j \geq 0$, tel que pour tout $k \geq j$, $rg(f^{p'})(\sigma[k+1]) > rg(f^{p'})(\sigma[k])$ et $p' = proc(\sigma[k])$.*

Démonstration du lemme 4.38. S'il existe $j \geq i$ tel que $p = \text{proc}(\sigma[j])$ alors pour tout $k \geq j$, il est clair que $p = \text{proc}(\sigma[k])$. Si par contre $p \neq \text{proc}(\sigma[j])$ pour tout $j \geq i$, alors cela signifie que pour tout $j \geq i$, il existe $p' \in \text{Proc}$ tel que $\text{rg}(f^{p'})(\sigma[j]) \geq \text{rg}(f^p)(\sigma[j])$. Or, s'il existe un indice $k \geq i$ tel que $\text{rg}(f^{p'})(\sigma[k+1]) \leq \text{rg}(f^{p'})(\sigma[k])$ alors pour tout $j \geq k$, $\text{rg}(f^p)(\sigma[j]) > \text{rg}(f^{p'})(\sigma[j])$. Cela implique, le nombre de processus étant fini, qu'il existe un indice $j \geq 0$ et un processus $p' \in \text{Proc}$, tel que $p' = \text{proc}(\sigma[j])$, et, pour tout $k \geq j$, $\text{rg}(f^{p'})(\sigma[k+1]) > \text{rg}(f^{p'})(\sigma[k])$. Donc, pour tout $k \geq j$, $p' = \text{proc}(\sigma[k])$. □

On montre à présent qu'un processus restant continuellement en tête de la file de priorité sans qu'une action qu'il propose ne soit ajoutée dans σ (c'est-à-dire, tel que la fonction rg augmente continuellement) ne propose nécessairement que des signaux externes :

Lemme 4.39. *Supposons que $\sigma \in \Sigma^\omega$. S'il existe un processus $p \in$ Proc et un indice $i \geq 0$, tels que pour tout $k \geq i$, $rg(f^p)(\sigma[k+1]) > rg(f^p)(\sigma[k])$, et $p = proc(\sigma[k])$, alors pour tout $n > i$, $f^p([\sigma[n]]) \in$ Out.*

Démonstration du lemme 4.39. Supposons qu'il existe $m > i$ tel que $f^p([\sigma[m]]) \in \Sigma\backslash\Gamma$ (pour tout $n > i$, $f^p([\sigma[n]])$ est définie, sinon $\mathrm{rg}(f^p)(\sigma[n]) = 0$). Soit $r_m = \pi_\Gamma(\sigma[m])$. Si $r_m = r$, alors $\sigma = \sigma[m'] \cdot u$, avec $m' \leq m$, $\sigma[m'] \in \{\varepsilon\} \cup (\Sigma^* \cdot \Gamma)$, et $u \in (\Sigma \setminus \Gamma)^\omega$. Sinon, $\sigma = \Phi(r_m) \cdot \sigma' = \sigma[m'] \cdot \mathrm{Com}(\sigma[m']) \cdot \sigma'$, avec $m' \leq m$, $\sigma[m'] \in \{\varepsilon\} \cup (\Sigma^* \cdot \Gamma)$, $\mathrm{Com}(\sigma[m']) \in (\Sigma \backslash \Gamma)^*$, et $\sigma' \in \Gamma \cdot \Sigma^\omega$.

Le seul cas où une action de communication définie pour un processus arrivant en tête de la file de priorités n'est pas ajouté dans Com, est le cas où la séquence d'actions de communication des processus est infinie et où on n'insère dans r que la première de ces actions. On montre formellement qu'ici, l'action de communication de f^p est nécessairement jouée : si $\sigma[m] = \sigma[m']$, alors $\sigma = \sigma[m'] \cdot f^p([\sigma[m]]) \cdot u'$ avec $u' \in \Sigma^\omega$, et $\mathrm{rg}(f^p)(\sigma[m+1]) \leq 1 \leq \mathrm{rg}(f^p)(\sigma[m])$, ce qui est en contradiction avec l'hypothèse. Si $\sigma[m] = \sigma[m'] \cdot u_1$, et $u_1 \in \Sigma \setminus \Gamma$, alors $\mathrm{proc}(\sigma[m']) = \mathrm{proc}(\sigma[m-1]) = p$ car $m - 1 \geq i$. Donc, par définition, $u_1 = f^p([\sigma[m']])$, et $\mathrm{rg}(f^p)(\sigma[m]) \leq \mathrm{rg}(f^p(\sigma[m-1])$, ce qui est en contradiction avec l'hypothèse. Si enfin, $\sigma[m] = \sigma[m'] \cdot u$ avec $u \in (\Sigma \setminus \Gamma)^+$, et $|u| > 1$, alors immédiatement, $\sigma = \sigma[m] \cdot b \cdot u' \cdot \sigma'$, avec $u' \in \Sigma \setminus \Gamma^+$, $b = f^p([\sigma[m]])$, et $\sigma' \in \Sigma^\omega$, ce qui est à nouveau en contradiction avec l'hypothèse. Donc, pour tout $n > i$, on a $f^p([\sigma[n]]) \in$ Out. □

Enfin, on affirme que s'il existe un moment à partir duquel un processus reste continuellement en tête de la file de priorités, en ne proposant que des actions externes, alors la stratégie du singleton va être de proposer continuellement les actions de ce processus.

Lemme 4.40. *Supposons que $\sigma \in \Sigma^\omega$. S'il existe $p \in$ Proc un processus et $i \geq 0$ un indice tels que, pour tout $n \geq i$, $p = proc(\sigma[n])$ et $f^p([\sigma[n]]) \in$ Out, alors $r \in \Gamma^\omega$, et pour tout préfixe r' de r tel que $\pi_\Gamma(\sigma[i]) \sqsubseteq r' \sqsubseteq r$, on a $f(r') \cap \mathrm{Out} = \{f^p([\Phi(r')])\}$.*

Démonstration du lemme 4.40. Supposons que $r \in \Gamma^*$. Alors $\sigma = \Phi(r) \cdot u$, $u \in (\Sigma \backslash \Gamma)^\omega$. Soit $n \geq \max(i, |\Phi(r)|)$, alors $p = \mathrm{proc}(\sigma[n])$, et $f^p([\sigma[n]]) \in$ Out, et $\sigma = \sigma[n] \cdot u'$, avec $u' \in (\Sigma \setminus \Gamma)^\omega$, ce qui est en contradiction avec la définition de σ.

Donc $r \in \Gamma^\omega$, et soit r' un préfixe de r tel que $\pi_\Gamma(\sigma[i]) \sqsubseteq r'$. Puisque $r \in \Gamma^\omega$, $\sigma[i] \sqsubseteq \Phi(\pi_\Gamma(\sigma[i]))$, et comme la fonction Φ est croissante, on a $\sigma[i] \sqsubseteq \Phi(r')$. Donc, par hypothèse, $p = \mathrm{proc}(\Phi(r'))$, et $f^p([\Phi(r')]) \in$ Out. Donc, par définition de f, on en déduit que $f(r') \cap \mathrm{Out} = f^p([\Phi(r')])$. □

À l'aide de ces trois lemmes, on montre que α est une exécution F-équitable si r est f-équitable. En effet, supposons que $\sigma = s_0 s_1 \cdots \in \Sigma^\omega$ n'est pas équitable. Alors il existe un processus $p \in$ Proc et un indice $i \geq 0$ tel que pour tout $k \geq i$, $f^p([\sigma[k]])$ est définie et $s_k \notin \Sigma_C^p$. Alors, pour tout $k \geq i$, $\mathrm{rg}(f^p)(\sigma[k+1]) > \mathrm{rg}(f^p)(\sigma[k])$. Par le lemme 4.38, il existe un processus $p' \in$ Proc et un indice $j \geq 0$ tel que, pour tout $k \geq j$, $\mathrm{rg}(f^{p'})(\sigma[k+1]) > \mathrm{rg}(f^{p'})(\sigma[k])$ et $p' = \mathrm{proc}(\sigma[k])$. Dans ce cas-là, le lemme 4.39 assure que pour tout $n > j$, $f^{p'}([\sigma[n]]) \in$ Out. Enfin le lemme 4.40 permet de conclure que $r \in \Gamma^\omega$, et que pour tout préfixe r' de r tel que $\pi_\Gamma(\sigma[j+1]) \sqsubseteq r' \sqsubseteq r$, on a $f(r') \cap \mathrm{Out}_{p'} \neq \emptyset$. Puisque pour tout $k \geq j$, $\mathrm{rg}(f^{p'})(\sigma[k+1]) > \mathrm{rg}(f^{p'})(\sigma[k])$, alors pour tout $k \geq j$, $s_k \notin \mathrm{Out}_{p'}$, ce qui implique que r n'est pas f-équitable.

Donc, pour toute exécution $r \in \Gamma^\infty$ qui soit f-compatible et f-équitable pour la partition $\{\mathrm{Out}_p \mid p \in \mathrm{Proc}\}$, on peut construire $\alpha = [\sigma] \in \mathbb{R}(\Sigma, D)$, exécution F-compatible et F-équitable dont r est une linéarisation de la partie observable $\pi_\Gamma(\alpha)$. Comme F est une stratégie

gagnante, $\pi_\Gamma(\alpha) \models \varphi$. La formule φ est close par extension, donc $r \models \varphi$. Donc f est bien une stratégie gagnante pour $(\overline{\mathcal{S}}, \varphi)$. □

On montre à présent que toute stratégie gagnante sur la structure singleton peut être distribuée sur une architecture fortement connexe.

Proposition 4.41. *S'il existe une stratégie gagnante pour $(\overline{\mathcal{S}}, \varphi)$ et équitable pour la partition des actions externes $\mathcal{P} = \{\mathrm{Out}_p \mid p \in \mathrm{Proc}\}$, alors il existe un ensemble d'alphabets de communication et une stratégie distribuée gagnante pour $(\mathcal{S}, \varphi)$. De plus, s'il existe une stratégie à mémoire finie pour le singleton alors on peut construire des ensembles d'alphabets de communication finis et une stratégie distribuée à mémoire finie pour $\mathcal{S}$.*

Démonstration. Pour obtenir ce résultat, on veut simuler une exécution *totalement ordonnée* compatible avec la stratégie du singleton sur le système distribué donné par la structure $\mathcal{S}$. Comme les actions de l'environnement sont incontrôlables, il est impossible d'obtenir des exécutions totalement ordonnées sur l'architecture distribuée, cependant le but de la construction est de limiter autant que possible les événements concurrents, de façon à obtenir des exécutions qui soient des *affaiblissements* des exécutions du singleton. Afin d'y parvenir, les processus de $\mathcal{S}$ vont simuler un passage de jeton. On va sélectionner un cycle dans le graphe de communication, et forcer les processus à jouer de façon séquentielle dans cet anneau virtuel – on remarque que le cycle peut ne pas être un cycle simple.

Soit $f : \Gamma^* \to 2^\Gamma$ une stratégie gagnante pour $(\overline{\mathcal{S}}, \varphi)$. On suppose que f est donnée par un *automate déterministe avec sortie* – un automate dont toutes les exécutions sont acceptantes, et auquel on ajoute une fonction associée aux états de l'automate. On dit que la stratégie f est *à mémoire finie* si l'automate qui la calcule est fini, i.e. son ensemble d'états est fini. Soit $\mathfrak{A}^f = (Q^f, \Gamma, \delta^f, s_0^f, \overline{f})$ l'automate calculant f, avec

$$\begin{aligned} Q^f &\text{ l'ensemble (fini si la stratégie est à mémoire finie) d'états} \\ \delta^f : Q^f \times \Gamma \to Q^f &\text{ la fonction de transition} \\ s_0^f &\text{ l'état initial} \\ \overline{f} : Q^f \to \mathrm{Out} &\text{ la stratégie proprement dite.} \end{aligned}$$

On remarque que l'automate $\mathfrak{A}^f$ ne calcule en fait que $f \cap \mathrm{Out}$. On va définir, pour chaque processus $p \in \mathrm{Proc}$ un automate avec sortie $\mathfrak{A}^p = (Q^p, \Sigma^p, \delta^p, s_0^p, \overline{f}^p)$ calculant sa stratégie f^p. Pour cela, on choisit un cycle de taille n dans l'architecture. On utilise les fonctions auxiliaires ring et succ_p définies par

$$\mathrm{ring} : \{1, \ldots, n\} \to \mathrm{Proc}$$

L'application ring est une application surjective qui associe à chaque élément de l'anneau que l'on cherche à simuler un processus de Proc. Elle vérifie, pour tout $1 \leq i < n$, $(\mathrm{ring}(i), \mathrm{ring}(i+1)) \in R$ et $(\mathrm{ring}(n), \mathrm{ring}(1)) \in R$. Comme on l'a déjà relevé, ring n'est pas nécessairement bijective et il se peut qu'un processus ait plusieurs antécédents.

$$\begin{aligned} \mathrm{succ}_p &: \{1, \ldots, n\} \to \{1, \ldots, n\} \\ \mathrm{succ}_p(i) &= \begin{cases} \min\{j \in ring^{-1}(p) \mid j > i\} & \text{si } \{j \in ring^{-1}(p) \mid j > i\} \neq \emptyset \\ \min\{j \in ring^{-1}(p)\} & \text{sinon.} \end{cases} \end{aligned}$$

L'application succ_p associe à un antécédent du processus p le prochain élément de l'anneau que p aura à simuler. Par exemple si le processus p a la place 2 et 5 dans le cycle, $\mathrm{succ}_p(2) = 5$ et $\mathrm{succ}_p(5) = 2$.

