Protocoles pour le rendez-vous et l'équité

Xavier Pandolfi

THESE

présentée par

Xavier PANDOLFI

pour obtenir le titre de DOCTEUR
de l'INSTITUT NATIONAL POLYTECHNIQUE DE GRENOBLE
(Arrêté ministériel du 23 novembre 1988)

Spécialité: Informatique

Protocoles pour le Rendez-Vous et l'Equité

Thèse soutenue le 14 Avril 1992 devant le jury:

J.-P. Verjus	président
G.-R. Perrin M. Raynal	rapporteurs
B. Plateau Ph. Jorrand	examinateurs

Thèse préparée au sein du Laboratoire d'Informatique Fondamentale et d'Intelligence Artificielle (LIFIA).

THESE

présentée par

Xavier PANDOLFI

pour obtenir le titre de DOCTEUR
de l'INSTITUT NATIONAL POLYTECHNIQUE DE GRENOBLE
(Arrêté ministériel du 23 novembre 1988)

Spécialité: Informatique

Protocoles pour le Rendez-Vous et l'Equité

Thèse soutenue le 14 Avril 1992 devant le jury:

J.-P. Verjus	président
G.-R. Perrin M. Raynal	rapporteurs
B. Plateau Ph. Jorrand	examinateurs

Thèse préparée au sein du Laboratoire d'Informatique Fondamentale et d'Intelligence Artificielle (LIFIA).

Résumé

Nous étudions la mise en œuvre du rendez-vous multiprocessus et de l'équité dans les langages du type "CSP généralisé".

Nous proposons une méthode de construction des protocoles. Cette méthode consiste à mettre en œuvre ces protocoles à partir d'un schéma de protocole, c'est-à-dire un protocole comportant des parties abstraites. La mise en œuvre effective d'un protocole se fait en remplaçant les parties abstraites par du code. Nous discutons le choix de ce code à l'aide d'exemples tirés de la littérature.

Nous étudions six notions d'équité dites classiques. Nous montrons que parmi ces six notions d'équité seules les notions d'équité dites fortes accroissent la vivacité. Nous montrons aussi qu'en général il est impossible de construire un protocole réalisant une de ces notions d'équité fortes : seule la "Strong Process Fairness" peut l'être, et cela seulement lorsque le rendez-vous est binaire. Nous étudions la construction des protocoles réalisant la "Weak Interaction Fairness" et la "Strong Process Fairness" (avec rendez-vous binaire). Nous dérivons, à partir d'une spécification, un schéma pour chacune de ces notions d'équité.

Abstract

We study the implementation of multiparty rendezvous and fairness for "generalized CSP-like" languages.

A method for constructing protocols is proposed. It is based on the use of protocol schemas, i.e. protocols containing abstract sections. A schema may be instantiated for obtaining a correct protocol by appropriately substituting abstract sections by effective code. Examples are given of such code for classical protocols.

We study six commonly used notions of fairness. We show that from these six notions, only strong notions of fairness are liveness enhancing. We also show that, among these strong notions of fairness, only "Strong Process Fairness" with binary rendezvous can have a distributed implementation. We study the design of protocols for "Weak Interaction Fairness" and "Strong Process Fairness" (binary rendezvous). We derive from a specification a schema for each of these notions of fairness.

A Edwige,

Benoît,

et Olivier.

Remerciements

Je remercie les membres du jury :

- Monsieur Jean-Pierre Verjus, Professeur à l'Institut National Polytechnique de Grenoble, qui a accepté de présider ce jury,
- Monsieur Michel Raynal, Professeur à l'Université de Rennes, et Monsieur Guy-René Perrin, Professeur à l'Université de Franche-Comté, qui ont accepté d'être rapporteurs,
- Madame Brigitte Plateau, Professeur à l'Institut National Polytechnique de Grenoble, qui a bien voulu s'intéresser à ce travail et participer au jury,
- Monsieur Philippe Jorrand, Directeur de Recherche au CNRS, qui m'a accueilli dans son équipe et qui a dirigé ce travail.

Je suis très redevable aux membres de l'équipe "Parallélisme" qui ont debuggé les premières versions de cette thèse, et plus particulièrement à Philippe Schnœbelen pour les nombreuses discussions que nous avons eues ensemble.

Xavier Pandolfi

Sommaire

Chapitre 1

Introduction

L'utilisation du parallélisme offre en théorie la possibilité d'accroître d'une manière significative les performances des ordinateurs. Toutefois, l'exploitation effective du parallélisme s'est avérée très difficile. Alors que dans le domaine du calcul numérique des gains en performance impressionnants ont été obtenus, les applications symboliques, avec leur structure plus irrégulière, n'ont pas connu un tel progrès. La plus importante des difficultés semble résider dans la programmation de ces applications sur des machines parallèles. Nous ne savons pas encore vraiment raisonner correctement sur les programmes parallèles, et on admet communément qu'une révolution conceptuelle est nécessaire dans notre façon d'aborder la programmation si l'on veut un jour profiter pleinement de la puissance des machines parallèles. Il faut développer des modèles, des concepts, des notations, des théories, des langages, etc. adaptés à la programmation parallèle.

Les mécanismes d'abstraction se sont révélés être des outils de programmation très utiles et ont été incorporés dans les langages de programmation modernes. Les principales abstractions proposées pour la programmation des machines séquentielles sont les abstractions des données (e.g. les types abstraits) et les abstractions du contrôle séquentiel (e.g. les boucles, les si-alors-sinon, les procédures).

L'architecture des machines et des réseaux encourage les programmeurs à voir les systèmes distribués comme des ensembles de processus séquentiels qui commu-

niquent et se coordonnent grâce à des échanges de messages point-à-point (one-to-one). Dès lors, les détails de la mémorisation (buffering) des messages, de la résolution des conflits, ou de la prévention des blocages (deadlock), par exemple, doivent souvent être pris en compte très tôt dans la conception des programmes parallèles. D'où le besoin de mécanismes d'abstraction pour le contrôle du parallélisme, la communication et la synchronisation.

1.1 Le rendez-vous

Dans un article célèbre, C.A.R. Hoare [Hoa78] proposa d'utiliser un modèle (le modèle CSP, pour "processus séquentiels communicants") où les calculs parallèles sont décrits en termes de modules "parallèles" indépendants qui coopèrent par le biais d'échanges explicites de messages. Hoare proposa l'utilisation du *rendez-vous* comme mécanisme d'abstraction de la communication et de la synchronisation. Ce mécanisme garantit que, quand un message est envoyé, il est reçu au même moment (conceptuellement). Ceci apporte une simplification énorme dans la conception et la vérification des programmes : il est inutile de raisonner sur des configurations où un message est "en route".

Ainsi, pour prouver qu'une collection de processus atteint un but commun, il est souvent nécessaire pour un processus de faire des assertions sur l'état des autres processus. Les processus "apprennent" l'état des autres processus en échangeant des messages. Non seulement la réception d'un message cause le transfert d'une valeur de l'émetteur vers le récepteur, mais aussi elle permet le "transfert" d'un prédicat [CM86]. Cela permet au récepteur de faire des assertions sur l'état de l'émetteur lors de l'émission du message, par exemple au sujet du progrès de l'émetteur dans ses calculs. Il est clair que les actions de l'émetteur, ultérieures à l'émission du message, peuvent invalider ces assertions. Lorsque le rendez-vous est utilisé, un échange de message représente un point de synchronisation à la fois dans l'exécution de l'émetteur et dans celle du récepteur. De ce fait, le message reçu correspond toujours à l'état courant de l'émetteur. De plus, quand l'échange se termine, l'émetteur peut faire des assertions sur l'état du récepteur.

Toutefois les communications point-à-point (one-to-one) ne sont pas toujours suffisamment abstraites. Par exemple, lorsque les processus coopèrent sur le mode client/serveur il est fréquent que plusieurs clients partagent un même serveur (many-to-one). On peut aussi avoir plusieurs serveurs identiques et un client qui ne se soucie pas de savoir lequel de ces serveurs traitera sa requête (one-to-many), ou même avoir plusieurs serveurs et plusieurs clients (many-to-many).

Récemment, plusieurs mécanismes d'interaction qui généralisent le *rendez-vous binaire* (one-to-one) à la CSP en un *rendez-vous multiprocessus* (many-to-many) ont été proposés (e.g. [FH83], [Cha87], [EFK89], [JS91]). Ces mécanismes ont en commun le fait qu'ils permettent à un nombre *arbitraire* de processus asynchrones

de se "retrouver" (synchroniser), d'établir temporairement un état global à ces processus, d'effectuer un certain nombre d'actions sur cet état global, puis de se séparer pour continuer indépendamment leur exécution. Nous appelons *interaction* un tel événement de communication et de synchronisation.

1.2 Expression et contrôle du non-déterminisme

Un processus peut vouloir effectuer une interaction parmi un groupe d'interactions, plutôt qu'une interaction fixée par avance. Comme on ne peut prévoir quelle sera l'interaction exécutée, ce comportement est *non-déterministe*. Dans certains cas il est utile de pouvoir contrôler dynamiquement le groupe des interactions désirées. Par exemple, un processus `buffer` peut accepter la demande d'un processus `producteur` de stocker un objet dans un buffer chaque fois que ledit buffer n'est pas plein, et il peut accepter la demande d'un processus `consommateur` de retirer un objet du buffer chaque fois que le buffer n'est pas vide. Pour programmer un tel comportement, et plus généralement pour exprimer et contrôler le non-déterminisme, il faut une notation.