Au cours de l'exécution, les processus vont reconstruire une exécution totalement ordonnée, linéarisation de l'exécution observable qu'ils sont en train de jouer, et respectant la stratégie du singleton. En se passant le jeton, les processus vont donc également se transmettre l'histoire courante sur le singleton – l'état courant de l'automate $\mathfrak{A}^f$. Le processus qui reçoit le jeton met à jour l'histoire courante, en ajoutant à la fin la séquence des événements locaux qu'il a observés depuis le dernier passage de jeton (dans ce cas, les seuls signaux de communication sont les passages de jetons). Lorsque la stratégie du singleton est à mémoire finie, les processus n'ont pas à mémoriser toute l'histoire des événements qu'ils ont observé localement (ce qui nécessiterait une mémoire non bornée), mais uniquement la fonction de transition partielle de $\mathfrak{A}^f$, donnant un état de Q^f d'arrivée, en fonction d'un état de départ, lorsque les événements ayant eu lieu sont ceux observés par le processus.

Pour $p \in \mathrm{Proc}$, les états de $\mathfrak{A}^p$ sont donc donnés par

$$Q^p = \Big((Q^f)^{Q^f} \times \bigcup_{i \in \mathrm{ring}^{-1}(p)} \{\mathrm{NToken}_i\}\Big) \cup \big(Q^f \times \bigcup_{i \in \mathrm{ring}^{-1}(p)} \{\mathrm{Token}_i, \mathrm{Token}'_i\}\big)$$

où Token_i, Token'_i et NToken_i sont des drapeaux indiquant si le processus courant possède le jeton en simulant le i-ème élément de l'anneau, auquel cas son état courant est l'état courant de $\mathfrak{A}^f$, ou bien s'il ne possède pas le jeton, et dans ce cas, il mémorise les événements qu'il observe sous forme d'une fonction de transition partielle de $\mathfrak{A}^f$. Par la suite on notera $\mathrm{Token}_p = \bigcup_{i \in \mathrm{ring}^{-1}(p)} \{\mathrm{Token}_i, \mathrm{Token}'_i\}$ et $\mathrm{Token} = \bigcup_{p \in \mathrm{Proc}} (Q^f \times \bigcup_{i \in \mathrm{ring}^{-1}(p)} \{\mathrm{Token}_i, \mathrm{Token}'_i\})$.

Lorsqu'un processus $p \in \mathrm{Proc}$ ne possède pas le jeton, l'automate $\mathfrak{A}^p$ mémorise dans son état courant l'histoire des événements externes visibles au processus p. Comme le seul processus habilité à émettre des signaux est celui ayant le jeton, les seuls événements externes pouvant avoir lieu sont des signaux émis par l'environnement. Ainsi, pour tout $p \in \mathrm{Proc}$ et tout $i \in \mathrm{ring}^{-1}(p)$, pour tout $\delta_\sigma \in (Q^f)^{Q^f}$ permettant de calculer la mémoire nécessaire à $\mathfrak{A}^f$ après avoir vu $\sigma \in \mathrm{In}_p^*$, et pour tout $a \in \mathrm{In}_p$, on pose

$$\delta^p((\delta_\sigma, \mathrm{NToken}_i), a) = (\delta_{\sigma \cdot a}, \mathrm{NToken}_i)$$

avec $\delta_{\sigma \cdot a}(s) = \delta^f(\delta_\sigma(s), a)$.

Lorsqu'un processus $p \in \mathrm{Proc}$ possède le jeton, il se base sur la stratégie f du singleton pour décider quels signaux émettre. Tant que la stratégie f du singleton propose de jouer une action de Out_p, sans pouvoir la jouer, la stratégie du processus p va être de proposer cette action. Si la stratégie du singleton change, i.e., propose une action contrôlable par un autre processus, ou n'est plus définie, alors le processus p va essayer de passer le jeton au processus suivant. De même, une fois que p a pu effectuer une action externe, sa stratégie est de passer le jeton. Ceci assure que le jeton n'est pas monopolisé par un seul processus, empêchant les autres de jouer. Pour modéliser cette différence entre la volonté de jouer une action externe, et la volonté de passer le jeton, le processus p peut être dans des états ayant deux drapeaux différents : Token_i et Token'_i. Formellement, pour tout $p \in \mathrm{Proc}$, $i \in \mathrm{ring}^{-1}(p)$, pour tout $s \in Q^f$, pour tout $a \in \mathrm{In}_p \cup \mathrm{Out}_p$,

$$\delta^p((s, \mathrm{Token}_i), a) = \begin{cases} (\delta^f(s,a), \mathrm{Token}_i) & \text{si } a \in \mathrm{In}_p \text{ et } \overline{f}(\delta^f(s,a)) \in \mathrm{Out}_p \\ (\delta^f(s,a), \mathrm{Token}'_i) & \text{sinon.} \end{cases} \tag{4.3}$$

Le premier cas représente les fois où le processus n'a pas eu l'occasion de jouer, et pour lesquels la stratégie du singleton est toujours de jouer dans la partition Out_p. Le second cas représente les fois où le processus a pu jouer, ou bien la stratégie du singleton n'est plus définie dans la partition Out_p, et donc p va passer le jeton.

À partir du moment où un processus p veut envoyer le jeton, il ne peut plus changer d'avis : il continue à mettre à jour l'histoire du singleton, en attendant de pouvoir effectuer son action : formellement, pour tout $p \in \mathrm{Proc}$, tout $i \in \mathrm{ring}^{-1}(p)$, pour tout tout $s \in Q^f$,

$$\delta^p((s, \mathrm{Token}'_i), a) = (\delta^f(s, a), \mathrm{Token}'_i) \text{ si } a \in \mathrm{In}_p.$$

Pour modéliser le passage de jeton, un processus envoie l'état courant de l'automate $\mathfrak{A}^f$ au processus suivant dans le cycle. Pour expliciter le processus destinataire du signal, et obtenir des ensembles d'alphabets de communication distincts deux à deux, on rajoute dans le signal le numéro du processus dans le cycle. Formellement, on définit la fonction de sortie de $\mathfrak{A}^p$ de la façon suivante : pour tout $s \in Q^f$, tout $i \in \mathrm{ring}^{-1}(p)$,

$$\overline{f^p}(s, \mathrm{Token}_i) = \overline{f}(s)$$
$$\overline{f^p}(s, \mathrm{Token}'_i) = (s, i)$$

Lorsque le processus dans l'état Token a émis le signal de transmission du jeton, il réinitialise sa mémoire de son histoire locale, et repasse dans l'état NToken associé au numéro suivant qu'il prendra dans le cycle. Le processus qui reçoit le signal passe lui en état Token et calcule la nouvelle séquence d'événements sur le singleton en ajoutant l'histoire des événemets locaux qu'il a mémorisée. Donc, pour tout $p \in \mathrm{Proc}$, pour tout $i \in \mathrm{ring}^{-1}(p)$, pour tout $s \in Q^f$, on définit :

$$\delta^p((s, \mathrm{Token}'_i), (s, i)) = (id, \mathrm{NToken}_{succ_p(i)})$$

et pour tout $\delta_\sigma \in (Q^f)^{Q^f}$, pour $i \in \{1, \cdots, n\}$ et $p = \mathrm{ring}((i \bmod n) + 1)$,

$$\delta^p((\delta_\sigma, \mathrm{NToken}_{(i \bmod n)+1}), (s, i)) = \begin{cases} (\delta_\sigma(s), \mathrm{Token}_{(i \bmod n)+1}) & \text{si } \overline{f}(\delta_\sigma(s)) \in \mathrm{Out}_p \\ (\delta_\sigma(s), \mathrm{Token}'_{(i \bmod n)+1}) & \text{sinon.} \end{cases} \quad (4.4)$$

Enfin l'état initial de l'automate $\mathfrak{A}^p$ est donné par :

$$s_0^p = \begin{cases} (s_0^f, \mathrm{Token}_1) & \text{si } \mathrm{ring}(1) = p \text{ et } \overline{f}(s_0^f) \in \mathrm{Out}_p \\ (s_0^f, \mathrm{Token}'_1) & \text{si } \mathrm{ring}(1) = p \text{ et } \overline{f}(s_0^f) \notin \mathrm{Out}_p \\ (id, \mathrm{NToken}_1) & \text{sinon} \end{cases}$$

Remarque 4.42. On a l'invariant suivant : si $\mathfrak{A}^p$ est dans un état (s, Token_i), avec $i \in \mathrm{ring}^{-1}(p)$, alors $\overline{f}(s)$ est définie et $\overline{f}(s) \in \mathrm{Out}_p$, et donc $\overline{f^p}(s, \mathrm{Token}_i) \in \mathrm{Out}_p$.

On définit, pour tout $i, j \in \{1, \dots, n\}$,

$$\Sigma^{i,j} = \begin{cases} Q^f \times \{i\} & \text{si } j = i+1 \text{ ou } i = n \text{ et } j = 1 \\ \emptyset & \text{sinon.} \end{cases}$$

Les alphabets de communication sont donc donnés par, pour tout $(p, q) \in R$,

$$\Sigma^{p,q} = \biguplus_{\substack{i \in \text{ring}^{-1}(p) \\ j \in \text{ring}^{-1}(q)}} \Sigma^{i,j}.$$

Ainsi, on a bien, pour tout $p \in \text{Proc}$, $\delta^p : Q^p \times \Sigma^p \to Q^p$.

Notation 4.43. On définit l'application $\overline{s} : \mathbb{M}(\Sigma, D) \to \Pi_{p \in \text{Proc}} Q^p$, qui, à toute trace d'exécution finie $\alpha \in \mathbb{M}(\Sigma, D)$, associe les états atteints par les différents automates $(\mathfrak{A}_p)_{p \in \text{Proc}}$, s'ils ont tous une exécution sur α : pour tout $\alpha \in \mathbb{M}(\Sigma, D)$, pour tout $p \in \text{Proc}$,

$$\overline{s}^p(\alpha) = \begin{cases} \delta^p(s_0^p, \pi_{\Sigma^p}(\alpha)) & \text{si } \mathfrak{A}^p \text{ a une exécution sur } \pi_{\Sigma^p}(\alpha) \\ \text{indéfini} & \text{sinon.} \end{cases}$$

et, pour tout $\alpha \in \mathbb{M}(\Sigma, D)$ tel que pour tout $p \in \text{Proc}$, $\overline{s}^p(\alpha)$ est défini, on pose

$$\overline{s}(\alpha) = (\overline{s}^p(\alpha))_{p \in \text{Proc}}.$$

On rappelle que comme $\pi_{\Sigma^p}(\alpha)$ est un ordre total, $\pi_{\Sigma^p}(\alpha)$ peut être vu comme un mot de Σ^*.

On définit de même, pour tout $\alpha \in \mathbb{M}(\Gamma, \overline{D})$, $\overline{s}^f(\alpha) = \delta^f(s_0^f, \alpha)$.

On pose alors, pour tout $\alpha \in \mathbb{M}(\Sigma, D)$, pour tout $p \in \text{Proc}$,

$$f^p(\alpha) = \overline{f^p}(\overline{s}^p(\alpha))$$

et f^p est bien à mémoire locale.

Avec ces notations, une exécution $\alpha = (X, \leq, \lambda) \in \mathbb{R}(\Sigma, D)$ est F-compatible, si, pour tout $p \in \text{Proc}$, tout $x \in \lambda^{-1}(\Sigma_C^p)$, $\lambda(x) = \overline{f^p}(\overline{s}^p(\alpha_{\Downarrow_\alpha x}))$.

Pour tout $p \in \text{Proc}$, tout $x \in \lambda^{-1}(\Sigma^p)$, si $\overline{s}^p(\alpha_{\downarrow_\alpha x}) \in \text{Token}$, on pourra dire que « p a le jeton en x ».

Comme on le souhaitait, les exécutions respectant la stratégie F présentent un certain nombre de caractéristiques. On remarque en particulier :

Remarque 4.44. Soit $\alpha = (X, \leq, \lambda) \in \mathbb{R}(\Sigma, D)$ une exécution F-compatible et F-équitable. Alors elle a les caractéristiques suivantes :

1. Pour tout α_1 préfixe de α, il existe un unique $p \in \text{Proc}$ tel que $\overline{s}^p(\alpha_1) \in \text{Token}$. De plus, pour tout $x \in X$ tel que $\alpha_1 \cdot \lambda(x)$ est un préfixe de α,
   - si $\lambda(x) \notin \Sigma^p$, $\overline{s}^p(\alpha_1 \cdot \lambda(x)) = \overline{s}^p(\alpha)$,
   - si $\lambda(x) \in \Sigma^p \cap \Gamma$, $\overline{s}^p(\alpha_1 \cdot \lambda(x)) \in \text{Token}$,
   - si $\lambda(x) \in \Sigma^p \setminus \Gamma$, alors $\overline{s}^p(\alpha_1 \cdot \lambda(x)) \notin \text{Token}$ et il existe un unique $p' \in \text{Proc}$ tel que $x \in \lambda^{-1}(\Sigma^{p,p'})$ et $\overline{s}^{p'}(\alpha_1 \cdot \lambda(x)) \in \text{Token}$.
2. Les événements étiquetés par des actions contrôlables sont totalement ordonnés.
3. Pour toute trace $\alpha' = (X', \leq, \lambda)$ préfixe de α, pour tout $p \in \text{Proc}$, il existe un événement $z \in X \setminus X'$ tel qu'il existe $q \in \text{Proc}$, $\lambda(z) \in \Sigma^{q,p}$ (et donc p a le jeton en z).