La plupart des langages de programmation proposant le rendez-vous comportent une construction basée sur les *commandes gardées* introduites par Dijkstra comme structure de contrôle séquentiel [Dij75]. Une telle construction consiste généralement en une liste de commandes gardées de la forme :

$$garde \longrightarrow instructions$$

La garde se compose d'une expression booléenne et d'un "pattern d'interaction". La garde *réussit* si l'expression booléenne est vraie et si l'interaction peut avoir lieu sans délai; la garde *échoue* si l'expression booléenne est fausse; ou la garde est *suspendue* si l'expression booléenne est vraie et l'interaction ne peut avoir lieu immédiatement. La construction dans son ensemble est exécutée de la manière suivante : si toutes les gardes échouent, la construction échoue; si il y a une ou plusieurs gardes qui réussissent, une de ces gardes est choisie, l'interaction de la garde est exécutée, puis les instructions suivant la garde sont exécutées; ou sinon l'exécution est différée jusqu'à ce qu'une garde réussisse [AS83].

Dans la plupart des langages le choix de la garde exécutée n'est pas *équitable*. Dans le modèle CSP, par exemple, si plusieurs gardes réussissent, une d'entre elles est choisie d'une manière non-déterministe, mais aucune hypothèse ne peut être faite sur la garde sélectionnée. Ainsi au cours d'une exécution répétée de la commande alternative[1], une même garde pourra être choisie à chaque fois, même si d'autres gardes réussissent.

Il est quelquefois indispensable de contrôler le choix de la garde exécutée, c'est-à-dire de réduire le non-déterminisme dans ce choix. Cela est illustré par le programme CSP suivant qui contrôle la position d'un point sur un écran [Ber80]

[1] c'est le nom de la construction proposée en CSP pour exprimer et contrôler le non-déterminisme

```
Update ::  *[ calcul nouvelles valeurs de x et y; Display!(x, y) ]
Display ::  *[ Update?(u, v) ⟶ skip □ Screen!(u, v) ⟶ skip ]
Screen ::  *[ Display?(w, z); rafraîchissement écran avec (w, z) ]
```

où le processus `Update` calcule périodiquement les coordonnées d'un point et les communique à `Display`. Le processus `Display` mémorise ces coordonnées lorsqu'il les reçoit, et les fournit au processus `Screen` quand celui-ci les réclame. Le processus `Screen` demande périodiquement les coordonnées du point et rafraîchit l'écran en conséquence. Pour que ce programme fasse ce que l'on attend, il faut pouvoir contrôler le choix des gardes du processus `Display`. En effet, si la première garde n'est jamais choisie le processus `Update` ne pourra communiquer les nouvelles coordonnées, et si la seconde garde n'est jamais choisie le processus `Screen` ne sera jamais informé des nouvelles coordonnées.

Différentes *notions d'équité* pour le choix des gardes, lorsque le mécanisme d'interaction est basé sur le rendez-vous, ont été proposées et étudiées (e.g. [AFK87], [BKS88b], [AFG90], [AFL90]).

1.3 Le problème de l'implémentation du rendez-vous

Proposer une construction pour un langage de programmation nécessite de savoir l'implémenter. L'implémentation du rendez-vous est aisée quand les patterns d'interaction ne peuvent pas apparaître dans les gardes d'une construction non-déterministe.

En revanche, l'implémentation du rendez-vous est un problème non trivial lorsque des patterns d'interaction peuvent apparaître dans les gardes. En effet, au problème de la synchronisation des processus participant à l'interaction exécutée vient s'ajouter le problème de la coordination des processus pour le choix de cette interaction. Il s'agit en fait d'un problème d'exclusion : un processus ne peut choisir qu'une seule garde parmi celles qui réussissent et donc ne peut participer qu'à une seule des interactions présentes dans ces gardes. La principale difficulté dans la réalisation de cette exclusion est due au fait que les processus participant à une interaction ont un rôle symétrique dans le choix de celle-ci.

En brisant cette symétrie, nous pouvons obtenir des protocoles pour le rendez-vous particulièrement efficaces [Sil79]. C'est le cas, par exemple, lorsqu'on garantit que parmi les processus qui interagissent il y en a toujours un seul qui exécute une construction non-déterministe, ou en d'autres termes, un seul processus doit effectuer un choix parmi plusieurs interactions possibles. L'algorithme d'implémentation est alors très simple : c'est ce processus particulier qui fait le choix de l'interaction qui sera effectuée et qui impose aux autres sa décision. C'est cette approche maître/esclaves qui a été adoptée lors de la définition de CSP ou OCCAM [May83] : seules les commandes de réception de message sont autorisées dans les gardes des commandes alternatives. Ceci assure que lorsqu'une paire de commandes

de communication correspondent (match), une au moins est inconditionnelle : la commande d'envoi du message, et c'est donc le processus récepteur qui impose son choix aux processus émetteurs.
Diverses autres restrictions permettant une implémentation efficace ont été proposées. Par exemple A. Silberschatz a proposé dans [Sil79] une implémentation des commandes alternatives de CSP généralisées[2] reposant sur l'utilisation d'une relation maître/esclaves entre les processus fournie par le programmeur. K. Mitchell [Mit86] a décrit une méthode (synchronising annotations) permettant de calculer une telle relation de dominance pour des programmes CCS [Mil80].

Certains problèmes s'expriment plus "naturellement" lorsque les processus ont un rôle symétrique dans le choix des interactions (voir [Ber80], [Bor83], [Bou88]). Mais, dans ce cadre, l'implémentation du rendez-vous n'est pas triviale. En particulier, F. de Boer et C. Palamidessi [dBP91] ont montré récemment qu'on ne peut compiler séparément les gardes d'une structure de contrôle non-déterministe quand les processus ont un rôle symétrique : seule une implémentation "globale" de la structure de contrôle est possible.

Le premier protocole pour le rendez-vous (binaire) ne reposant pas sur une hypothèse d'asymétrie dans les processus a été proposé par A.J. Bernstein [Ber80]. Depuis, de nouveaux protocoles pour le rendez-vous binaire sont régulièrement proposés (voir Bibliographie). Certains de ces protocoles ont été utilisés pour l'implémentation de langages de programmation (e.g. Joyce [BH87], Pascal-m [AB83]).

Plus récemment des protocoles pour le rendez-vous multiprocessus ont été proposés (voir Bibliographie) pour implémenter Action Systems [BKS88a] (e.g. [BKS84]), Shared Actions [MR87] (e.g. [Ram87a]), RADDLE [For86] (e.g. [CM88]) ou LOTOS [BB87] (e.g. [vBGW89]) par exemple.

La notion d'équité réalisée par la majorité des protocoles est la "Weak Interaction Fairness" ($\mathcal{WIF}$) (e.g. [BS83]) : si une interaction est exécutable à un instant donné alors elle ne le sera plus au bout d'un temps fini (par le fait de son exécution, ou de l'exécution d'une autre interaction ayant un participant commun). Quelques protocoles ont été proposés pour réaliser des notions d'équité plus fortes (plus restrictives) (e.g. [Sis84], [AFL90], [PK90], [BT91]).

1.4 Notre contribution

Bien que le nombre de protocoles publiés soit important, leur description est souvent obscure et leurs propriétés sont trop souvent liées à un mécanisme de communication d'un langage particulier. En effet, peu d'auteurs ont pris le soin de décrire

[2] commandes alternatives permettant à la fois des commandes de réception et des commandes d'envoi de message dans leurs gardes

les principes de construction des protocoles qu'ils ont proposés. En fait les seuls auteurs qui à notre connaissance ont détaillé la construction de leur protocole sont K. Chandy et J. Misra [CM88]. Il est très souvent difficile de retrouver les principes de construction d'un protocole, et il est encore plus difficile de savoir quels choix ont été effectués et d'établir leurs conséquences sur les propriétés du protocole.

Nous proposons au chapitre 4 une méthode de construction des protocoles pour le rendez-vous multiprocessus. Cette méthode consiste simplement à mettre en œuvre ces protocoles à partir d'un schéma donné. Plus précisément nous dérivons, à partir d'une spécification du problème, un *schéma de protocole*, c'est-à-dire un protocole comportant des parties abstraites. La mise en œuvre effective d'un protocole se fait simplement en remplaçant les parties abstraites par du code. Nous discutons le choix de ce code à l'aide d'exemples tirés de la littérature.

La mise en œuvre de protocoles à partir d'un schéma comporte plusieurs avantages. Le premier, et non le moindre, est que l'on ne part pas de zéro. Le schéma sert de guide dans la construction du protocole. Il représente la connaissance et l'expérience acquise dans la construction des protocoles.

Un second avantage est que la preuve de correction d'un schéma vis-à-vis de la spécification constitue une "preuve générique" pour tous les protocoles construits à partir de ce schéma [BK88].

Notre méthode permet de construire aisément de nombreux protocoles nouveaux. Mais elle permet aussi de retrouver et de comprendre les protocoles déjà publiés. Nous pensons qu'elle peut même servir de base pour une taxonomie (à la manière de [Tho88]) de ces protocoles.

La valeur pratique d'une notion d'équité peut être jugée en termes de sa contribution dans le développement et la preuve d'algorithmes, mais aussi en termes du coût de son implémentation. Nous étudions dans le chapitre 2 six notions d'équité dites "classiques". L'utilisation d'une notion d'équité ne se justifie que si elle permet un accroissement de la vivacité (liveness) de certains programmes. Nous montrons que parmi ces six notions d'équité seules les notions d'équité dites "fortes" accroissent la vivacité. Nous montrons aussi qu'en général il est impossible de construire un protocole réalisant une de ces notions d'équité fortes : seule la "Strong Process Fairness" ($\mathcal{SPF}$) peut l'être, et cela seulement lorsque le rendez-vous est binaire. Ces résultats sont essentiellement tirés de [AFK87].