4. Pour tous $p, q \in \text{Proc}$, tous $x \in \lambda^{-1}(\Sigma^p)$, $y \in \lambda^{-1}(\Sigma^q)$, si $\overline{s}^p(\alpha_{\downarrow_\alpha x}) \in \text{Token}$ et $\overline{s}^q(\alpha_{\downarrow_\alpha y}) \in \text{Token}$, on a $x \leq y$ ou $y \leq x$.

Démonstration. 1. On le démontre par récurrence sur les préfixes de α. Si α_1 est la trace vide, alors $\overline{s}(\alpha_1) = (s_0^p)_{p\in\text{Proc}}$, et le processus ring(1) est le seul à vérifier $\overline{s}^{\text{ring}(1)}(\alpha_1) \in \text{Token}$. Soit α_1 préfixe de α, $p \in \text{Proc}$ tel que $\overline{s}^p(\alpha_1) \in \text{Token}$ et soit $x \in X$ tel que $\alpha_1 \cdot \lambda(x)$ est un préfixe de α. Par définition de F, pour tout $p'' \neq p$, $F(\alpha_1) \cap \Sigma_C^{p''} = \emptyset$. Alors, si $x \notin \lambda^{-1}(\Sigma^p)$, $x \in \lambda^{-1}(\text{In})$ et soit $p' \in \text{Proc}$ tel que $x \in \lambda^{-1}(\Sigma^{p'})$. Par définition de $\delta^{p'}$, on a $\overline{s}^{p'}(\alpha_1 \cdot \lambda(x)) \notin \text{Token}$ et pour tout $p'' \neq p'$, $\overline{s}^{p''}(\alpha_1 \cdot \lambda(x)) = \overline{s}^{p''}(\alpha_1)$, donc p est le seul processus à vérifier $\overline{s}^p(\alpha_1 \cdot \lambda(x)) \in \text{Token}$. Si $x \in \lambda^{-1}(\Sigma^p \cap \Gamma)$, les stratégies étant à mémoire locale, pour tout $p' \neq p$, $\overline{s}^{p'}(\alpha_1 \cdot \lambda(x)) = \overline{s}^{p'}(\alpha_1)$ et par définition de δ^p, on a $\overline{s}^p(\alpha_1 \cdot \lambda(x)) \in \text{Token}$. Si $x \in \lambda^{-1}(\Sigma^p \setminus \Gamma)$, alors il existe un unique $p' \in \text{Proc}$ tel que $x \in \lambda^{-1}(\Sigma^{p,p'})$ et $\overline{s}^p(\alpha_1 \cdot \lambda(x)) \notin \text{Token}$ et $\overline{s}^{p'}(\alpha_1 \cdot \lambda(x)) \in \text{Token}$. Pour tout $p'' \neq p, p'$, on a $\overline{s}^{p''}(\alpha_1 \cdot \lambda(x)) = \overline{s}^{p''}(\alpha_1) \notin \text{Token}$. Donc il existe un unique processus, maintenant p', vérifiant $\overline{s}^{p'}(\alpha_1 \cdot \lambda(x)) \in \text{Token}$.

2. Cette propriété découle immédiatement de 1. En effet, supposons qu'il existe deux événements $z_1, z_2 \in \lambda^{-1}(\Sigma_C)$ concurrents. Alors soient $p_1 \in \text{Proc}$ tel que $z_1 \in \lambda^{-1}(\Sigma^{p_1})$ et $\overline{s}^{p_1}(\alpha_{\Downarrow_\alpha z_1}) \in \text{Token}$ et $p_2 \in \text{Proc}$ tel que $z_2 \in \lambda^{-1}(\Sigma^{p_2})$ et $\overline{s}^{p_2}(\alpha_{\Downarrow_\alpha z_2}) \in \text{Token}$. Soit $\alpha' = (X', \leq', \lambda)$ trace préfixe de α telle que $X' = \Downarrow_\alpha z_1 \cup \Downarrow_\alpha z_2$. Comme z_1 et z_2 sont concurrents, alors pour tout $z \in X' \cap \lambda^{-1}(\Sigma^{p_1})$, $z < z_1$ et donc $\overline{s}^{p_1}(\alpha') = \overline{s}^{p_1}(\alpha_{\Downarrow_\alpha z_1})$. De même, $\overline{s}^{p_2}(\alpha') = \overline{s}^{p_2}(\alpha_{\Downarrow_\alpha z_2})$. On obtient donc $\overline{s}^{p_1}(\alpha') \in \text{Token}$ et $\overline{s}^{p_2}(\alpha') \in \text{Token}$, ce qui est impossible.

3. Soit $\alpha' = (X', \leq, \lambda)$ une trace préfixe de α, et soit $p \in \text{Proc}$ tel que $\overline{s}^p(\alpha') \in \text{Token}$. Supposons que $\{x \in X \setminus X' \mid \lambda(x) \in \Sigma_C^p\}$ est vide. Alors on peut montrer que pour toute trace préfixe α'' telle que $\alpha' \leq \alpha'' \leq \alpha$, $\overline{s}^p(\alpha'') \in \text{Token}$, et donc $f^p(\alpha'')$ est définie. On en déduit que α n'est pas F-équitable, ce qui est en contradiction avec l'hypothèse. Soit donc x_0 élément minimal de l'ensemble $\{x \in X \setminus X' \mid \lambda(x) \in \Sigma_C^p\}$, et on montre qu'alors il existe nécessairement une action de communication dans cet ensemble, i.e., $\{x \in X \setminus X' \mid \lambda(x) \in \Sigma_C^p \setminus \Gamma\} \neq \emptyset$. En effet, si $\lambda(x_0) \in \Sigma_C^p \setminus \Gamma$, alors l'ensemble est immédiatement non vide. Sinon, si $\lambda(x_0) \in \text{Out}_p$, alors par définition de δ^p, $\overline{s}^p(\alpha_{\Downarrow_\alpha x_0}) \in Q^f \times \{\text{Token}_i\}$, et $\overline{s}^p(\alpha_{\downarrow_\alpha x_0}) \in Q^f \times \{\text{Token}_i'\}$, avec $i \in \text{ring}^{-1}(p)$. Comme α est équitable, on conclut qu'il existe nécessairement un élément $x_1 > x$ tel que $\lambda(x_1) \in \Sigma_C^p \setminus \Gamma$ (sinon, pour tout α'' trace préfixe vérifiant $\alpha_{\downarrow_\alpha x_0} \leq \alpha'' \leq \alpha$, $\overline{s}^p(\alpha'') \in Q^f \times \{\text{Token}_i'\}$ et $f^p(\alpha'') \in \Sigma_C^p \setminus \Gamma$). Soit $x' \in X$ le plus petit élément de l'ensemble $\{x \in X \setminus X' \mid \lambda(x) \in \Sigma_C^p \setminus \Gamma\}$. Alors il existe un unique processus $p' \in \text{Proc}$ tel que $\lambda(x') \in \Sigma^{p,p'}$: en fait, $p' = \text{ring}((i \bmod n) + 1)$.

   On pose maintenant $X'' = X' \cup \downarrow_\alpha x'$, et $\alpha'' = (X'', \leq, \lambda)$ trace préfixe de α. On a $\overline{s}^{p'}(\alpha'') \in \text{Token}$, et on peut répéter le raisonnement ci-dessus sur le processus p'. Comme $X' \subseteq X''$, et que l'application ring est surjective, en itérant le raisonnement, on obtient, pour tout $p \in \text{Proc}$, l'existence d'un élément $z \in X \setminus X'$ tel que $\lambda(z) \in \Sigma^{q,p}$, pour $q \in \text{Proc}$.

4. Supposons $x \parallel_\alpha y$. Alors $p \neq q$, et on considère la trace préfixe de α définie par $\alpha_1 = (\downarrow_\alpha x \cup \downarrow_\alpha y, \leq, \lambda)$. On remarque que $\downarrow_\alpha x \cap \lambda^{-1}(\Sigma^p) = (\downarrow_\alpha x \cup \downarrow_\alpha y) \cap \lambda^{-1}(\Sigma^p)$. (sinon, il existe $z \in \lambda^{-1}(\Sigma^p)$ tel que $z \leq y$ et $x < z$, ce qui implique $x < y$, en contradiction avec l'hypothèse). De même, $\downarrow_\alpha y \cap \lambda^{-1}(\Sigma^q) = (\downarrow_\alpha x \cup \downarrow_\alpha y) \cap \lambda^{-1}(\Sigma^q)$.

Donc $\overline{s}^p(\alpha_1) = \overline{s}^p(\pi_{\Sigma^p}(\alpha_1)) = \overline{s}^p(\pi^p_\Sigma(\alpha_{\downarrow_\alpha x})) = \overline{s}^p(\alpha_{\downarrow_\alpha x}) \in$ Token, et, par le même raisonnement, $\overline{s}^q(\alpha_1) \in$ Token, ce qui est impossible, d'après 1.

□

On montre à présent que la stratégie distribuée ainsi définie est gagnante.

Remarque 4.45. Il est important pour que la démonstration fonctionne, que l'on se réduise au problème de SSD asynchrone équitable sur le singleton, avec équité par *type d'actions*. En effet, une exécution équitable du singleton pour une stratégie f fixée dans le problème de SSD asynchrone équitable classique est telle que, si le singleton désire continuellement effectuer une action quelconque à partir d'un moment, alors le processus sera activé infiniment souvent. Ici on s'est réduit au problème dans lequel une exécution équitable du singleton est telle que, si le singleton désire continuellement effectuer une action *au sein de la même partie des actions* (typiquement, les signaux de communication externe d'un processus donné), alors ce type d'action sera joué infiniment souvent. Cela implique que dans une exécution équitable, si la stratégie du singleton est continuellement définie, mais pas toujours dans la même partie des actions, il n'est pas nécessaire que le singleton soit activé infiniment souvent. Ainsi, l'exécution *distribuée* correspondante, dans laquelle le processus ayant le jeton n'est jamais celui devant jouer selon la stratégie du singleton (lorsque le processus p a le jeton, le singleton veut jouer dans la partie des actions Out_q, etc.), et donc dans laquelle les processus ne désirent jamais jouer d'action externe, correspond bien à une exécution équitable du singleton, ce qui n'aurait pas été le cas avec la définition d'équité ne faisant pas de distinction entre les différentes actions du singleton.

Par ailleurs, on remarque que si l'on aurait pu définir le problème pour des partitions plus fines des actions des processus dans le cas distribué, on ne peut pas aller jusqu'à la partition la plus fine $\mathcal{P} = \{\{a\} \mid a \in \Sigma\}$. En effet, dans ce dernier cas, la notion d'équité ne permet plus d'assurer que dans une exécution le jeton soit transmis infiniment entre les processus : le processus ayant le jeton et désirant continuellement le transmettre ne veut pas envoyer le même signal à chaque instant, mais bien la valeur courante de l'histoire sur le singleton qu'il a reconstruite, qui peut changer à chaque nouveau signal reçu de l'environnement.

Soit $\alpha = (X, \leq, \lambda) \in \mathbb{R}(\Sigma, D)$ une exécution F-compatible et F-équitable. On définit une extension d'ordre $\alpha' = (X, \leq', \lambda)$ de la façon suivante : deux événements $x \in \lambda^{-1}(\Sigma^p)$ et $y \in \lambda^{-1}(\Sigma^q)$ concurrents dans α sont ordonnés par $x \leq' y$ si la prochaine action de communication après x *précède* la prochaine action de communication après y, ou bien si la prochaine action de communication après x et y est la même et est un signal de p vers q. L'ordre α' représente la séquence d'actions du singleton que les processus vont simuler au cours d'une exécution. En effet, si $x \in \lambda^{-1}(\Sigma^p)$ et $y \in \lambda^{-1}(\Sigma^q)$ sont concurrents dans α et que la prochaine action de communication z après x est telle que $z \in \lambda^{-1}(\Sigma^{p,q})$, alors cela signifie que p a le jeton en x. Donc jusqu'à l'événement z c'est le processus p qui peut envoyer des signaux externes. N'ayant pas connaissance de y, il ne peut simuler le singleton sur une histoire contenant y, c'est pourquoi on veut que les événements vus par p soit placés avant ceux vus par q. De façon générale, si le prochain événement de communication après x précède le prochain événement de communication après y (on a vu dans la remarque 4.44 que de tels événements de communication existent toujours, et de plus sont tous ordonnés), cela signifie qu'un processus a eu le jeton en x avant qu'un processus ait le jeton en y.

Formellement, pour tout $x \in X$, soit $x' = \min\{y \geq x \mid y \in \lambda^{-1}(\Sigma \setminus \Gamma)\}$. Un tel élément existe toujours (par la remarque 4.44 (3)) et est unique (par la remarque 4.44 (2)).

On définit alors

$$x \leq' y \text{ si et seulement si } \begin{cases} x \leq y & \text{ou} \\ x \parallel_\alpha y & \text{et } x' < y', \text{ ou} \\ x \parallel_\alpha y & \text{et } x' = y' \in \lambda^{-1}(\Sigma^{p,q}), \text{ et } x \in \lambda^{-1}(\Sigma^p),\ y \in \lambda^{-1}(\Sigma^q) \end{cases}$$

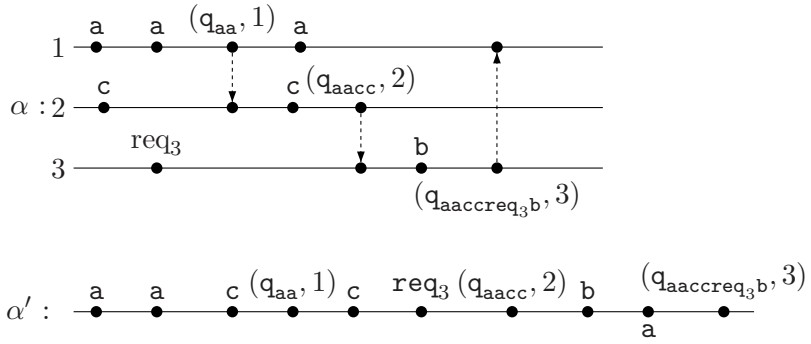


FIG. 4.10 – Une exécution distribuée et sa linéarisation

Avant de montrer que la relation ainsi définie est bien une relation d'ordre totale, on remarque les faits suivants :

Remarque 4.46.

1. Pour tout $x \in X$, on a soit $x = x'$, soit $x \in \lambda^{-1}(\Gamma)$ et il existe un unique processus $p \in \mathrm{Proc}$ tel que $\lambda(x) \in \Sigma^p \cap \Gamma$. De plus, l'ensemble $\{y \in X \mid x \leq y \leq x'\} \subseteq \lambda^{-1}(\Sigma^p)$.
2. si $x \parallel_\alpha y$ et $x' = y'$ alors $x, y \in \lambda^{-1}(\Gamma)$. En effet, si $x \in \lambda^{-1}(\Sigma \setminus \Gamma)$, alors $x = x' = y'$, et $y \leq y'$ implique que $y \leq x$, ce qui est en contradiction avec l'hypothèse.
3. Si $x \leq' y$, alors $x' \leq y'$. En effet, si $x \parallel_\alpha y$, alors on le déduit immédiatement de la définition. Si $x \leq y$ et $x = x'$, alors $x' \leq y \leq y'$. Si $x \leq y$ et $x < x'$, alors on pose $p \in \mathrm{Proc}$ l'unique processus tel que $x \in \lambda^{-1}(\Sigma^p)$. On a $\{z \in X \mid z \geq x\} = \{z \in X \mid x \leq z \leq x'\} \cup \{z \in X \mid z \geq x'\} = \{z \in \lambda^{-1}(\Sigma^p) \mid z \geq x\} \cup \{z \in X \mid z \geq x'\}$. Alors, soit $x \leq y < x'$ et $y' = x'$, soit $x' \leq y \leq y'$.
4. Si $x' < y'$ alors $x <' y$. En effet, si $x \leq y$ ou si $x \parallel_\alpha y$, on le déduit de la définition. Sinon, si $y < x$, alors $y < x \leq x' < y'$. Or y' est le plus petit élément vérifiant $y \leq y'$ et $y' \in \lambda^{-1}(\Sigma \setminus \Gamma)$. Comme $x' \in \lambda^{-1}(\Sigma \setminus \Gamma)$, on obtient une contradiction.
5. S'il existe $x \in \lambda^{-1}(\Gamma \cap \Sigma^{\mathrm{ring}(1)})$ tel que, pour tout $y \in \lambda^{-1}(\Sigma^{\mathrm{ring}(1)})$, $x \leq y$, alors pour tout $y \in X$, $x \leq' y$.

   En effet, soit $y \in X$. Alors $y \not< x$ (sinon il existe $z \in \lambda^{-1}(\Sigma^{\mathrm{ring}(1)})$ tel que $y \leq z < x$). Deux cas sont donc possibles :
   - Si $x \leq y$, alors $x \leq' y$.
   - si $x \parallel_\alpha y$, alors si $x' < y'$, on a $x <' y$. Si $y' < x'$, alors nécessairement $x \parallel_\alpha y'$ (le cas $y' < x$ implique que $y < x$ ce qui est impossible, et le cas $x < y' < x'$ est contradictoire avec la définition de x'). Soit $p \in \mathrm{Proc}$ tel que $y' \in \lambda^{-1}(\Sigma^p)$ et $\overline{s}^p(\alpha_{\downarrow_\alpha y'}) \in \mathrm{Token}$ (un tel processus existe, par la remarque 4.44 (1), car y' correspond à un passage de jeton). Puisque $x \parallel_\alpha y'$, $p \neq \mathrm{ring}(1)$. Or, par définition de $\mathfrak{A}^{\mathrm{ring}(1)}$, $\overline{s}^{\mathrm{ring}(1)}(\alpha_{\downarrow_\alpha x}) \in \mathrm{Token}$. Ceci

est impossible par la remarque 4.44 (4). Si enfin $x' = y'$, alors il existe $p \in \text{Proc}$ tel que $\lambda(x') \in \Sigma^{\text{ring}(1),p}$ et $\lambda(y) \in \Sigma^p$ (si $\lambda(x') \in \Sigma^{p,\text{ring}(1)}$, alors on aurait $\overline{s}^q(\alpha_{\downarrow_\alpha y}) \in \text{Token}$, et ceci est en contradiction avec l'hypothèse $x \parallel_\alpha y$, par la remarque 4.44 (4)). Donc $x \leq' y$.

Lemme 4.47. *α' est un ordre total.*

Démonstration. Montrons d'abord que $\leq'$ est une relation d'ordre.

- Il est clair que $x \leq' x$ par réflexivité de $\leq$.
- Soient $x, y \in X$ tels que $x \leq' y$ et $y \leq' x$. Si $x \leq y$, alors $y \leq' x$ implique que $y \leq x$. De même, si $y \leq x$ alors $x \leq' y$ implique que $x \leq y$. Donc si $x \leq y$ ou $y \leq x$, nécessairement, $x \leq y$ et $y \leq x$, et par antisymétrie de $\leq$ on en déduit que $x = y$. Si $x \parallel_\alpha y$, alors, d'après la remarque 4.46, $x' \leq y'$. De la même façon on déduit du fait que $y \leq' x$ que $y' \leq x'$. Donc, par antisymétrie de $\leq$, on obtient $x' = y'$. Alors, par la remarque 4.46, on sait que $x, y \in \lambda^{-1}(\Gamma)$. Soient $p, q \in \text{Proc}$ les uniques processus vérifiant $x \in \lambda^{-1}(\Sigma^p)$ et $y \in \lambda^{-1}(\Sigma^q)$. La définition de $\leq'$ implique que $x', y' \in \lambda^{-1}(\Sigma^{p,q}) \cap \lambda^{-1}(\Sigma^{q,p})$. Les alphabets de communication étant deux à deux disjoints, on obtient une contradiction. Donc si $x \leq' y$ et $y \leq' x$ on a $x = y$.
- Soient $x, y, z \in X$ tels que $x \leq' y$ et $y \leq' z$. On cherche à montrer qu'alors $x \leq' z$. Par la remarque 4.46, et par transitivité de $\leq$, on a $x' \leq y' \leq z'$. Si $x' < z'$, alors par définition de $\leq'$, on a $x \leq' z$. Sinon, $x' = y' = z'$. On distingue deux cas
  - si $x \parallel_\alpha y$, alors la remarque 4.46 permet de déduire qu'il existe un unique $p \in \text{Proc}$ et un unique $q \in \text{Proc}$ tels que respectivement, $x \in \lambda^{-1}(\Sigma^p)$ et $y \in \lambda^{-1}(\Sigma^q)$. Alors $x' = y' = z' \in \lambda^{-1}(\Sigma^{p,q})$, et $z \in \lambda^{-1}(\Sigma^p) \cup \lambda^{-1}(\Sigma^q)$.
    Si $z \in \lambda^{-1}(\Sigma^p)$, alors supposons que $z \leq x$. Cela implique que $z \leq' x$. Dans ce cas, on en déduit que $z \parallel_\alpha y$ (sinon, soit $y \leq z \leq x$, ce qui est en contradiction avec $x \parallel_\alpha y$, soit, y étant un événement étiqueté par une action locale, et x' étant la prochaine action partagée après z, on aurait $z \leq x' \leq y$. Or, $y < y' = x'$, on aboutit à une contradiction). Par définition de $\leq'$, comme $y' = z' \in \lambda^{-1}(\Sigma^{p,q})$, $z \leq' y$. Par antisymétrie de $\leq'$, on obtient $z = y$, ce qui est impossible. Donc $x \leq z$, et $x \leq' z$.
    Si $z \in \lambda^{-1}(\Sigma^q \setminus \Sigma^p)$, alors $z \in \lambda^{-1}(\Gamma)$ et soit $z < y$, donc $z <' y$ ce qui est en contradiction avec $y \leq' z$, soit $y \leq z$. Dans ce cas, comme $z \in \lambda^{-1}(\Gamma)$, alors $z < z'$, et $x \parallel_\alpha z$, avec $x' = z' \in \lambda^{-1}(\Sigma^{p,q})$ (une fois de plus, x et z étant des événements correspondant à des actions locales sur deux processus distincts, la prochaine action partagée après x et après z étant x', $x \leq z$ impliquerait $x' \leq z$ et $z \leq x$ impliquerait $x' \leq x$, ce qui est impossible). Donc, la définition de $\leq'$ permet de conclure que $x \leq' z$.
  - si $x \leq y$, alors par la remarque 4.46, soit $y = y'$, soit $y \in \lambda^{-1}(\Gamma)$. Si $y = y'$, alors la relation $z \leq z' = y'$ implique que $z \leq y$, et donc $z \leq' y$. Par antisymétrie de $\leq'$, on a alors $z = y$ et $x \leq z$ donc $x \leq' z$. Sinon, on a $x \leq y < x' = y'$ et il existe un unique processus $p \in \text{Proc}$ tel que $x, y \in \lambda^{-1}(\Sigma^p)$. Si $y \leq z$, alors on obtient immédiatement $x \leq' z$. Si $z < y$, alors $z <' y$, ce qui est impossible. Si enfin $y \parallel_\alpha z$, comme $y' = z'$ et $y \leq' z$, alors il existe $q \in \text{Proc}$ tel que $y', z' \in \lambda^{-1}(\Sigma^{p,q})$ avec $z \in \lambda^{-1}(\Sigma^q)$. Comme $x' = y' = z'$, $x \leq' z$.

On montre ensuite que $\leq'$ est une relation totale. Soient $p, q \in \text{Proc}$ et $x \in \lambda^{-1}(\Sigma^p)$ et $y \in \lambda^{-1}(\Sigma^q)$ tels que $x \parallel_\alpha y$. D'après la remarque 4.44 (2), $x' \leq y'$ ou $y' \leq x'$. Si $x' = y'$, alors nécessairement $\lambda(x') \in \Sigma^{p,q} \uplus \Sigma^{q,p}$. Donc $x \leq' y$ ou $y \leq' x$. □

On supprime à présent de α' les actions de communication internes. Soit $\alpha_f = \pi_\Gamma(\alpha')$, un

ordre total dont on va montrer qu'il correspond à une exécution f-compatible et f-équitable :

Lemme 4.48. *α_f est une exécution f-compatible.*

Pour cela on montre que, à chaque instant, le processus possédant le jeton, et donc à même de produire une sortie, maintient une histoire consistante avec la linéarisation α_f. On commence par définir l'histoire du singleton simulée par les processus : soit $p \in \mathrm{Proc}$ et $x \in \lambda^{-1}(\Sigma^p)$ tels que le processus p a le jeton en x. Alors à cet instant, l'état interne de $\mathfrak{A}^p$ est l'état dans lequel se trouverait l'automate $\mathfrak{A}^f$ après avoir vu les événements observables *dans le passé* de x. Formellement on note

Notation 4.49. Pour tout $\alpha_1 = (X_1, \leq, \lambda) \in \mathbb{M}(\Sigma, D)$ trace préfixe de l'exécution α, soit $p \in \mathrm{Proc}$ tel que $\overline{s}^p(\alpha_1) \in \mathrm{Token}$. On note

$$\zeta(\alpha_1) = \begin{cases} \max(X_1 \cap \lambda^{-1}(\Sigma^p)) & \text{si } X_1 \cap \lambda^{-1}(\Sigma^p) \neq \emptyset \\ \text{indéfini} & \text{sinon.} \end{cases}$$

On définit alors l'ordre total

$$\overline{\alpha_1} = (\overline{X_1}, \leq', \lambda)$$

avec $\overline{X_1} = {\downarrow_\alpha}\zeta(\alpha_1)$ et la convention que ${\downarrow_\alpha}\zeta(\alpha_1) = \emptyset$ si $\zeta(\alpha_1)$ n'est pas défini.

Remarque 4.50. si $X_1 \cap \lambda^{-1}(\Sigma^p) = \emptyset$ et $\overline{s}^p(\alpha_1) \in \mathrm{Token}$, nécessairement, $p = \mathrm{ring}(1)$, et $\overline{s}^p(\alpha_1) = \{s_0^f\} \times \mathrm{Token}_p$.

On va utiliser les deux lemmes suivants : le premier établit le fait que lorsqu'un processus a le jeton en x, l'ensemble des événements dans le passé causal de x dans α est exactement l'ensemble des événements dans le passé causal de x dans α'.