Cela justifie notre choix d'étudier la construction des protocoles réalisant $\mathcal{WIF}$ et $\mathcal{SPF}$ (avec rendez-vous binaire). Nous proposons dans le chapitre 4 un schéma pour chacune de ces notions d'équité.

Pour spécifier le problème et décrire les schémas, nous avons utilisé la méthode *Transition-Axiom* de L. Lamport [LS85, Lam89]. Nous décrivons dans le chapitre 3 ces notations, ainsi que les techniques utilisées pour prouver la correction des

schémas vis-à-vis de la spécification.

A notre connaissance aucun travail de synthèse sur l'implémentation du rendez-vous n'a été publié à ce jour. Nous espérons contribuer ici à une meilleure compréhension de ces protocoles en termes des principes de construction et des techniques de mise en œuvre utilisés.

Ce travail a été entrepris dans le cadre du projet de recherche européen ESPRIT projet 415 "Parallel Architectures and Languages for Advanced Information Processing - a VLSI-directed approach" [KPB+90].

Chapitre 2

Notions d'équité pour des langages basés sur le rendez-vous

De nombreuses notions d'équité ont été proposées et étudiées à la fois du point de vue de la sémantique et du point de vue de la théorie des preuves (e.g. [Fra86]). Le choix d'une notion d'équité pour un modèle ou un langage repose en grande partie sur des critères subjectifs, implicites. Néanmoins trois critères sémantiques simples ont été proposés dans [AFK87]. Nous rappelons en Section 2.1 la définition d'une notion d'équité ainsi que les définitions de ces trois critères.

Différentes *notions d'équité* pour le choix des gardes, lorsque le mécanisme d'interaction est basé sur le rendez-vous, ont été proposées et étudiées (e.g. [AFK87], [BKS88b], [AFG90], [AFL90]). Nous étudions dans ce chapitre six notions d'équité "classiques" : trois notions d'équité "faibles" et trois notions d'équité "fortes". Elles sont définies en Section 2.3.

L'utilisation d'une notion d'équité ne se justifie que si elle permet un accroissement de la vivacité de certains programmes. Nous montrons en Section 2.3 que, parmi ces six notions d'équité, seules les notions d'équité fortes accroissent la vivacité. Nous montrons aussi qu'en général aucune de ces notions d'équité fortes n'est compatible avec l'équivalence sur les exécutions induite par l'ordre partiel sur les ac-

tions. Nous proposons en Section 2.4 des solutions pour que ce critère soit satisfait.

En Section 2.5 nous interprétons ces résultats en termes de la possibilité (ou de l'impossibilité) d'une implémentation distribuée de ces notions d'équité.

Les Sections 2.1 et 2.3 reprennent le contenu de [AFK87] à la lecture duquel nous renvoyons systématiquement pour plus de précisions.

Les exemples de programmes non-déterministes illustrant les résultats de ce chapitre sont écrits en IP (Interacting Processes) [Fra89], un mini-langage basé sur le rendez-vous multiprocessus. Ce mini-langage est défini en Section 2.2.

2.1 Les notions d'équité

2.1.1 Modèle

Un programme *distribué* est constitué par une collection de *processus*. Ces processus ont des états disjoints et exécutent des *actions* atomiques. Le modèle attribue chaque action soit à un seul processus, et dans ce cas nous dirons que cette action est *locale* au processus, soit à plusieurs processus, et dans ce cas nous dirons que c'est une action *conjointe* (joint action) ou une *interaction* de ces processus. L'ensemble des processus auxquels est attribuée une interaction particulière est appelé l'ensemble des *participants* à cette interaction.

Une *exécution* d'un programme est une séquence maximale de transitions

$$\sigma_0 \xrightarrow{\alpha_1} \sigma_1 \xrightarrow{\alpha_2} \sigma_2 \cdots$$

où les σ_i sont des états (globaux) du programme, les α_i des actions atomiques, et $\sigma_i \xrightarrow{\alpha_{i+1}} \sigma_{i+1}$ dénote une exécution de l'action α_{i+1} qui transforme l'état σ_i en σ_{i+1}.

Nous supposons aussi que l'état du programme détermine un prédicat *exécutable* (enabled) sur les actions :

Définition 2.1

1. *Une* action *est* exécutable *dans un état donné si elle peut servir de prochaine action exécutée.*
2. *Un* processus *est* exécutable *dans un état donné si une action qui lui est attribuée est exécutable dans cet état.*
3. *Un* processus *est* prêt *pour une action dans un état donné si son état local est la projection d'un état du programme dans lequel l'action est exécutable et l'action lui est attribuée.*
4. *Un* groupe de processus *est* exécutable *dans un état donné si il existe une interaction exécutable dans cet état qui a* exactement *ce groupe de processus pour participants.*

Lorsque le modèle n'admet que le rendez-vous binaire, les groupes (paires de processus) sont aussi appelés canaux (channels) [AFK87].

Nous supposerons qu'il existe, dans une exécution, un *ordre partiel* sur les actions (locales et conjointes) appelé *relation de causalité* (e.g. [Lam78]). Nous supposerons de plus que cet ordre est total sur les actions locales à chaque processus, i.e. deux actions locales à un même processus sont ordonnées.

Définition 2.2

1. *Deux actions atomiques sont* indépendantes *si elles ne sont pas comparables par l'ordre partiel.*
2. *Si π et ρ sont deux exécutions, alors $\pi \equiv \rho$ si et seulement si π peut être obtenue à partir de ρ en transposant (éventuellement une infinité de fois) simultanément deux actions indépendantes (i.e. π est une permutation de ρ qui respecte l'ordre partiel sur les actions).*

Nous appellerons *projection* d'une exécution π sur un processus p, notée $[\pi]_p$, le résultat de la destruction dans π de toutes les actions qui ne sont pas attribuées à p et la restriction des états aux seules variables utilisées par p. Notons qu'en général $[\pi]_p$ n'est pas une exécution.

Le lemme suivant nous sera utile dans la suite de ce chapitre. C'est une conséquence de notre hypothèse de totalité de la relation de causalité sur les actions locales à un processus.

Lemme 2.3 (Projections égales)
Si $\pi \equiv \rho$ alors pour chaque processus p, $[\pi]_p = [\rho]_p$

Nous supposerons une sémantique pour les programmes comportant à la fois l'ensemble des exécutions et la relation de causalité sur les actions.

Enfin nous supposerons que :

(1) *Chaque interaction est suivie immédiatement par un état dans lequel aucun de ses processus participants n'est prêt pour une interaction* (noninstantaneous readiness). C'est-à-dire que quand un processus a exécuté une interaction, il entre dans un état local où *aucune* des interactions auxquelles il peut participer n'est exécutable. Ce processus doit nécessairement exécuter une action locale avant de pouvoir à nouveau participer à une interaction.

Cette hypothèse simplifie la définition de la notion d'interaction *continuellement* exécutable. Elle est justifiée par le fait que l'exécution d'une interaction prend du temps au niveau implémentation même si au niveau programme elle

est considérée comme atomique. Ainsi l'exécution d'une interaction peut temporairement rendre non exécutables d'autres interactions. Cette hypothèse nous permet donc de confondre *continuellement* avec *sans interruption.*

(2) Choix uniforme : *Un choix entre une action locale et une interaction n'est jamais possible* (i.e. le cas ne se présente pas). Cette hypothèse associée à la précédente garantie que les notions d'équité que nous considérons ici ne sont pas sensibles à l'addition d'actions locales à un processus (e.g. *skip*).

(3) Progrès minimal [OL82] : *Chaque processus qui est dans un état où une action locale est exécutable exécutera une action au bout d'un temps fini.* Cette propriété de vivacité garantie que les processus ne "s'arrêteront" pas tant qu'ils pourront exécuter des actions locales.

Ces trois hypothèses garantissent que les interactions constitueront le sujet principal des notions d'équité. La propriété de progrès minimal servira de propriété de vivacité de référence pour évaluer le gain en vivacité obtenu grâce aux notions d'équité.

2.1.2 Définitions

Ainsi que nous l'avons noté dans l'introduction de ce chapitre, une notion d'équité sert à exclure un certain nombre d'exécutions (celles qui sont *inéquitables*) qui sinon auraient été légales.

Définition 2.4 (Notion d'équité) *Etant donné un programme (distribué)* $\mathbf{P}$, $\mathbf{comp(P)}$ *est l'ensemble des exécutions de* $\mathbf{P}$ *respectant la propriété de progrès minimal.*

Une notion d'équité $\mathcal{F}$ *est une règle permettant, pour tout programme* $\mathbf{P}$, *de choisir un sous-ensemble d'exécutions* $\mathcal{F}(\mathbf{P}) \subseteq \mathbf{comp(P)}$ *tel que* $\mathcal{F}(\mathbf{P})$ *contient toutes les exécutions finies de* $\mathbf{comp(P)}$.

Il est nécessaire que chaque programme ait au moins une exécution équitable. En effet, sans cette contrainte, aucun scheduler (qui doit produire une exécution équitable) ne pourrait traiter correctement l'équité. De plus, aucun scheduler raisonnable ne pouvant prédire à chaque instant les extensions possibles d'une exécution, il doit être possible d'étendre toute exécution partielle en une exécution équitable.

Définition 2.5 (Réalisabilité) $\mathcal{F}$ *est* réalisable *(feasible) ssi pour tout programme* $\mathbf{P}$ *chaque segment initial (préfixe) fini d'une exécution de* $\mathbf{comp(P)}$ *peut être étendu en une exécution de* $\mathcal{F}(\mathbf{P})$.

Les exécutions sont des séquences d'exécutions d'actions. Il est clair que l'ordre des exécutions d'actions indépendantes est arbitraire. Une notion d'équité devrait donc respecter l'équivalence $\equiv$ définie plus haut sur les exécutions. C'est-à-dire, si deux exécutions infinies sont équivalentes alors elles doivent être toutes les deux équitables ou toutes les deux inéquitables. Si ce critère n'est pas satisfait alors la notion d'équité dépend de la "vitesse" des processus ou de la position de l'observateur, ce qui n'est pas souhaitable.

Définition 2.6 (Compatibilité avec l'équivalence) $\mathcal{F}$ *est* compatible avec l'équivalence *(equivalence robust) ssi pour tout programme* $\mathbf{P}$ *et pour toute paire d'exécutions* π *et* ρ *de* $\mathbf{comp(P)}$

$$(\pi \in \mathcal{F}(\mathbf{P}) \wedge \pi \equiv \rho) \Rightarrow \rho \in \mathcal{F}(\mathbf{P})$$

Tous les modèles possèdent une *propriété de vivacité fondamentale* : ici la propriété de progrès minimal. L'ajout d'une notion d'équité ne se justifie que si elle permet un accroissement de la vivacité de certains programmes, c'est-à-dire s'il existe un programme qui possède une propriété de vivacité qu'il n'a pas sans cette contrainte supplémentaire (i.e. l'équité).

Il est suffisant de considérer ici l'impact des notions d'équité sur la terminaison. En effet il est bien connu que les autres propriétés de vivacité peuvent être réduites à la terminaison pour un programme dérivé [GFMdR85].

Définition 2.7 (Accroissement de la vivacité) $\mathcal{F}$ accroît la vivacité *(liveness enhancing) ssi il existe un programme* $\mathbf{P}$ *tel que* $\mathbf{comp(P)}$ *contient une exécution infinie, mais toutes les exécutions de* $\mathcal{F}(\mathbf{P})$ *sont finies.*

C'est-à-dire que $\mathbf{P}$ termine si $\mathcal{F}$.

2.2 Le langage IP (Interacting Processes)

Nous présentons dans cette section un mini-langage, appelé *IP (Interacting Processes)*, défini pour la première fois dans [Fra89]. Ce langage est une abstraction et une simplification des langages de programmation incluant des mécanismes basés sur le rendez-vous multiprocessus. IP nous servira dans ce chapitre à écrire des programmes non-déterministes.

En IP, un programme $P :: [\ ||_{i=1,n}\ P_i\ ]$ est composé de $n \geq 1$ (n fixé) processus ayant des états locaux *disjoints* (i.e. pas de variable partagée). Un processus P_i est constitué par une instruction S prenant une des formes suivantes :

- *skip* : instruction sans effet sur l'état du processus.

- *Affectation* $x := e$: où x est une variable locale à P_i et e est une expression sur l'état local de P_i. Les affectations ont leur sémantique habituelle.

- *Interaction* $I[\ \overline{v} := \overline{e}\ ]$: I est le *nom* de l'interaction et $[\ \overline{v} := \overline{e}\ ]$ est une affectation parallèle (optionnelle) constituant le *corps local* de l'interaction. Toutes les variables du n-uplet de variables $\overline{v}$ sont locales à P_i et distinctes deux à deux. Le n-uplet d'expressions $\overline{e}$ peut utiliser des variables locales aux autres processus participants à I et donc *non locales* à P_i. L'ensemble des *participants* à I est constitué par tous les processus qui font syntaxiquement référence à I. Quand un processus (durant une exécution) atteint une instruction d'interaction il est dit *prêt* pour cette interaction. Une interaction I est *exécutable* (enabled) seulement quand tous ses participants sont prêts pour elle. I peut être exécutée seulement si elle est exécutable. Une interaction synchronise donc tous ses participants. L'exécution de I consiste en l'exécution en parallèle des affectations de tous les corps de I locaux à ses participants. Les occurrences d'une variable dans la partie droite d'une affectation dans un corps local de I font référence à la valeur initiale de cette variable, i.e. à sa valeur à l'instant où l'exécution de I a débuté. Il n'y a pas de synchronisation sur la fin d'une interaction : lorsqu'un participant a terminé l'exécution du corps local d'une interaction il exécute le reste de ses instructions sans attendre la terminaison des exécutions des corps locaux aux autres participants. Notons que lorsque le corps d'une interaction local à un processus est vide (i.e. $I[\ ]$), l'exécution de l'interaction a pour seul effet de synchroniser ce processus avec les autres participants.

- *Composition séquentielle* $S_1\ ;\ S_2$: S_1 est d'abord exécutée. Lorsque cette exécution termine, S_2 est exécutée.

- *Sélection non-déterministe* $[\ \Box_{k=1,m}\ B_k;\ I_k[\ \overline{v}_k := \overline{e}_k\ ] \longrightarrow S_k\ ]$: Ici $B_k;\ I_k[\ \overline{v}_k := \overline{e}_k\ ]$ est une *garde* composée de deux parties. La partie B_k est une expression booléenne sur l'état local de P_i. La partie $I_k[\ \overline{v}_k := \overline{e}_k\ ]$ est une interaction. S_k peut être n'importe quelle instruction. Quand une sélection non-déterministe est évaluée dans un état donné, la k-ème garde est *ouverte* (open) si B_k est vraie dans cet état (notons que P_i est alors prêt pour l'interaction I_k dans cet état). Cette garde est *exécutable* (enabled) si et seulement si elle est ouverte et I_k est exécutable. L'exécution de cette instruction se fait en plusieurs étapes :

  1. évaluation de toutes les parties booléennes pour déterminer les gardes ouvertes.
  2. Si aucune garde n'est ouverte l'instruction *échoue* (fails). Sinon une garde exécutable est *franchie* (passed) (simultanément avec l'exécution de tous les autres corps de I_k locaux aux autres participants)
  3. puis S_k est exécutée.

Quand il existe des gardes ouvertes mais aucune garde exécutable, l'exécution est *bloquée* (blocked) jusqu'à ce qu'une garde ouverte devienne exécutable.

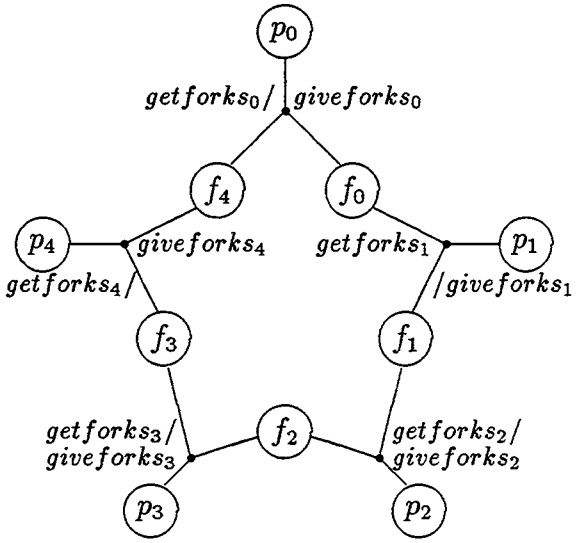
- *Itération non-déterministe* $*[\ \Box_{k=1,m}\ B_k;\ I_k[\ \overline{v}_k := \overline{e}_k\ ] \longrightarrow S_k\ ]$: Similaire à la sélection non-déterministe, mais l'exécution de l'instruction termine quand aucune garde n'est ouverte, et l'exécution de l'instruction est répétée après chaque exécution d'une commande gardée.

Deux sémantiques différentes pour IP sont proposées dans [AFG90] : une *sémantique sérialisée* (serialized semantics) et une *sémantique concurrente* (overlapping semantics). Dans la sémantique sérialisée une interaction est représentée par une seule action atomique qui transforme l'état du programme (suivant les affectations du corps de l'interaction) tandis que dans la sémantique concurrente une interaction est représentée par plusieurs transitions (ou *fragments d'interaction*), la première transformant l'état (c'est la seule qui est conjointe), les autres (locales aux participants) libérant les participants de l'interaction (i.e. un participant "exécute" l'interaction tant qu'il n'a pas été libéré et c'est seulement après avoir été libéré qu'il peut participer à une autre interaction). Ainsi les exécutions d'interactions indépendantes peuvent se chevaucher (overlap) dans la sémantique concurrente, tandis que dans la sémantique sérialisée les exécutions sont effectuées les unes après les autres (interleaved).

La sémantique concurrente donne manifestement un modèle plus réaliste que la sémantique sérialisée pour des exécutions de programmes IP sur des machines parallèles. La sémantique sérialisée, elle, semble plus facile à utiliser pour faire des preuves de correction [BKS88b]. C'est cette sémantique qui est choisie dans [Fra89] où un système de preuve pour la correction partielle des programmes IP est présenté.

Ces deux sémantiques satisfont les contraintes requises en Section 2.1.1. Nous avons décidé (arbitrairement) que la sémantique des programmes IP proposés dans ce chapitre sera la sémantique sérialisée. Toutefois il est important de noter que l'intérêt des exemples de programmes présentés ici n'est pas affecté par le choix de l'une ou l'autre de ces sémantiques.