Lemme 4.51. *Pour tout $\alpha_1 = (X_1, \leq, \lambda) \in \mathbb{M}(\Sigma, D)$ préfixe de α, soit $p \in \mathrm{Proc}$ tel que $\overline{s}^p(\alpha_1) \in \mathrm{Token}$. Alors, si $X_1 \cap \lambda^{-1}(\Sigma^p) \neq \emptyset$, ${\downarrow_\alpha}\zeta(\alpha_1) = {\downarrow_{\alpha'}}\zeta(\alpha_1)$.*

Démonstration du lemme 4.51. Comme α' est une extension linéaire de α, on a immédiatement ${\downarrow_\alpha}\zeta(\alpha_1) \subseteq {\downarrow_{\alpha'}}\zeta(\alpha_1)$. Soit donc $y \in X$ tel que $y \leq' \zeta(\alpha_1)$, et on va montrer que $y \leq \zeta(\alpha_1)$. Dans la suite de cette démonstration on pose $z = \zeta(\alpha_1)$. Soit $\alpha_{\downarrow_\alpha y'}$ la trace préfixe de α constituée des éléments dans le passé de $y' = \min\{z \geq y \mid z \in \lambda^{-1}(\Sigma \setminus \Gamma)\}$, et soit $r \in \mathrm{Proc}$ le processus tel que $\overline{s}^r(\alpha_{\downarrow_\alpha y'}) \in \mathrm{Token}$. Par la remarque 4.44 (1), un tel processus existe toujours, et est unique. De plus, $\lambda(y') \in \Sigma^r$. Comme $\overline{s}^p(\alpha_1) \in \mathrm{Token}$, et que z est l'élément maximal de $X_1 \cap \lambda^{-1}(\Sigma^p)$, par la remarque 4.44 (4) on sait que $y' \leq z$ ou $z \leq y'$. Si $y' \leq z$, alors immédiatement $y \leq z$. Si $z \leq y'$, alors soit $y = y'$, et alors puisque $y \leq' z$, nécessairement $z = y$, soit $y < y'$ et supposons que $y \parallel_\alpha z$. Alors on a $z \leq z' \leq y'$ et, par définition de $\leq'$, $y' \leq z'$, donc $y' = z'$. Par définition de $\leq'$, cela implique que $z' \in \lambda^{-1}(\Sigma^{q,p})$ et $y \in \lambda^{-1}(\Sigma^q)$ et $z \in \lambda^{-1}(\Sigma^p)$. Par définition de $\overline{f}^q$, et comme α est F-compatible, $\overline{s}^q(\alpha_{\Downarrow_\alpha y'}) \in \mathrm{Token}$. Alors, puisque quel que soit y_1 tel que $y \leq y_1 < y'$, $\lambda(y_1) \in \Gamma$, on en déduit que $\overline{s}^q(\alpha_{\downarrow_\alpha y}) \in \mathrm{Token}$, ce qui est en contradiction avec $y \parallel_\alpha z$ (par la remarque 4.44 (4)). Donc $y \leq z$, et ${\downarrow_\alpha}\zeta(\alpha_1) = {\downarrow_{\alpha'}}\zeta(\alpha_1)$. □

Le second lemme affirme que l'état de $\mathfrak{A}^f$ simulé par un processus ayant le jeton correspond à l'état que $\mathfrak{A}^f$ aurait atteint après avoir exécuté les actions dans le passé causal de l'événement courant de α, mais ordonnées par $\leq'$.

Lemme 4.52. *Pour tout $\alpha_1 = (X_1, \leq, \lambda) \in \mathbb{M}(\Sigma, D)$ préfixe de α, pour tout $p \in$ Proc tel que $\overline{s}^p(\alpha_1) \in$ Token, $\overline{s}^p(\alpha_1) \in \{\overline{s}^f(\pi_\Gamma(\overline{\alpha_1}))\} \times \text{Token}_p$.*

Ainsi, α_f étant la projection de α' sur les actions de Γ, on en déduit que le processus ayant le jeton simule correctement l'exécution α_f.

Avant de prouver cette affirmation, on montre comment les deux lemmes précédents permettent de démontrer le résultat :

Démonstration du lemme 4.48. Soit $p \in$ Proc et $x \in \lambda^{-1}(\text{Out}_p)$. Comme α est une exécution F-compatible, $\lambda(x) = \overline{f}^p(\overline{s}^p(\alpha_{\Downarrow_\alpha x}))$. Ceci implique qu'il existe $i \in \text{ring}^{-1}(p)$, et $s \in Q^f$ tels que $\overline{s}^p(\alpha_{\Downarrow_\alpha x}) = (s, \text{Token}_i)$, et $\overline{f}(s) = \lambda(x)$. Comme $\alpha_{\Downarrow_\alpha x}$ est un préfixe de α, le lemme 4.52 nous dit que $\overline{s}^f(\pi_\Gamma(\overline{\alpha_{\Downarrow_\alpha x}})) = s$, avec $\overline{\alpha_{\Downarrow_\alpha x}} = (\overline{\Downarrow_\alpha x}, \leq', \lambda)$. Soit $\alpha_1^f = (\Downarrow_{\alpha_f} x, \leq', \lambda)$. On veut alors montrer que $\pi_\Gamma(\overline{\alpha_{\Downarrow_\alpha x}}) = \alpha_1^f$. Ainsi, $\overline{s}^f(\pi_\Gamma(\overline{\alpha_{\Downarrow_\alpha x}})) = \overline{s}^f(\alpha_1^f) = s$ et $\lambda(x) = \overline{f}(\overline{s}^f(\alpha_1^f))$.

Si x n'a pas de prédécesseur sur $\lambda^{-1}(\Sigma^p)$, alors $\overline{\Downarrow_\alpha x} = \emptyset$, et $p = \text{ring}(1)$. Par la remarque 4.46 (5), on sait que x est l'élément minimal de α', donc de α_f. Donc $\Downarrow_{\alpha_f} x = \emptyset$, et $\pi_\Gamma(\overline{\alpha_{\Downarrow_\alpha x}}) = \alpha_1^f$.

Sinon, $\overline{\Downarrow_\alpha x} = \downarrow_\alpha \zeta(\alpha_1)$. Dans ce cas, le lemme 4.51 implique que $\downarrow_\alpha \zeta(\alpha_1) = \downarrow'_\alpha \zeta(\alpha_1) = \{y \in X \mid y \leq' \zeta(\alpha_1)\} = \{y \in X \mid y <' x\}$ (x étant un événement local, l'ensemble de ses prédécesseurs stricts est égal à l'ensemble des événements dans le passé de son unique prédécesseur. Or, il est clair que $\zeta(\alpha_1)$ est le prédécesseur de x dans α). Donc $\pi_\Gamma(\overline{\alpha_{\Downarrow_\alpha x}}) = \alpha_1^f$.

On en déduit que $\alpha_f \in \Gamma^\infty$ est bien une exécution f-compatible. □

Démonstration du lemme 4.52. On procède par récurrence sur les préfixes de α.

Si α_1 est la trace vide, alors $\overline{s}^{\text{ring}(1)}(\alpha_1) \in \{s_0^f\} \times \text{Token}_{\text{ring}(1)}$. De plus, $\overline{\alpha_1}$ est également vide, donc on a bien $\overline{s}^{\text{ring}(1)}(\alpha_1) \in \{\overline{s}^f(\pi_\Gamma(\overline{\alpha_1}))\} \times \text{Token}_{\text{ring}(1)}$.

Soit $\alpha_1 = (X_1, \leq, \lambda)$ une trace préfixe de α et $p \in$ Proc tel que $\overline{s}^p(\alpha) \in$ Token. Soit $x \in X \setminus X_1$ tel que $\Downarrow x \subseteq X_1$. Soit $a = \lambda(x)$. On a $\alpha_1 \cdot a = (X_1 \cup \{x\}, \leq, \lambda)$ est un préfixe de α. Si $a \notin \Sigma^p$, par la remarque 4.44 (1), $\overline{s}^p(\alpha_1 \cdot a) = \overline{s}^p(\alpha_1)$. Par hypothèse de récurrence, $\overline{s}^p(\alpha_1 \cdot a) \in \{\overline{s}^f(\pi_\Gamma(\overline{\alpha_1}))\} \times \text{Token}_p$. De plus, les éléments de $\overline{(\alpha_1 \cdot a)}$ sont exactement $\overline{X_1}$, puisque $\max(X_1 \cap \lambda^{-1}(\Sigma^p)) = \max(X_1 \cup \{x\} \cap \lambda^{-1}(\Sigma^p))$. Donc $\overline{\alpha_1 \cdot a} = \overline{\alpha_1}$ et $\overline{s}^p(\alpha_1 \cdot a) \in \{\overline{s}^f(\pi_\Gamma(\overline{\alpha_1 \cdot a}))\} \times \text{Token}_p$.

Si $a \in \Sigma^p \cap \Gamma$, alors $\overline{s}^p(\alpha_1 \cdot a) \in$ Token, et, en utilisant l'hypothèse de récurrence, $\overline{s}^p(\alpha_1 \cdot a) \in \{\delta^f(\overline{s}^f(\pi_\Gamma(\overline{\alpha_1})), a)\} \times \text{Token}_p$. Par ailleurs, $x = \max(X_1 \cup \{x\} \cap \lambda^{-1}(\Sigma^p))$ et $\Downarrow_\alpha x = \downarrow_\alpha \max(X_1 \cap \lambda^{-1}(\Sigma^p))$ (car x est un événement local), donc $\overline{\alpha_1 \cdot a} = \overline{\alpha_1} \cdot a$. Donc $\delta^f(\overline{s}^f(\pi_\Gamma(\overline{\alpha_1})), a) = \overline{s}^f(\pi_\Gamma(\overline{\alpha_1 \cdot a}))$. On obtient donc $\overline{s}^p(\alpha_1 \cdot a) \in \{\overline{s}^f(\pi_\Gamma(\overline{\alpha_1 \cdot a}))\} \times \text{Token}_p$.

Le cas le plus délicat est si $a \in \Sigma^p \setminus \Gamma$. Il s'agit du moment où les processus se transmettent le jeton, et où ils réordonnent les événements concurrents qui ont eu lieu : ceux transmis par le processus émetteur du signal, et ceux reçus par le processus récepteur tant qu'il n'avait pas le jeton. Il s'agit de montrer formellement que l'état calculé par le processus récepteur correspond bien à l'état atteint par $\mathfrak{A}^f$ après avoir vu les événements dans le passé de ce signal de communication, *ordonnés par* $\leq'$. Soit donc $q \in$ Proc tel que $\overline{s}^q(\alpha_1 \cdot a) \in$ Token. On définit last-token(x) le dernier événement de Σ^q correspondant à un envoi de jeton. Formellement, on pose

$$\text{last-token}(x) = \max(\lambda^{-1}(\Sigma^q) \cap \downarrow_\alpha \zeta(\alpha_1))$$

avec la convention que $\downarrow_\alpha \zeta(\alpha_1) = \emptyset$ si $\zeta(\alpha_1)$ n'est pas défini. Il est clair que last-token(x) correspond bien au dernier événement de communication sur q avant x : comme $\zeta(\alpha_1) \in \Sigma^p$,

et $p \neq q$, le plus grand événement dans $\lambda^{-1}(\Sigma^q \cap \downarrow_\alpha \zeta(\alpha_1))$ est nécessairement dans $\lambda^{-1}(\Sigma^q \setminus \Gamma)$. De plus, il n'existe aucun événement $z \in \lambda^{-1}(\Sigma^q \setminus \Gamma)$ tel que last-token$(x) < z < x$. Si un tel z existe, alors soit $z \parallel_\alpha \zeta(\alpha_1)$, et alors il existe un processus $p' \neq p$ tel que p' a le jeton en z. Comme, par ailleurs, p a le jeton en $\zeta(\alpha_1)$, la remarque 4.44 (4) nous dit que c'est impossible. Sinon $\zeta(\alpha_1) < z$, et on obtient une contradiction avec la définition de $\zeta(\alpha_1)$ (si $z \in \lambda^{-1}(\Sigma^p$, alors $\zeta(\alpha_1) \neq \max(X_1 \cap \lambda^{-1}(\Sigma^p))$, si $z \notin \lambda^{-1}(\Sigma^p)$, alors il existe $z' \in \lambda^{-1}(\Sigma^p)$ tel que $\zeta(\alpha_1) < z' < z$). On définit alors l'ordre total formé des actions locales observées par q entre last-token(x) et x :

$$\alpha(\text{last-token}(x)) = (X_{\alpha(\text{last-token}(x))}, \leq, \lambda)$$

où $X_{\alpha(\text{last-token}(x))} = \lambda^{-1}(\Sigma^q) \cap (\Downarrow_\alpha x \setminus \downarrow_\alpha \zeta(\alpha_1))$.

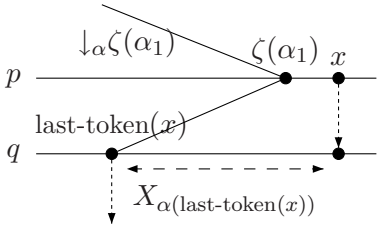


FIG. 4.11 – Les éléments de $X_{\alpha(\text{last-token}(x))}$

Si last-token(x) existe, nécessairement, c'est une émission du jeton par q, donc par définition de $\mathfrak{A}^q$, $\overline{s}^q(\alpha_{\Downarrow_\alpha \text{last-token}(x)}) \in \text{Token}$, et on en déduit que $\overline{s}^q(\alpha_1) = (\delta_{\alpha(\text{last-token}(x))}, \text{NToken}_i)$, pour $i \in \text{ring}^{-1}(q)$. La définition de δ^q et l'hypothèse de récurrence permettent de conclure que

$$\overline{s}^q(\alpha_1 \cdot a) \in \{\delta^f(s, \alpha(\text{last-token}(x)))\} \times \text{Token}_q.$$

où $s = \overline{s}^f(\pi_\Gamma(\overline{\alpha_1}))$.