Un exemple de programme IP est donné en Figure 2.1.



$$PHIL :: [\ p_0 \parallel \ldots \parallel p_4 \parallel f_0 \parallel \ldots \parallel f_4\ ]$$

where

$p_i :: s_i :=' t';$

$*[\ s_i =' t' \longrightarrow s_i :=' h'$

$\square$

$s_i =' h';\ getforks_i[s_i :=' e'] \longrightarrow giveforks_i[s_i :=' t']$

$]$

$f_i :: *[\ getforks_i[] \longrightarrow giveforks_i[]$

$\square$

$getforks_{(i+1)\mathrm{mod}5}[] \longrightarrow giveforks_{(i+1)\mathrm{mod}5}[]$

$]$

Figure 2.1: Une solution sans blocage du problème des philosophes mangeurs

Exemple: (philosophes mangeurs) Il s'agit d'une solution du problème bien connu des cinq philosophes mangeurs [Dij71]. Initialement, tous les philosophes p_i *pensent* (think) ($s_i =' t'$). Ils peuvent indépendamment devenir *affamés* (hungry) ($s_i =' h'$). Quand le k-ème philosophe p_k est affamé il peut *manger* (eat) ($s_k =' e'$) en interagissant avec les $(k-1)$-ème et k-ème fourchettes, $f_{(k-1)\mathrm{mod}5}$ et f_k, via l'interaction tripartite $getforks_k$ (voir dessin). Lorsque p_k a fini de manger, il redevient pensant ($s_k =' t'$) en interagissant avec les mêmes fourchettes via l'interaction $giveforks_k$. Il est clair que cette solution est sans blocage[1]. $\square$

[1]Les philosophes sont dits *bloqués* (deadlocked) quand aucun philosophe ne mange et aucun des philosophes affamés ne peut parvenir à manger.

2.3 Six notions d'équité classiques

2.3.1 Définitions

Les processus, les interactions, ou les collections d'interactions possibles pour un groupe de processus peuvent être les sujets des notions d'équité. Ainsi pour un langage basé sur le rendez-vous, il est raisonnable de définir une notion d'équité qui garantit qu'une action sera exécutée par chaque processus satisfaisant une certaine condition, ou bien que chaque interaction satisfaisant une condition donnée sera exécutée, ou bien que pour chaque groupe de processus satisfaisant une condition donnée, une interaction, parmi l'ensemble des interactions attribuées à ce groupe de processus, sera exécutée.

Une fois le sujet de la notion d'équité choisi, il faut déterminer la condition à réaliser. Deux types de conditions bien connus donnent lieu soit à des notions d'équité dites *faibles* pour lesquelles le choix doit être *continuellement* possible à partir d'un certain point dans l'exécution, soit à des notions d'équité dites *fortes* pour lesquelles le choix doit être *infiniment souvent* possible [Fra86].

En considérant toutes les paires sujet/condition possibles, on peut obtenir six notions d'équité : elles sont définies en Table 2.1.

Sujet \ *Condition*	*Faible*	*Forte*
Processus	$\mathcal{WPF}$: Au cours d'une exécution infinie chaque processus continuellement exécutable interagit au bout d'un temps fini.	$\mathcal{SPF}$: Au cours d'une exécution infinie chaque processus infiniment souvent exécutable interagit infiniment souvent.
Groupe	$\mathcal{WGF}$: Au cours d'une exécution infinie chaque groupe de processus continuellement exécutable interagit au bout d'un temps fini.	$\mathcal{SGF}$: Au cours d'une exécution infinie chaque groupe de processus infiniment souvent exécutable interagit infiniment souvent.
Interaction	$\mathcal{WIF}$: Au cours d'une exécution infinie chaque interaction continuellement exécutable est exécutée au bout d'un temps fini.	$\mathcal{SIF}$: Au cours d'une exécution infinie chaque interaction infiniment souvent exécutable est exécutée infiniment souvent.

Table 2.1: Six notions d'équité pour des langages basés sur le rendez-vous

Remarques sur les définitions en Table 2.1 :

1. Par définition, toutes les exécutions finies sont équitables quelle que soit la notion d'équité considérée.

2. Des commandes gardées imbriquées (nested) sont parfois considérées (ce n'est pas le cas dans cette thèse). Les notions d'équité définies ici ne font référence qu'aux interactions apparaissant dans les gardes au niveau le plus haut de ces commandes (top-level fairness). Et dorénavant, nous ne ferons référence qu'à ces interactions (top-level interactions). Nous renvoyons à [Fra86] pour plus de détails sur les notions d'équité prenant en compte tous les niveaux d'imbrication (all-levels fairness).

3. Une autre notion d'équité est parfois utilisée (e.g. [CM88], [Bag89a]) :

   Définition 2.8 (pseudo-$\mathcal{WIF}$) *Si, au cours d'une exécution infinie, une interaction I est exécutable alors au bout d'un temps fini un de ses participants exécutera une interaction (qui peut être différente de I).*

   En d'autres termes, si une interaction est exécutable alors au bout d'un temps fini elle ne le sera plus. On peut aisément prouver que *pseudo-*$\mathcal{WIF}$ et $\mathcal{WIF}$ sont équivalentes ([AFL90]) c'est-à-dire que pour tout programme **P**, *pseudo-*$\mathcal{WIF}(\mathbf{P}) = \mathcal{WIF}(\mathbf{P})$.

Quelques exemples font apparaître clairement l'intérêt de ces notions d'équité.

$$
\begin{array}{l}
\quad P :: [\ P_1 \ ||\ P_2 \ ||\ P_3\ ] \\
\textit{where} \\
\quad P_1 :: I[] \\
\\
\quad P_2 :: b_2 := \mathit{true}; \\
\qquad *[\ b_2;\ I[] \longrightarrow b_2 := \mathit{false} \\
\qquad\quad \square \\
\qquad\quad b_2;\ J[] \longrightarrow \mathit{skip} \\
\qquad\ ]; \\
\qquad K[] \\
\\
\quad P_3 :: b_3 := \mathit{true}; \\
\qquad *[\ b_3;\ J[] \longrightarrow \mathit{skip} \\
\qquad\quad \square \\
\qquad\quad b_3;\ K[] \longrightarrow b_3 := \mathit{false} \\
\qquad\ ]
\end{array}
$$

Figure 2.2: Processus négligé

Exemple: (Processus négligé) Dans ce programme (Figure 2.2) P_2 et P_3 peuvent interagir (via J) un nombre quelconque (éventuellement infini) de fois. Une

fois que P_1 et P_2 ont interagi (via I) les gardes sont fermées et tous les processus terminent au bout d'un temps fini (après exécution de K).

La terminaison du programme dépend donc ici de la possibilité pour P_1 d'interagir. Ainsi il suffit d'imposer une contrainte d'équité sur les processus ($\mathcal{SPF}$) pour que le programme termine toujours. □

Le programme de la Figure 2.3 termine avec une notion d'équité sur les groupes de processus mais pas avec une notion d'équité sur les processus.

$$
\begin{array}{l}
\quad P :: [\ P_1 \ ||\ P_2 \ ||\ P_3\ ] \\
\textit{where} \\
\quad P_1 :: b_0 := \mathit{true};\ b_1 := \mathit{true}; \\
\qquad *[\ b_0;\ I[b_0 := c_2] \longrightarrow \mathit{skip} \\
\qquad\quad \square \\
\qquad\quad b_1;\ J[b_1 := c_3] \longrightarrow \mathit{skip} \\
\qquad] \\
\\
\quad P_2 :: b_2 := \mathit{true};\ c_2 := \mathit{true}; \\
\qquad *[\ b_2;\ I[] \longrightarrow b_2 := c_2 \\
\qquad\quad \square \\
\qquad\quad c_2;\ K[] \longrightarrow c_2 := \mathit{false} \\
\qquad] \\
\\
\quad P_3 :: b_3 := \mathit{true};\ c_3 := \mathit{true}; \\
\qquad *[\ b_3;\ J[] \longrightarrow b_3 := c_3 \\
\qquad\quad \square \\
\qquad\quad c_3;\ K[] \longrightarrow c_3 := \mathit{false} \\
\qquad]
\end{array}
$$

Figure 2.3: Groupe de processus négligé

Exemple: (Groupe de processus négligé) Dans cet exemple P_1 peut interagir indéfiniment avec P_2 (via I) et P_3 (via J). Mais une fois que P_2 et P_3 ont interagi (via K) tous les processus terminent au bout d'un temps fini.

La terminaison du programme dépend donc ici de la possibilité pour P_2 et P_3 d'interagir. Une notion d'équité sur les groupes de processus ($\mathcal{SGF}$) est suffisante pour que le programme termine toujours. □

Enfin nous présentons en Figure 2.4 un programme dont la terminaison est garantie par une notion d'équité sur les interactions et pas par les notions d'équité sur les groupes de processus.

$$
\begin{array}{l}
\quad P :: [\ P_1 \ ||\ P_2\ ] \\
\textit{where} \\
\quad P_1 :: b_1 := \textit{true}; \\
\qquad *[\ b_1;\ I[] \longrightarrow \textit{skip} \\
\qquad\quad \Box \\
\qquad\quad b_1;\ J[] \longrightarrow b_1 := \textit{false} \\
\qquad] \\
\quad P_2 :: b_2 := \textit{true}; \\
\qquad *[\ b_2;\ I[] \longrightarrow \textit{skip} \\
\qquad\quad \Box \\
\qquad\quad b_2;\ J[] \longrightarrow b_2 := \textit{false} \\
\qquad]
\end{array}
$$

Figure 2.4: Interaction négligée

Exemple: (Interaction négligée) Puisque le programme comporte seulement deux processus (un seul groupe) les notions d'équité sur les processus et sur les groupes sont garanties.

P_1 et P_2 peuvent interagir indéfiniment via I. Mais dès qu'ils ont interagi via J le programme termine. Une notion d'équité sur les interactions ($\mathcal{SIF}$) est donc suffisante pour garantir la terminaison de ce programme. □

Dans les sections suivantes, nous montrons que si ces notions d'équité sont toutes réalisables, aucune notion d'équité faible n'accroît la vivacité et seules $\mathcal{WGF}$ et $\mathcal{WIF}$ sont compatibles avec l'équivalence. Cependant, lorsque le rendez-vous est binaire, $\mathcal{SPF}$ satisfait les trois critères. Ces résultats sont résumés en Table 2.2.

notion d'équité	réalisable	compatible avec l'équivalence	accroît la vivacité
$\mathcal{SIF}$	oui	non	oui
$\mathcal{SGF}$	oui	non	oui
$\mathcal{SPF}$	oui	non*	oui
$\mathcal{WIF}$	oui	oui	non
$\mathcal{WGF}$	oui	oui	non
$\mathcal{WPF}$	oui	non	non

* oui lorsque le rendez-vous est binaire

Table 2.2: Caractéristiques des notions d'équité

2.3.2 Réalisabilité

Proposition 2.9 *Les six notions d'équité sont réalisables.*

Idée de la preuve Pour chaque notion d'équité considérée un *scheduler* (explicit scheduler [OA88]) est exhibé. Un scheduler peut être vu comme un programme exécuté en parallèle avec le programme **P** considéré et ayant accès à toutes les variables de **P**. De plus le scheduler a accès aux variables de contrôle de tous les processus de **P** et ainsi peut diriger leur exécution.
On montre que n'importe quel préfixe d'une exécution légale peut être produit par le scheduler. Et si un préfixe d'une exécution a été produit par le scheduler alors il produira une extension telle que l'exécution obtenue satisfait la notion d'équité considérée. □

Exemple: (Un scheduler pour $\mathcal{WGF}$) Etant donné un programme **P**, on associe une variable *priorité* distincte à chaque groupe de processus qui peuvent tous participer (syntaxiquement) à une action. En particulier, pour les actions locales les groupes contiendront seulement le processus auquel est attribuée l'action. Un scheduler pour $\mathcal{WGF}$ peut être décrit ainsi :

(1) pour *chaque groupe* faire
si *il est exécutable* alors *décrémenter sa variable priorité de 1*
sinon *affecter à la variable priorité un entier positif arbitraire*

(2) *choisir un groupe exécutable avec une valeur minimale dans la variable priorité*

(3) *affecter à la variable priorité du groupe choisi un entier positif arbitraire*

(4) si *une action locale a été choisie* alors *exécuter cette action*
sinon *choisir et exécuter une des interactions exécutables du groupe*

(5) *retourner à l'étape* (1).

L'utilisation des variables priorité associées aux processus simples, définies pour les actions locales, garantie que le scheduler produit des exécutions satisfaisant la propriété de progrès minimal.

L'utilisation des affectations de valeur arbitraire et la possibilité d'avoir plusieurs variables priorité ayant une valeur minimale rendent ce scheduler non-déterministe. [AFK87] montre que ce scheduler peut produire toutes les exécutions équitables selon $\mathcal{WGF}$ et seulement celles-ci.

Des schedulers peuvent être obtenus pour les autres notions d'équité en modifiant les conditions d'exécutabilité et l'affectation des variables priorité. □

2.3.3 Gain en vivacité

Proposition 2.10 *$\mathcal{SPF}$, $\mathcal{SGF}$ et $\mathcal{SIF}$ accroissent la vivacité.*

Preuve Les exemples de la Section 2.3.1 (Figures 2.2, 2.3 et 2.4) décrivent des programmes qui terminent lorsque respectivement $\mathcal{SPF}$, $\mathcal{SGF}$ et $\mathcal{SIF}$ sont considérées et qui ne terminent pas sinon. □

Proposition 2.11 *$\mathcal{WPF}$, $\mathcal{WGF}$ et $\mathcal{WIF}$ n'accroissent pas la vivacité.*

Idée de la preuve Pour chaque programme **P**, on montre que si **comp(P)** contient une exécution infinie alors il contient une exécution équitable (selon la notion d'équité considérée) infinie. Et donc la notion d'équité considérée ne permet pas la terminaison de plus de programmes que la propriété de progrès minimal.
Une preuve détaillée est donnée dans [AFK87]. □

2.3.4 Compatibilité avec l'équivalence

Proposition 2.12

- *$\mathcal{WGF}$ et $\mathcal{WIF}$ sont compatibles avec l'équivalence,*
- *$\mathcal{WPF}$, $\mathcal{SPF}$, $\mathcal{SGF}$ et $\mathcal{SIF}$ ne sont pas compatibles avec l'équivalence,*
- *$\mathcal{SPF}$ est compatible avec l'équivalence si les interactions sont binaires.*

Preuve
• Pour montrer qu'une notion d'équité $\mathcal{F}$ n'est pas compatible avec l'équivalence on exhibe un programme **P** dont deux exécutions π et ρ sont telles que

$$\pi \in \mathcal{F}(\mathbf{P}) \wedge \pi \equiv \rho \wedge \rho \notin \mathcal{F}(\mathbf{P})$$

Ainsi pour montrer que $\mathcal{SPF}$ n'est pas compatible avec l'équivalence lorsqu'il existe une interaction comportant plus de deux participants nous considérons deux exécutions du programme de la Figure 2.5 :

La première exécution consiste en la répétition infinie du segment 1/,2/,3/,4/,5/,6/

1/ P_2 et P_3 exécutent J

2/ P_2 exécute *skip*

3/ P_3 exécute *skip*

4/ P_3 et P_4 exécutent K

5/ P_3 exécute *skip*

6/ P_4 exécute *skip*

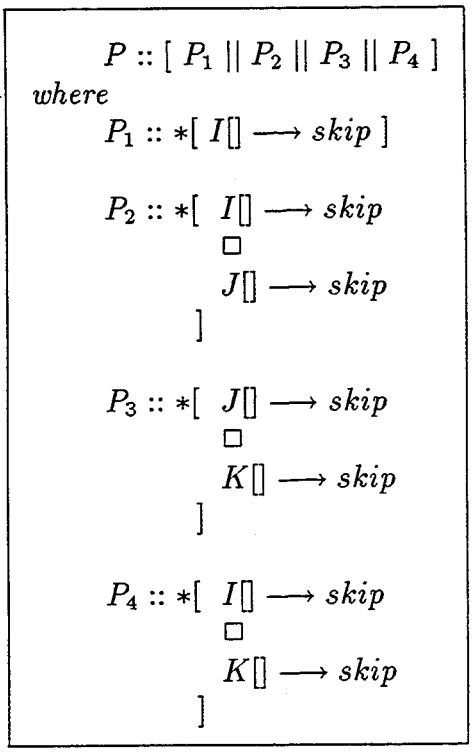


Figure 2.5:

Dans cette exécution, P_1 est infiniment souvent exécutable (initialement, après 2/ et après 6/) mais n'exécute aucune action. Ce n'est donc pas une exécution équitable selon $\mathcal{SPF}$.
La seconde exécution consiste en 1/ puis la répétition infinie du segment 3/,4/,2/,-5/,1/,6/. Ici, après la première étape 1/, P_1 n'est jamais exécutable. Il est clair que cette exécution est équitable selon $\mathcal{SPF}$ et qu'elle est équivalente à la précédente.

L'effet désiré est obtenu ici en différant les actions locales, rendant ainsi les processus partenaires de P_1 indisponibles et de ce fait interdisant l'interaction I. Notons qu'au moins trois participants dans une interaction sont nécessaires pour construire un tel exemple.

• Pour montrer qu'une notion d'équité $\mathcal{F}$ est compatible avec l'équivalence on montre que pour chaque programme $\mathbf{P}$, si π et ρ sont deux exécutions de $\mathbf{P}$ alors

$$(\pi \notin \mathcal{F}(\mathbf{P}) \wedge \pi \equiv \rho) \Rightarrow \rho \notin \mathcal{F}(\mathbf{P})$$

Ainsi nous montrons que $\mathcal{SPF}$ est compatible avec l'équivalence lorsque toutes les interactions comportent seulement deux participants.

Si une exécution π, d'un programme $\mathbf{P}$, est non équitable selon $\mathcal{SPF}$ alors il existe à partir d'un certain point de π un processus P_i qui est infiniment souvent

exécutable mais qui ne participe jamais à une interaction. Et donc P_i est, à partir d'un certain point de π, toujours prêt à interagir. En effet, d'après l'hypothèse de Choix uniforme, P_i ne peut pas choisir une action locale.
Considérons maintenant une exécution ρ équivalente à π. D'après le lemme des Projections égales, à partir d'un certain point de ρ, P_i est aussi toujours prêt pour au moins une interaction. D'après ce même lemme, il existe une infinité d'états dans lesquels les partenaires de P_i sont prêts à interagir avec lui. C'est-à-dire, puisque le rendez-vous est binaire, qu'il existe une infinité d'états dans lesquels des interactions, ayant P_i pour participant, sont exécutables. ρ n'est donc pas équitable selon $\mathcal{SPF}$. □

La compatibilité (ou l'incompatibilité) des autres notions d'équité se prouve de la même manière.
Il est aisé de trouver un programme dont certaines exécutions sont rendues équitables selon $\mathcal{WPF}$ en inhibant l'exécutabilité d'un processus, en faisant en sorte que les partenaires possibles de ce processus exécutent d'autres actions. Même s'il existe des exécutions équivalentes dans lesquelles une interaction est toujours possible pour le processus. Ce phénomène d'inhibition de l'exécutabilité d'un processus due à l'ordre d'exécution des actions est appelé *conspiration contre le processus.* C'est ce même phénomène qui est utilisé dans la première partie de la preuve précédente. Un phénomène comparable de conspiration contre un groupe ou une interaction peut être utilisé pour prouver que $\mathcal{SGF}$ et $\mathcal{SIF}$ ne sont pas compatibles avec l'équivalence.
$\mathcal{WGF}$ et $\mathcal{WIF}$ sont compatibles avec l'équivalence. En effet, une interaction est continuellement exécutable seulement si aucun de ses participants n'exécute une autre action. Si l'interaction n'est pas continuellement exécutable à cause de l'exécution d'une autre action, alors cette autre action sera aussi exécutée dans n'importe quelle exécution équivalente.