Soit $\overline{\alpha_1 \cdot a} = (X_{\overline{\alpha_1 \cdot a}}, \leq, \lambda)$. Par définition, $X_{\overline{\alpha_1 \cdot a}} = \downarrow_\alpha \zeta(\alpha_1 \cdot a) = \downarrow_\alpha x$, $\overline{X_1} = \downarrow_\alpha \zeta(\alpha_1)$ et $X_{\alpha(\text{last-token}(x))} = \lambda^{-1}(\Sigma^q) \cap (\Downarrow_\alpha x \setminus \downarrow_\alpha \zeta(\alpha_1))$. En fait, $\Downarrow_\alpha x \setminus \downarrow_\alpha \zeta(\alpha_1) \subseteq \lambda^{-1}(\Sigma^q)$. En effet, puisque $x \in \lambda^{-1}(\Sigma^{p,q})$, si on note $x_q = \max(\lambda^{-1}(\Sigma^q) \cap \Downarrow_\alpha x)$, on a $\Downarrow_\alpha x = (\downarrow_\alpha \zeta(\alpha_1)) \cup (\downarrow_\alpha x_q)$. Or, tout $z \in \lambda^{-1}(\Sigma^q)$ tel que last-token$(x) < z < x$ est tel que $z \in \lambda^{-1}(\Gamma)$, donc tout élément de $\Downarrow_\alpha x \setminus \downarrow_\alpha \zeta(\alpha_1) \in \lambda^{-1}(\Sigma^q)$. Si $\downarrow_\alpha \zeta(\alpha_1) = \emptyset$, alors nécessairement $p = \text{ring}(1)$, et x est la première communication de l'exécution. Donc $\Downarrow_\alpha x \subseteq \lambda^{-1}(\Sigma^q)$. Il est donc évident à partir de ces définitions que $\overline{X_1} \cup X_{\alpha(\text{last-token}(x))} \cup \{x\} = \downarrow_\alpha \zeta(\alpha_1) \cup (\Downarrow_\alpha x \setminus \downarrow_\alpha \zeta(\alpha_1)) \cup \{x\} = X_{\overline{\alpha_1 \cdot a}}$.

Pour tout $y \in X_{\alpha(\text{last-token}(x))}$, on a $\zeta(\alpha_1) \leq' y$. En effet, ou bien $\zeta(\alpha_1) < y$, ou bien $\zeta(\alpha_1) \parallel_\alpha y$ et alors $\zeta(\alpha_1)' \leq x = y'$. Alors, deux cas sont possibles : soit $\zeta(\alpha_1)' = \zeta(\alpha_1) < x = y'$ et donc $\zeta(\alpha_1) \leq' y$. Soit $\zeta(\alpha_1)' = x = y' \in \lambda^{-1}(\Sigma^{p,q})$ donc $\zeta(\alpha_1) \leq' y$. Donc $\overline{\alpha_1 \cdot a} = \overline{\alpha_1} \cdot \alpha(\text{last-token}(x)) \cdot a$ et $\pi_\Gamma(\overline{\alpha_1 \cdot a}) = \pi_\Gamma(\overline{\alpha_1}) \cdot \alpha(\text{last-token}(x))$.

On a donc bien $\overline{s}^f(\pi_\Gamma(\overline{\alpha_1 \cdot a})) = \delta^f(s, \alpha(\text{last-token}(x)))$ avec $s = \overline{s}^f(\pi_\Gamma(\overline{\alpha_1}))$, ce qui conclut la démonstration. □

Lemme 4.53. *α_f est une exécution f-équitable, pour la partition $\mathcal{P} = \{\text{Out}_p \mid p \in \text{Proc}\}$.*

Démonstration. Soit un processus $p \in \text{Proc}$ et $\alpha'_f = (X'_f, \leq', \lambda)$ une trace préfixe de α_f telle que pour toute trace α''_f vérifiant $\alpha'_f \leq \alpha''_f \leq \alpha_f$, on a $\overline{f}(\overline{s}^f(\alpha''_f)) \in \text{Out}_p$. On considère

alors une telle trace préfixe $\alpha''_f = (X''_f, \leq', \lambda)$. Soit $q \in \text{Proc}$ et $x \in \lambda^{-1}(\Sigma^{q,p}) \setminus X''_f$ (un tel événement existe d'après la remarque 4.44 (3)). Supposons que, pour tout $y \geq x$, $\lambda(y) \notin \text{Out}_p$. Alors on montre que

$$\forall y \geq x \text{ tel que } \lambda(y) \in \Sigma^p, \overline{s}^p(\alpha_{\downarrow_\alpha y}) = (\overline{s}^f(\overline{\alpha_{\downarrow_\alpha y}}^f), \text{Token}_i) \qquad P(y)$$

avec $i \in \text{ring}^{-1}(p)$, et $\overline{\alpha_{\downarrow_\alpha y}}^f = \pi_\Gamma(\overline{\alpha_{\downarrow_\alpha y}})$.

On démontre la propriété $P(y)$ par récurrence. On sait que $\overline{s}^p(\alpha_{\downarrow_\alpha x}) \in \text{Token}$, donc $x = \zeta(\alpha_{\downarrow_\alpha x})$. Alors, par le lemme 4.51, $\downarrow_\alpha x = \downarrow_{\alpha'} x$. Par le lemme 4.52, on obtient $\overline{s}^p(\alpha_{\downarrow_\alpha x}) \in \{\overline{s}^f(\overline{\alpha_{\downarrow_\alpha x}}^f)\} \times \text{Token}$. Par ailleurs, $\leq'$ étant un ordre total, on sait que $X''_f \subseteq \downarrow_{\alpha'} x$ (puisque $x \notin X''_f$). Donc $\alpha'_f \leq \alpha''_f \leq \overline{\alpha_{\downarrow_\alpha x}}^f$, et $\overline{f}(\overline{s}^f(\overline{\alpha_{\downarrow_\alpha x}}^f)) \in \text{Out}_p$. Donc par l'équation (4.4) de la définition de δ^p, on en déduit que $\overline{s}^p(\alpha_{\downarrow_\alpha x}) = (\overline{s}^f(\overline{\alpha_{\downarrow_\alpha x}}^f), \text{Token}_i)$, pour $i \in \text{ring}^{-1}(p)$. Ceci constitue le cas de base de la récurrence.

Soit à présent $y \geq x$, tel que $\lambda(y) \in \Sigma^p$, et supposons $P(y)$ vérifiée. Alors par l'invariant de $\mathfrak{A}^p$ relevé dans la remarque 4.42, puisque $\overline{s}^p(\alpha_{\downarrow_\alpha y}) = (\overline{s}^f(\overline{\alpha_{\downarrow_\alpha y}}^f), \text{Token}_i)$, pour $i \in \text{ring}^{-1}(p)$, on a $\overline{f^p}(\overline{s}^p(\alpha_{\downarrow_\alpha y})) \in \text{Out}_p$.

Soit z le successeur de y sur Σ^p, et soit $a \in \Sigma^p$ tel que $\lambda(z) = a$. Si $\lambda(z) = \overline{f^p}(\overline{s}^p(\alpha_{\downarrow_\alpha y})) \in \text{Out}_p$, alors il existe $y \geq x$ tel que $\lambda(y) \in \text{Out}_p$, ce qui est en contradiction avec l'hypothèse. Donc, par définition des stratégies, $\lambda(z) \in \text{In}_p$, et $\overline{s}^p(\alpha_{\downarrow_\alpha z}) \in \text{Token}$. On sait que $\overline{\alpha_{\downarrow_\alpha z}} = \overline{\alpha_{\downarrow_\alpha y}} \cdot a$. Comme $\alpha''_f \leq \overline{\alpha_{\downarrow_\alpha y}}^f \leq \overline{\alpha_{\downarrow_\alpha z}}^f$, on a donc par hypothèse $\overline{f}(\overline{s}^f(\overline{\alpha_{\downarrow_\alpha z}}^f)) \in \text{Out}_p$, avec $\overline{s}^f(\overline{\alpha_{\downarrow_\alpha z}}^f) = \delta^f(\overline{s}^f(\overline{\alpha_{\downarrow_\alpha y}}^f), a)$. Donc, en appliquant la fonction de transition δ^p de l'équation (4.3), on obtient $\overline{s}^p(\alpha_{\downarrow_\alpha z}) = \delta^p(\overline{s}^p(\alpha_{\downarrow_\alpha y}), a) = (\delta^f(\overline{s}^f(\overline{\alpha_{\downarrow_\alpha y}}^f), a), \text{Token}_i) = (\overline{s}^f(\overline{\alpha_{\downarrow_\alpha z}}^f), \text{Token}_i)$, et $P(z)$ est vérifiée.

L'invariant de $\mathfrak{A}^p$ permet de conclure que pour tout $y \geq x$ tel que $\lambda(y) \in \Sigma^p$, $f^p(\alpha_{\downarrow_\alpha y}) \in \text{Out}_p$, donc pour toute trace α'' telle que $\alpha_{\downarrow_\alpha x} \leq \alpha'' \leq \alpha$, $F(\alpha'') \cap \Sigma^p_C \neq \emptyset$. Comme α est F-équitable, ceci est en contradiction avec l'hypothèse que pour tout $x' \geq x$, $\lambda(x') \notin \text{Out}_p$. Donc il existe $x' \geq x$, $\lambda(x') \in \text{Out}_p$, et pour tout $\alpha''_f = (X''_f, \leq', \lambda)$ trace préfixe de α_f telle que $\alpha'_f \leq \alpha''_f \leq \alpha_f$, il existe $x \in X_f \setminus X''_f$ tel que $\lambda(x) \in \text{Out}_p$, donc α_f est f-équitable pour la partition $\{\text{Out}_p \mid p \in \text{Proc}\}$. □

Il nous reste à montrer que l'exécution observable $\pi_\Gamma(\alpha)$ est un affaiblissement de α_f. On rappelle (voir la définition donnée par l'équation (4.1) page 111) que pour un ordre partiel $t = (X, \leq, \lambda)$, on définit l'ensemble

$$W_t = \{(x, x') \in X^2 \mid \exists p \in \text{Proc},\ \lambda(x) \notin \Sigma^p \wedge \lambda(x') \in \text{In}_p \wedge z < z' \\ \wedge (\neg \exists y, \lambda(y) \in \text{Out}_p \wedge x < y < x')\}$$

comme étant l'ensemble maximal de paires d'éléments pour lesquels la relation d'ordre peut être supprimée lors d'un affaiblissement.

On va pour cela utiliser le lemme suivant :

Lemme 4.54. *Pour tous $x, y \in \lambda^{-1}(\Gamma)$ tels que $x \parallel_\alpha y$, si $x \leq' y$ alors $\lambda(y) \in \text{In}$.*

Démonstration. Soient $x, y \in \lambda^{-1}(\Gamma)$ concurrents dans α et tels que $x \leq' y$. Supposons qu'il existe $q \in \text{Proc}$ tel que $\lambda(y) \in \text{Out}_q$. Alors il existe $p \neq q \in \text{Proc}$ tel que $\lambda(x) \in \text{In}_p$ (on rappelle que d'après la remarque 4.44 (2), les signaux contrôlables par le système, donc en particulier les sorties, sont tous ordonnés dans α). Considérons les deux événements étiquetés par des signaux de communication interne x' et y' vérifiant $x' = \min\{z \geq x \mid z \in \lambda^{-1}(\Sigma \setminus \Gamma)\}$

et $y' = \min\{z \geq y \mid z \in \lambda^{-1}(\Sigma \setminus \Gamma)\}$. Par définition de $\leq'$, on a $x' \leq y'$. Comme $\lambda(y) \in \mathrm{Out}_q$, nécessairement, $\overline{s}^q(\downarrow_\alpha y) \in \mathrm{Token}$, et le prochain signal de communication impliquant q est un signal émis par q. Donc, si $x' = y'$, $y' \notin \lambda^{-1}(\Sigma^{p,q})$ et on n'a pas $x \leq' y$. Donc $x' < y'$, et même $x' \leq z' < y'$ où $z' \in \lambda^{-1}(\Sigma^q \setminus \Gamma)$ est le signal de communication précédant y', i.e., tel que pour tout $z' < z < y'$, si $\lambda(z) \in \Sigma^q$, alors $\lambda(z) \in \Gamma$. Par définition de y', $z' < y$. On obtient donc $x < x' \leq z' < y$, ce qui est en contradiction avec l'hypothèse que $x \parallel_\alpha y$.