2.4 Notions d'équité lorsque la conspiration est limitée

Une *conspiration contre une interaction* I apparaît si, à partir d'un certain point d'une exécution, tous les participants à I sont infiniment souvent prêts pour I mais jamais au même moment, i.e. jamais dans le même état.
On peut définir d'une manière similaire la conspiration contre un groupe de processus ou contre un processus.

Notons qu'une conspiration peut être inhérente à la sémantique d'un programme. C'est le cas par exemple du programme de la Figure 2.6 où dès que l'interaction J est exécutée, P_1 et P_2 sont infiniment souvent prêts pour I mais jamais en même temps.

On peut aisément montrer que l'existence de conspirations contre des interactions (resp. groupe, processus) dues au seul ordre des exécutions (scheduling) des

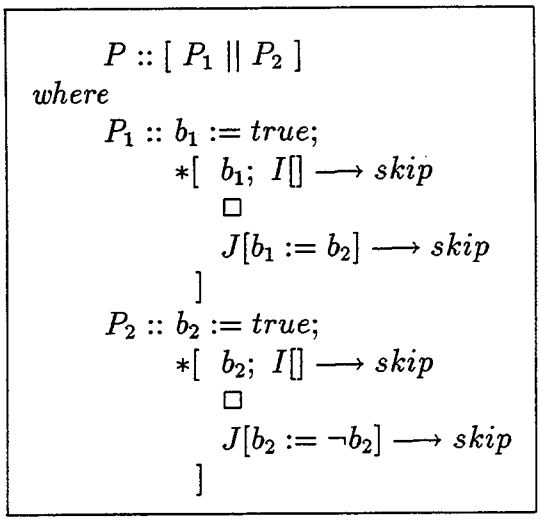


Figure 2.6: Conspiration inhérente à la sémantique d'un programme

actions indépendantes est la seule cause de l'incompatibilité de $\mathcal{SIF}$ (resp. $\mathcal{SGF}$, $\mathcal{SPF}$) avec l'équivalence.

En général les travaux ayant trait à un modèle basé sur le rendez-vous et aux notions d'équité présentées dans ce chapitre ont dû traiter ce problème de l'incompatibilité avec l'équivalence.

Une solution immédiate est, bien sûr, de ne considérer que des programmes sans conspiration. C'est le cas par exemple de [BKS88b] et [GFK84] qui définissent des assertions sémantiques suffisantes pour garantir la compatibilité avec l'équivalence quand on ne considère que des programmes satisfaisant ces assertions. Ainsi dans [GFK84] un système de preuve non complet est proposé pour CSP avec $\mathcal{SIF}$. Il contient des règles permettant de montrer que pour un programme donné la notion d'équité respecte les classes d'équivalence des exécutions du programme. Les autres règles permettent de prouver la terminaison d'un tel programme.

Une autre solution simple est d'imposer des contraintes syntaxiques suffisantes pour que *tous* les programmes soient sans conspiration. C'est le cas par exemple dans les modèles que nous décrivons dans la section suivante.

2.4.1 Modèles où le contrôle est localisé

Le contrôle est dit *localisé* quand pour toute interaction exécutable il y a toujours un seul de ses processus participants qui exécute une commande gardée. C'est donc ce processus particulier, appelé processus *maître*, qui décide l'éventuelle exécution de cette interaction et qui impose sa décision aux autres processus participants, appelés *esclaves*.

Proposition 2.13 *Dans un modèle où le contrôle est localisé, les six notions d'équité sont compatibles avec l'équivalence.*

Preuve Nous montrons que $\mathcal{SIF}$ est compatible avec l'équivalence. Les preuves pour les autres notions d'équité sont similaires.

Si une exécution π, d'un programme **P**, est non équitable selon $\mathcal{SIF}$ alors il existe à partir d'un certain point de π une interaction I qui est infiniment souvent exécutable mais qui n'est jamais exécutée. Soit P_I le processus maître de I. P_I est donc, à partir de ce point, le seul participant à I à ne pas être toujours prêt pour I. Les autres participants (esclaves) sont nécessairement toujours prêts à interagir. En effet, le contrôle étant localisé ces processus ne peuvent pas choisir et effectuer une autre action.
Considérons maintenant une exécution ρ équivalente à π. D'après le lemme des Projections égales, à partir d'un certain point de ρ, les participants esclaves de I sont toujours prêts pour I. D'après ce même lemme, P_I est infiniment souvent prêt à interagir avec eux. C'est-à-dire qu'il existe une infinité d'états dans lesquels I est exécutable. Mais, toujours d'après le lemme des Projections égales, I n'est jamais exécutée. ρ n'est donc pas équitable selon $\mathcal{SIF}$. □

Différents modèles avec contrôle localisé ont été proposés jusqu'à présent. Par exemple, en CSP seules les commandes de réception de message sont autorisées dans les gardes des commandes alternatives. Ceci assure que lorsque une paire de commandes de communication correspondent (match), une au moins est inconditionnelle : la commande d'envoi du message, et c'est donc le processus récepteur qui impose son choix aux processus émetteurs.

Notons que dans le modèle ACF (ADA Communication Fragment) et dans les modèles avec envoi non bloquant (nonblocking send), considérés dans [AFK87], le contrôle est localisé. En effet dans ACF le processus maître est celui qui effectue une commande *select* ou simplement une commande *accept*, et les esclaves sont les processus effectuant un *call*. Et dans un modèle avec envoi non bloquant, le maître est celui qui effectue une commande gardée (où les gardes ne peuvent contenir que des commandes de réception (receive) de message), les esclaves étant les processus qui lui ont envoyé un message.

Les solutions, que nous venons d'évoquer, ne permettent pas de considérer des programmes, tels le programme de la Figure 2.1 (philosophes mangeurs), qui contiennent une conspiration. Or seul l'ordre des actions indépendantes est à l'origine de ces conspirations. En contrôlant (dynamiquement) cet ordre il doit être possible de les éliminer. C'est ce qui est réalisé par l'Hyperéquité [AFG90]. Nous décrivons brièvement cette nouvelle notion d'équité.

2.4.2 Hyperéquité $\mathcal{HF}$

Pour que $\mathcal{SIF}$ respecte les classes d'équivalence des exécutions d'un programme il suffit de contrôler les exécutions des processus de manière que les interactions soient exécutables au bout d'un temps fini lorsque leurs participants sont infiniment souvent prêts pour elles. Cela peut être réalisé simplement en *gelant* (frozen) les participants prêts pour une interaction suffisamment longtemps c'est-à-dire en garantissant qu'ils resteront prêts pour cette interaction jusqu'à ce qu'elle soit exécutable. C'est cette idée qui est à l'origine de l'Hyperéquité [AFG90].

Définition 2.14 (Résistance à la conspiration)

1. *Une* configuration $< [S_1 \mid\mid \ldots \mid\mid S_n], \sigma >$ *est constituée par un programme et un état global. Elle représente un point d'une exécution où* S_i *est le reste du programme que le processus* P_i *doit encore exécuter et* σ *est l'état courant en ce point.*

2. *Soit* $\mathbf{P}$ *un programme, I une interaction de* $\mathbf{P}$ *et A un ensemble arbitraire constitué de processus participant à I. Le* programme dérivé $\mathbf{P}_{I,A}$ *est obtenu à partir de* $\mathbf{P}$ *en fermant les gardes locales de toutes les interactions différentes de I dans tous les processus de A.*
   En d'autres termes, dans le programme dérivé les processus de A ne peuvent pas participer à des interactions autres que I.

3. *Une interaction I est* résistante à la conspiration *dans un programme* $\mathbf{P}$ *ssi pour chaque exécution* π *équitable selon* $\mathcal{SIF}$ *la condition suivante est satisfaite :*

   > *Soit* π_1 *un préfixe fini de* π *et* $< S, \sigma >$ *sa configuration finale. Soit* A_I *l'ensemble des participants de I qui sont prêts pour I dans cette configuration. Alors, au cours de chaque exécution* π_2*, équitable selon* $\mathcal{SIF}$*, du programme dérivé* S_{I,A_I} *(débutant dans l'état* σ*), il existe un participant* $P_j \notin A_I$ *qui est prêt pour I au bout d'un temps fini.*

Les processus de A_I sont dits *gelés sur* I (I-frozen). La *résistance à la conspiration* est garantie lorsque les participants d'une interaction I peuvent devenir prêts pour I *indépendamment* les uns des autres. Cette indépendance se manifeste par le fait que le gel pour I d'un sous-ensemble arbitraire de ses participants n'interdit pas aux autres participants de devenir eux aussi prêts pour I. Ainsi un scheduler qui gèle les uns après les autres les participants de I garantit que I sera exécutable au bout d'un temps fini.