Donc $\lambda(y) \in \mathrm{In}$. □

On peut à présent conclure la démonstration. Soient z_1 et $z_2 \in \lambda^{-1}(\Gamma)$ tels que $z_1 <_{\alpha_f} z_2$, i.e., $z_1 <' z_2$ et $z_1 \parallel_\alpha z_2$. On va montrer que $(z_1, z_2) \in W_{\alpha_f}$. D'après le lemme 4.54, on sait que $\lambda(z_2) \in \mathrm{In}$. Soit $p \in \mathrm{Proc}$ le processus tel que $\lambda(z_2) \in \mathrm{In}_p$. Alors $\lambda(z_1) \notin \Sigma^p$. Supposons maintenant qu'il existe y tel que $\lambda(y) \in \mathrm{Out}_p$ et $z_1 <' y <' z_2$. Alors $z_1 \parallel_\alpha y$ (sinon on aurait $z_1 \leq z_2$). Or, le lemme 4.54 implique que $\lambda(y) \in \mathrm{In}$, ce qui contredit $\lambda(y) \in \mathrm{Out}_p$. Donc $(z_1, z_2) \in W_{\alpha_f}$. On en déduit que $\pi_\Gamma(\alpha)$ est une extension de l'affaiblissement de α_f.

On a montré que α_f est une exécution f-compatible et f-équitable pour la partition des actions $\mathcal{P} = \{\mathrm{Out}_p \mid p \in \mathrm{Proc}\}$. Alors, f étant une stratégie gagnante pour $(\overline{\mathcal{S}}, \varphi)$, on a $\alpha_f \models \varphi$. Donc, par les propriétés de clôture d'AlocTL, $\pi_\Gamma(\alpha) \models \varphi$. On en conclut que F est une stratégie distribuée gagnante pour $(\mathcal{A}, \varphi)$, où $\mathcal{A}$ est l'architecture induite par l'alphabet de communication $(\Sigma^{p,q})_{(p,q)\in R}$. □

Les propositions 4.35 et 4.41 permettent donc de démontrer le résultat annoncé par le théorème 4.34. En effet, pour résoudre le problème de SSD asynchrone équitable sur $(\mathcal{S}, \varphi)$ avec $\mathcal{S}$ une structure fortement connexe, et $\varphi \in$ AlocTL, il suffit de trouver une stratégie gagnante pour le problème de SSD asynchrone équitable, *vis-à-vis d'une partition particulière des actions*, sur $(\overline{\mathcal{S}}, \varphi)$, avec $\overline{\mathcal{S}}$ et φ une structure singleton et une spécification LTL tels que définis page 126. Par le théorème 4.22, ce dernier problème est décidable, et de plus s'il existe une stratégie gagnante, il en existe une à mémoire finie. Donc, d'après la proposition 4.41, on en conclut que s'il existe une stratégie distribuée gagnante pour $(\mathcal{S}, \varphi)$, il en existe une à mémoire finie.

3 Bilan

On a défini un nouveau modèle pour le problème de synthèse de systèmes distribués asynchrones, dans lequel on a considéré des notions d'équité, pour lequel la sous-classe des structures fortement connexes est décidable. Ceci constitue une augmentation du nombre d'architectures décidables par rapport au cas synchrone : en effet, on a vu qu'en général le problème de synthèse de systèmes distribués synchrones n'est pas décidable pour les architectures fortement connexes. On montrera également dans un travail à soumettre prochainement que le problème de SSD asynchrone équitable est décidable pour les structures totalement déconnectées telles que celles correspondant au graphe de communication représenté sur la figure 2.4 page 32, ainsi que celles correspondant à la figure 4.1 page 104.

Chapitre 5

Conclusion

1 Bilan

Ce travail avait pour but d'étudier les différentes causes d'indécidabilité du problème de synthèse des systèmes distribués ouverts, afin de proposer des restrictions raisonnables sur les hypothèses permettant de dégager des cas particuliers – naturels d'un point de vue applicatif – qui soient décidables.

Systèmes synchrones Historiquement, la synthèse des systèmes distribués a été abordée en supposant des exécutions synchrones des processus. Sous ces hypothèses, le nombre de cas décidables est très restreint. On a montré qu'autoriser les spécifications à contraindre toutes les variables du système était une hypothèse trop forte pour l'établissement d'un critère de décidabilité basé sur la structure des architectures. En effet, les seules architectures décidables sous ces hypothèses sont celles dans lesquelles l'information provenant de l'environnement est transmise linéairement (sans branchement) parmi les processus. Au contraire, on a mis en évidence que restreindre les spécifications aux formules ne décrivant que les comportements acceptables externes permet d'agrandir substantiellement la classe des architectures décidables, tout en restant une restriction naturelle d'un point de vue pratique. Par ailleurs, si le problème de synthèse est décidable pour les architectures UWC si et seulement si elles sont à information linéairement préordonnée, on peut obtenir la décidabilité du problème pour toute cette sous-classe en se restreignant aux spécifications robustes : contrairement aux spécifications externes générales, les spécifications robustes tiennent compte de la structure de l'architecture en ne mettant en relation que les valeurs de variables connectées dans l'architecture. Enfin, on a mis en lumière le fait intéressant que, bien qu'il importe peu pour la décidabilité que les processus en sortie reçoivent l'information de l'environnement avec un retard (dû aux délais de transmission dans l'architecture), il est par contre crucial qu'ils reçoivent *toute* l'information possible. En effet, la preuve d'indécidabilité de [PR90] reposait fortement sur le fait que la spécification faisait dépendre les valeurs en sortie d'un processus de valeurs en entrée auxquelles il n'avait pas accès. On a montré que, si ce processus peut connaître ces valeurs, non seulement la preuve d'indécidabilité ne fonctionne plus, mais l'architecture devient décidable pour le problème de SSD synchrone (car elle devient UWC à information linéairement préordonnée). Mais on a également montré qu'il suffit que le processus manque un bit sur toute la séquence des valeurs de cette variable en entrée pour que l'architecture redevienne indécidable.

Systèmes asynchrones Pour les systèmes asynchrones, la situation semble moins critique. On a défini un nouveau cadre pour le problème de synthèse de systèmes asynchrones, qui correspond à une restriction d'une part des communications à des communications par *signaux*, et d'autre part des spécifications à des spécifications externes et *acceptables*. Par ailleurs, on a considéré le problème de synthèse sous des conditions d'équité pour les processus. On a montré que dans ce cas, on obtient la décidabilité du problème pour une sous-classe intéressante d'architectures, les architectures fortement connexes, qui ne sont en général pas décidables, en particulier dans le cas synchrone. La démonstration fonctionne par une réduction au cas particulier du singleton, ce qui nous donne en plus l'assurance de l'existence d'une stratégie à mémoire finie, dans le cas où une stratégie distribuée est possible.

Le modèle choisi autorise une mémoire uniquement locale aux contrôleurs, qui est plus concrète que la mémoire causale considérée dans [GLZ04, MTY05]. Il est envisageable que l'on obtienne la décidabilité du problème pour l'ensemble des architectures dans notre cadre, le résultat présenté dans ce document ne constituant qu'une première étape de résolution.

2 Perspectives

Dans les deux cas, les solutions apportées dans ce document ouvrent un certain nombre de pistes de recherche future. Dans le cas de systèmes synchrones, on a démontré que le problème de synthèse est décidable pour toutes les architectures UWC lorsqu'on se restreint aux spécifications robustes. Il serait intéressant de voir si cela est toujours vrai pour les architectures bien connectées. On peut également chercher à étendre encore la classe des architectures décidables lorsqu'elles sont à information linéairement préordonnée. La technique de preuve pour les architectures UWC consiste à découper le « travail » des processus en deux étapes : une partie routage de l'information, et une partie calcul proprement dit par les processus en sortie. Un candidat potentiel pour une classe décidable serait les architectures dans lesquelles il n'est pas possible de router toute l'information jusqu'aux processus en sortie, mais dans lesquelles on peut effectuer le calcul demandé par la spécification plus tôt, puis router le résultat jusqu'à la sortie. Par ailleurs, on a établi la complexité du problème consistant à vérifier si une architecture est UWC. Pour les architectures bien connectées, dans le cas où il n'y a pas de délai, le problème se ramène à un problème de flot et peut donc être résolu en temps polynomial. Cependant, si on considère des processus ayant des délais non nuls, le problème devient plus compliqué, et il pourrait être intéressant d'établir sa complexité dans le cas général.

Pour les systèmes asynchrones, le modèle que l'on a mis en place semble assez prometteur. Tout d'abord, on n'a pour le moment mis en lumière aucun résultat d'indécidabilité avec ces hypothèses. Une prolongation naturelle et importante de ce travail serait de voir si l'on peut obtenir la décidabilité pour toutes les architectures. Avec cet objectif, les résultats exposés dans ce document constituent une première étape pour une résolution modulaire du problème, et il semble possible d'adapter la technique de démonstration consistant à se réduire au singleton à d'autres cas (comme les architectures totalement déconnectées, ou les architectures de type pipeline comme celle représentée sur la figure 4.1 page 104). Dans le cas contraire, i.e., s'il se trouve que l'on ne peut pas obtenir la décidabilité du problème dans tous les cas, une démonstration d'indécidabilité dans ce modèle apporterait des éléments de compréhension supplémentaires sur le problème de SSD asynchrone.

Par ailleurs, on s'est ici contenté de définir la logique AlocTL comme candidate à nos

spécifications acceptables. Ce n'est probablement pas le formalisme de spécification le plus puissant permettant d'obtenir le résultat. On pourrait donc étudier l'expressivité de cette logique d'une part, et d'autre part définir éventuellement d'autres logiques pour les spécifications acceptables.

De façon plus générale, ce travail peut également être prolongé par l'étude de la synthèse de systèmes distribués tolérants aux fautes. Jusqu'à présent, les processus considérés étaient supposés « infaillibles ». Dans les applications réelles, les processus peuvent disfonctionner. On s'intéresse donc à savoir si une certaine propriété est réalisable par un système distribué dans lequel les processus peuvent faillir à un moment donné. Ce travail pourrait utiliser comme point de départ les travaux sur le Model-checking de systèmes tolérants aux fautes.

Bibliographie

[ACLY00] Rudolf Ahlswede, Ning Cai, Shuo-Yen Robert Li, and Raymond W. Yeung. Network information flow. *IEEE Transactions on Information Theory*, 46(4) :1204–1216, 2000.

[ALW89] Martín Abadi, Leslie Lamport, and Pierre Wolper. Realizable and unrealizable specifications of reactive systems. In Giorgio Ausiello, Mariangiola Dezani-Ciancaglini, and Simona Ronchi Della Rocca, editors, *Proceedings of the 16th International Colloquium on Automata, Languages and Programming (ICALP'89)*, volume 372 of *Lecture Notes in Computer Science*, pages 1–17. Springer, 1989.

[AM94] Anuchit Anuchitankul and Zohar Manna. Realizability and synthesis of reactive modules. In David L. Dill, editor, *Proceedings of the 6th International Conference on Computer Aided Verification (CAV'94)*, volume 818 of *Lecture Notes in Computer Science*, pages 156–168, Stanford, California, USA, 1994. Springer.

[AVW03] André Arnold, Aymeric Vincent, and Igor Walukiewicz. Games for synthesis of controllers with partial observation. *Theoretical Computer Science*, 1(303) :7–34, 2003.

[BJ05] Julien Bernet and David Janin. Tree automata and discrete distributed games. In Maciej Liskiewicz and Rüdiger Reischuk, editors, *Proceedings of the 15th International Symposium on Fundamentals of Computation Theory (FCT'05)*, volume 3623 of *Lecture Notes in Computer Science*, pages 540–551. Springer, 2005.

[BJ06] Julien Bernet and David Janin. On distributed program specification and synthesis in architectures with cycles. In Elie Najm, Jean-François Pradat-Peyre, and Véronique Donzeau-Gouge, editors, *Proceedings of the 26th IFIP WG6.1 International Conference on Formal Techniques for Networked and Distributed Systems (FORTE'06)*, volume 4229 of *Lecture Notes in Computer Science*, pages 175–190. Springer, 2006.

[BL69] J. Richard Büchi and Lawrence H. Landweber. Solving sequential conditions by finite-state strategies. *Transactions of the American Mathematical Society*, 138 :295–311, 1969.

[CE81] Edmund M. Clarke and E. Allen Emerson. Design and synthesis of synchronization skeletons using branching time temporal logic. In *Proceedings of the IBM Workshop on Logics of Programs*, 1981.

[CGS09] Thomas Chatain, Paul Gastin, and Nathalie Sznajder. Natural specifications yield decidability for distributed synthesis of asynchronous systems. In *Proceedings of the 35th International Conference on Current Trends in Theory and Practice of Computer Science (SOFSEM'09)*, Lecture Notes in Computer Science. Springer, 2009.

[Chu63] Alonzo Church. Logic, arithmetics, and automata. In *Proceedings of the International Congress of Mathematicians*, pages 23–35, 1963.

[DG01] Volker Diekert and Paul Gastin. Local temporal logic is expressively complete for cograph dependence alphabets. In Robert Nieuwenhuis and Andrei Voronkov, editors, *Proceedings of the 8th International Conference on Logic for Programming, Artificial Intelligence, and Reasoning (LPAR'01)*, volume 2250 of *Lecture Notes in Computer Science*, pages 55–69. Springer, 2001.

[EC82] E. Allen Emerson and Edmund. M. Clarke. Using branching time temporal logic to synthesize synchronization skeletons. *Science of Computer Programming*, 2(3) :241–266, 1982.

[EH83] E. Allen Emerson and Joseph Y. Halpern. "Sometimes" and "Not Never" revisited : On branching versus linear time. In *Proceedings of the 10th Annual ACM Symposium on Principles of Programming Languages (POPL'83)*, pages 127–140. ACM Press, 1983.

[EJ99] E. Allen Emerson and Charanjit S. Jutla. The complexity of tree automata and logics of programs. *SIAM Journal on Computing*, 29(1) :132–158, 1999.

[FS05] Bernd Finkbeiner and Sven Schewe. Uniform distributed synthesis. In *Proceedings of the 20th IEEE Annual Symposium on Logic in Computer Science (LICS'05)*, pages 321–330. IEEE Computer Society Press, 2005.

[FS06] Bernd Finkbeiner and Sven Schewe. Synthesis of asynchronous systems. In Germán Puebla, editor, *Proceedings of the International Symposium on Logic-based Program Synthesis and Transformation (LOPSTR'06)*, volume 4407 of *Lecture Notes in Computer Science*, pages 127–142. Springer, 2006.

[Gab87] Dov M. Gabbay. The declarative past and imperative future : Executable temporal logic for interactive systems. In Behnam Banieqbal, Howard Barringer, and Amir Pnueli, editors, *Temporal Logic in Specification, Proceedings*, volume 398 of *Lecture Notes in Computer Science*, pages 409–448. Springer, 1987.

[Gas90] Paul Gastin. Infinite traces. In Irène Guessarian, editor, *Semantics of Systems of Concurrent Processes, Proceedings of LITP Spring School on Theoretical Computer Science*, volume 469 of *Lecture Notes in Computer Science*, pages 277–308. Springer, 1990.

[GK07] Paul Gastin and Dietrich Kuske. Uniform satisfiability in PSPACE for local temporal logics over Mazurkiewicz traces. *Fundamenta Informaticae*, 80(1-3) :169–197, 2007.

[GLZ04] Paul Gastin, Benjamin Lerman, and Marc Zeitoun. Distributed games with causal memory are decidable for series-parallel systems. In Kamal Lodaya and Meena Mahajan, editors, *Proceedings of the 24th Conference on Foundations of Software Technology and Theoretical Computer Science (FSTTCS'04)*, volume 3328 of *Lecture Notes in Computer Science*, pages 275–286. Springer, 2004.

[GP95] Paul Gastin and Antoine Petit. *The book of traces*, chapter 11 : Infinite Traces. World Scientific, 1995.

[GPSS80] Dov M. Gabbay, Amir Pnueli, Saharon Shelah, and Jonathan Stavi. On the temporal basis of fairness. In *Proceedings of the 7th Annual ACM Symposium on Principles of Programming Languages (POPL'80)*, pages 163–173. ACM Press, 1980.

[GSZ06] Paul Gastin, Nathalie Sznajder, and Marc Zeitoun. Distributed synthesis for well-connected architectures. In Naveen Garg and S. Arun-Kumar, editors, *Proceedings of the 26th Conference on Foundations of Software Technology and Theoretical Computer Science (FSTTCS'06)*, volume 4337 of *Lecture Notes in Computer Science*, pages 321–332. Springer, 2006.

[GSZ09] Paul Gastin, Nathalie Sznajder, and Marc Zeitoun. Distributed synthesis for well-connected architectures. *Formal Methods in System Design*, 34(3) :215–237, 2009.

[Kam68] H. W. Kamp. *Tense Logic and the Theory of Linear Order*. Ph.d. thesis, University of California, Los Angeles, 1968.

[KMTV00] Orna Kupferman, P. Madhusudan, P. S. Thiagarajan, and Moshe Y. Vardi. Open systems in reactive environments : Control and synthesis. In Catuscia Palamidessi, editor, *Proceedings of the 11th International Conference on Concurrency Theory (CONCUR'00)*, volume 1877 of *Lecture Notes in Computer Science*, pages 92–107. Springer, 2000.

[Koz83] Dexter Kozen. Results on the propositional mu-calculus. *Theoretical Computer Science*, 27 :333–354, 1983.

[KV97] Orna Kupferman and Moshe Y. Vardi. Synthesis with incomplete information. In *Proceedings of the 2nd International Conference on Temporal Logic (ICTL'97)*, pages 91–106, 1997.

[KV99] Orna Kupferman and Moshe Y. Vardi. Church's problem revisited. *The Bulletin of Symbolic Logic*, 5(2) :245–263, 1999.

[KV00] Orna Kupferman and Moshe Y. Vardi. μ-calculus synthesis. In Mogens Nielsen and Branislav Rovan, editors, *Proceedings of the 25th International Symposium on Mathematical Foundations of Computer Science (MFCS'00)*, volume 1893 of *Lecture Notes in Computer Science*, pages 497–507. Springer, 2000.

[KV01] Orna Kupferman and Moshe Y. Vardi. Synthesizing distributed systems. In Joseph Y. Halpern, editor, *Proceedings of the 16th IEEE Annual Symposium on Logic in Computer Science (LICS'01)*. IEEE Computer Society Press, 2001.

[Ler05] Benjamin Lerman. *Vérification et Spécification des Systèmes Distribués*. PhD thesis, Université Denis Diderot - Paris VII, Paris, France, 2005. http://tel.archives-ouvertes.fr/tel-00322322/fr/.

[LT89] Nancy Lynch and Mark R. Tuttle. An introduction to input/output automata. *CWI-Quarterly*, 2(3) :219–246, 1989.

[Maz77] Antoni Mazurkiewicz. Concurrent program schemes and their interpretations. DAIMI report PB 78, Aarhus University, 1977.

[Maz86] Antoni W. Mazurkiewicz. Trace theory. In Wilfried Brauer, Wolfgang Reisig, and Grzegorz Rozenberg, editors, *Petri Nets : Central Models and Their Properties, Advances in Petri Nets 1986, Part II, Proceedings of an Advanced Course, Bad Honnef, 8.-19. September 1986*, volume 255 of *Lecture Notes in Computer Science*, pages 279–324. Springer, 1986.

[MS87] David E. Muller and Paul E. Schupp. Alternating automata on infinite trees. *Theoretical Computer Science*, 54 :267–276, 1987.

[MS95] David E. Muller and Paul E. Schupp. Simulating alternating tree automata by nondeterministic automata : New results and new proofs of theorems of Rabin, McNaughton and Safra. *Theoretical Computer Science*, 141(1&2) :69–107, 1995.

[MT01] P. Madhusudan and P. S. Thiagarajan. Distributed controller synthesis for local specifications. In Fernando Orejas, Paul G. Spirakis, and Jan van Leeuwen, editors, *Proceedings of the 28th International Colloquium on Automata, Languages and Programming (ICALP'01)*, volume 2076 of *Lecture Notes in Computer Science*, pages 396–407. Springer, 2001.

[MT02a] P. Madhusudan and P. S. Thiagarajan. Branching time controllers for discrete event systems. *Theoretical Computer Science*, 274(1-2) :117–149, 2002. Concurrency theory (Nice, 1998).

[MT02b] P. Madhusudan and P. S. Thiagarajan. A decidable class of asynchronous distributed controllers. In Lubos Brim, Petr Jancar, Mojmír Kretínský, and Antonín Kucera, editors, *Proceedings of the 13th International Conference on Concurrency Theory (CONCUR'02)*, volume 2421 of *Lecture Notes in Computer Science*, pages 145–160. Springer, 2002.

[MTY05] P. Madhusudan, P. S. Thiagarajan, and Shaofa Yang. The MSO theory of connectedly communicating processes. In Ramaswamy Ramanujam and Sandeep Sen, editors, *Proceedings of the 25th Conference on Foundations of Software Technology and Theoretical Computer Science (FSTTCS'05)*, volume 3821 of *Lecture Notes in Computer Science*, pages 201–212. Springer, 2005.

[MW80] Zohar Manna and Richard J. Waldinger. A deductive approach to program synthesis. *Association for Computing Machinery Transactions on Programming Languages and Systems*, 2(1) :90–121, 1980.

[MW84] Zohar Manna and Pierre Wolper. Synthesis of communicating processes from temporal logic specifcations. *ACM Transactions on Programming Languages and Systems*, 6(1) :68–93, 1984.

[MW03] Swarup Mohalik and Igor Walukiewicz. Distributed games. In Paritosh K. Pandya and Jaikumar Radhakrishnan, editors, *Proceedings of the 23rd Conference on Foundations of Software Technology and Theoretical Computer Science (FSTTCS'03)*, volume 2914 of *Lecture Notes in Computer Science*, pages 338–351. Springer, 2003.

[MWZ09] Anca Muscholl, Igor Walukiewicz, and Marc Zeitoun. A look at the control of asynchronous automata. In Kamal Lodaya, Madhavan Mukund, and Ramanujam R., editors, *Perspectives in Concurrency Theory*, pages 356–371. Universities Press, 2009.

[Pnu77] Amir Pnueli. The temporal logic of programs. In *Proceedings of the 18th Annual IEEE Symposium on Foundations of Computer Science (FOCS'77)*, pages 46–57. IEEE Computer Society Press, 1977.

[PR79] Gary L. Peterson and John H. Reif. Multiple-person alternation. In *Proceedings of the 20th Annual IEEE Symposium on Foundations of Computer Science (FOCS'79)*, pages 348–363. IEEE Computer Society Press, 1979.

[PR89a] Amir Pnueli and Roni Rosner. On the synthesis of a reactive module. In *Proceedings of the 16th Annual ACM Symposium on Principles of Programming Languages (POPL'89)*, pages 179–190. ACM, 1989.

[PR89b] Amir Pnueli and Roni Rosner. On the synthesis of an asynchronous reactive module. In Giorgio Ausiello, Mariangiola Dezani-Ciancaglini, and Simona Ronchi Della Rocca, editors, *Proceedings of the 16th International Colloquium on Automata, Languages and Programming (ICALP'89)*, volume 372 of *Lecture Notes in Computer Science*, pages 652–671. Springer, 1989.

[PR90] Amir Pnueli and Roni Rosner. Distributed reactive systems are hard to synthesize. In *Proceedings of the 31st Annual IEEE Symposium on Foundations of Computer Science (FOCS'90)*, volume II, pages 746–757. IEEE Computer Society Press, 1990.

[Pri67] Arthur Prior. *Past, Present and Future*. Oxford University Press, 1967.

[Rab70] Michael Oser Rabin. Weakly definable relations and special automata. In Y. Bar-Hillel, editor, *Proceedings of the Symposium of Mathematical Logic and Foundations of Set Theory*, pages 1–23, 1970.

[Rab72] Michael Oser Rabin. *Automata on Infinite Objects and Church's Problem*. American Mathematical Society, 1972.

[RD95] Grzegorz Rozenberg and Volker Diekert, editors. *Book of Traces*. World Scientific, Singapore, 1995.

[Ric53] Henry Gordon Rice. Classes of recursively enumerable sets and their decision problems. *Transactions of the American Mathematical Society*, 74(2) :358–366, 1953.

[RLL04] April Rasala Lehman and Eric Lehman. Complexity classification of network information flow problems. In J. Ian Munro, editor, *Proceedings of the 15th ACM-SIAM Symposium on Discrete Algorithms (SODA'04)*, pages 142–150. Society for Industrial and Applied Mathematics, 2004.

[Ros92] Roni Rosner. *Modular Synthesis of Reactive Systems*. PhD thesis, Weizmann Institute of Science, Rehovot, Israel, 1992.

[RW89] P. J. G. Ramadge and W. M. Wonham. The control of discrete event systems. In *Proceedings of the IEEE*, volume 77, pages 81–98. IEEE Press, 1989.

[Tho90] Wolfgang Thomas. Automata on infinite objects. In Jan van Leeuwen, editor, *Handbook of Theoretical Computer Science : Volume B : Formal Models and Semantics*, pages 133–191. Elsevier, 1990.

[Var95] Moshe Y. Vardi. An automata-theoretic approach to fair realizability and synthesis. In Pierre Wolper, editor, *Proceedings of the 7th International Conference on Computer Aided Verification (CAV'95)*, volume 939 of *Lecture Notes in Computer Science*, pages 267–278. Springer, 1995.

[vdMV98] Ron van der Meyden and Moshe Y. Vardi. Synthesis from knowledge-based specifications. In Davide Sangiorgi and Robert de Simone, editors, *Proceedings of the 9th International Conference on Concurrency Theory (CONCUR'98)*, volume 1466 of *Lecture Notes in Computer Science*, pages 34–49. Springer, 1998.

[vdMW05] Ron van der Meyden and Thomas Wilke. Synthesis of distributed systems from knowledge-based specifications. In Martín Abadi and Luca de Alfaro, editors, *Proceedings of the 16th International Conference on Concurrency Theory (CONCUR'05)*, volume 3653 of *Lecture Notes in Computer Science*, pages 562–576. Springer, 2005.

[VS85] Moshe Y. Vardi and Larry J. Stockmeyer. Improved upper and lower bounds for modal logics of programs : Preliminary report. In *Proceedings of the 7th Annual ACM Symposium on Theory of Computing (STOC'85)*, pages 240–251. ACM Press, 1985.

[WTD91] Howard Wong-Toi and David L. Dill. Synthesizing processes and schedulers from temporal specifications. In Edmund M. Clarke and Robert P. Kurshan, editors, *Proceedings of the 2nd International Conference on Computer Aided Verification (CAV'90)*, volume 531 of *Lecture Notes in Computer Science*, pages 272–281. Springer, 1991.

[Zie87] Wieslaw Zielonka. Notes on finite asynchronous automata. *ITA*, 21(2) :99–135, 1987.

[Zie95] Wieslaw Zielonka. *The book of traces*, chapter 7 : Asynchronous Automata. World Scientific, 1995.