Définition 2.15 (Hyperéquité $\mathcal{HF}$) *Soit* **P** *un programme dans lequel toutes les interactions sont résistantes à la conspiration. Alors une exécution infinie* π *de* **P** *est* hyperéquitable *(*$\pi \in \mathcal{HF}(\mathbf{P})$*) ssi* π *est équitable selon* $\mathcal{SIF}$ *(*$\pi \in \mathcal{SIF}(\mathbf{P})$*) et chaque interaction est infiniment souvent exécutable dans* π*. De plus, toute exécution finie est hyperéquitable.*

Si dans **P** *il existe une interaction qui n'est pas résistante à la conspiration alors toutes les exécutions de* **P** *sont hyperéquitables.*

[AFG90] montre que $\mathcal{HF}$ est réalisable, accroît la vivacité (par exemple le programme de la Figure 2.1 est une solution sans famine[2] du problème des philosophes mangeurs) et est compatible avec l'équivalence. Nous renvoyons à cet article pour plus de détails.

Notons tout de même la différence essentielle entre l'Hyperéquité et les notions d'équité étudiées précédemment : l'Hyperéquité implique l'exécutabilité au bout d'un temps fini (et par la suite l'exécution grâce à $\mathcal{SIF}$) si la résistance à la conspiration est assurée. En revanche, les notions d'équité précédentes impliquent l'exécution seulement si une condition sur l'exécutabilité est satisfaite. Une condition d'exécutabilité pour une interaction particulière I d'un programme **P** peut être satisfaite dans certaines exécutions π mais pas dans d'autres, i.e. c'est une fonction de I, **P** et π, tandis que la résistance à la conspiration n'est pas une propriété des exécutions prises individuellement : c'est une fonction seulement de I et **P**.

2.5 Conclusions

La définition des critères d'acceptabilité et leur évaluation systématique a permis de mettre en lumière les avantages et les inconvénients des différentes notions d'équité étudiées.

Le critère le plus important est bien sûr l'accroissement de la vivacité des programmes. Nous avons montré que les trois notions d'équité faibles n'accroissent pas la vivacité. Nous avons montré de plus que $\mathcal{WIF}$ est réalisable et compatible avec l'équivalence. Cela suffit à justifier notre choix d'utiliser dans la suite $\mathcal{WIF}$ comme propriété de vivacité fondamentale. Ce choix permettra en fait une comparaison plus aisée de nos résultats avec d'autres travaux qui utilisent aussi $\mathcal{WIF}$ (e.g. [Lev88]).

Nous nous intéressons dans cette thèse à l'implémentation distribuée du rendez-vous quand une des notions d'équité définies dans ce chapitre est employée. La réalisabilité de la notion d'équité choisie est dans ce cadre un critère essentiel. En

[2] Une solution est dite *sans famine* quand tout philosophe affamé parvient toujours à manger au bout d'un temps fini.

fait la réalisabilité permet seulement de montrer qu'une implémentation centralisée existe : le scheduler. Bien sûr [OA88] montre comment intégrer (embedding) le scheduler dans le programme considéré en conservant la structure parallèle du programme. Néanmoins cette intégration nécessite quand même une certaine centralisation du contrôle (e.g. synchronisation grâce à la commande *await* nécessitant une mémoire partagée).

Il nous semble que seules les notions d'équité compatibles avec l'équivalence peuvent être implémentées d'une manière distribuée. Par implémentation distribuée nous entendons :

1. Il est impossible pour le scheduler de déterminer a priori si un processus donné deviendra prêt.
2. Le scheduler ne peut pas contrôler les actions d'un processus quand il n'est pas prêt. En particulier le scheduler ne peut pas contrôler les changements d'état du processus vers l'état prêt.
3. Les changements d'état des processus ne sont pas immédiatement observables (ceci est dû à l'asynchronisme des processus et du scheduler).

Dans une telle implémentation le scheduler ne peut pas distinguer des exécutions équivalentes selon $\equiv$. Si la notion d'équité n'est pas compatible avec l'équivalence, rien n'empêche donc le scheduler de juger équitable une exécution qui ne l'est pas. Ce raisonnement informel est corroboré par les résultats d'impossibilité de [TB91] où Y.-K. Tsay et R. Bagrodia montrent que $\mathcal{SPF}$ (lorsqu'il y a des interactions qui ont plus de deux participants) et $\mathcal{SIF}$ sont impossibles à implémenter d'une manière distribuée.

En revanche lorsqu'il est toujours possible de supposer que tout processus deviendra prêt au bout d'un temps fini, $\mathcal{SIF}$ peut être garantie aisément. En effet à n'importe quel moment de l'exécution le scheduler sait que toute interaction deviendra exécutable au bout d'un temps fini. Il suffit alors que le scheduler traite l'une après l'autre toutes les interactions et pour chaque interaction attende jusqu'à ce qu'elle soit exécutable. [Sis84] décrit un tel scheduler.

De même $\mathcal{SIF}$ peut être implémentée aisément lorsqu'on suppose que le scheduler peut contrôler les changements d'état des processus ou a une connaissance immédiate de ces changements. En effet, le scheduler a alors toujours une vue précise de l'état global du système et donc des interactions exécutables.

Parmi les notions d'équité étudiées ici seules $\mathcal{SPF}$ (quand le rendez-vous est binaire) et $\mathcal{HF}$ satisfont les trois critères à la fois. $\mathcal{HF}$ est d'ailleurs définie pour cela ...

Nous avons montré qu'on ne peut pas éliminer facilement les conspirations. Les solutions les plus simples sont trop contraignantes : il s'agit soit d'ignorer les programmes avec conspiration, soit d'empêcher leur existence en réduisant la symétrie

dans les processus. La meilleure solution semble être d'éliminer dynamiquement les conspirations grâce à un contrôle de l'exécution des processus. Nous avons montré que cela peut être réalisé pour une classe de programmes dits "résistants à la conspiration".

La résistance à la conspiration est une condition assez forte mais elle apparaît souvent lorsque les processus sont faiblement liés (loosely coupled) dans la mesure où il n'y a pas de dépendance causale forte dans les séquences d'interactions consécutives auxquelles un processus participe. Le problème des philosophes mangeurs (Figure 2.1) est un exemple typique de situation où il y a résistance à la conspiration. L'existence d'une classe plus générale de programmes contrôlables reste un problème ouvert.

Les interactions constituent le sujet principal des notions d'équité définies ici. Des notions d'équité concernant toutes les actions (locales et conjointes) ont aussi été étudiées (e.g. [BKS88b], [AFL90]). Les contraintes sur les modèles sont bien sûr adaptées. Ainsi [AFL90] définit la *U-équité* (U-fairness aussi appelée *handshake fairness* dans [BKS88b]) et montre qu'elle satisfait les trois critères dans un modèle où toutes les actions sont gardées et où la seule contrainte imposée est le progrès minimal. Pour plus de détails nous renvoyons à la lecture de [AFL90].

Chapitre 3

Spécification et construction des protocoles

Nous rappelons en Section 3.1 les principes de la *méthode Transition-Axiom* [LS85, Lam89] de L. Lamport pour écrire les spécifications.

Puis nous définissons en Section 3.2 un mini-langage de programmation suffisant pour écrire nos schémas de protocoles.

Enfin, en Section 3.3, nous faisons quelques rappels sur les techniques utilisées pour vérifier la correction des protocoles et des schémas de protocoles.

3.1 Spécifications

Le matériel de cette section est en grande partie tiré de [Lam83] et [LS85]. Nous renvoyons systématiquement à la lecture de ces articles pour plus de détails.

Pour écrire des spécifications, nous utilisons la logique du temps linéaire de [Lam80b]. Nous donnons ici une brève description des *assertions* (formules de logique temporelle bien formées) dont nous aurons besoin.

3.1.1 Logique temporelle

Comme au chapitre précédent nous supposons que l'exécution d'un programme est représentée par une séquence de transitions d'état de la forme

$$\sigma \xrightarrow{\alpha} \sigma'$$

qui dénote le fait que l'action atomique α transforme l'état σ du programme en l'état σ'.

Plus précisément un programme est représenté un triplet (S, A, Π) où S est un ensemble d'états, A un ensemble d'actions et Π un ensemble de séquences infinies de la forme

$$\sigma_0 \xrightarrow{\alpha_1} \sigma_1 \xrightarrow{\alpha_2} \sigma_2 \cdots$$

avec les σ_i dans S et les α_i dans A.
Nous supposerons que pour chaque π de Π, la séquence

$$\pi^{+n} \equiv \sigma_n \xrightarrow{\alpha_{n+1}} \sigma_{n+1} \xrightarrow{\alpha_{n+2}} \sigma_{n+2} \cdots$$

est aussi dans Π, quelque soit $n \geq 0$ (tail closure).

Un état représente un instantané (snapshot) du programme à un instant donné. Il contient toutes les informations nécessaires pour déterminer les comportements futurs possibles du programme. L'état doit donc contenir non seulement la valeur des variables du programme, mais aussi les valeurs des variables de contrôle (program counters) des processus, les valeurs des paramètres, le contenu des queues de messages, l'état des lignes de transmission ...

Une *fonction d'état* est une fonction dont le domaine est S et dont le codomaine est inclus dans un ensemble de valeurs V contenant les éléments $true$ et $false$. Un *prédicat d'état* est une fonction d'état dont le codomaine est inclus dans $\{true, false\}$. Nous supposons un calcul des prédicats du premier ordre avec égalité pour les fonctions d'état, permettant la quantification sur V. Nous écrivons $\sigma \models P$ pour dénoter la valeur du prédicat d'état P sur l'élément σ de S.

Nous utiliserons deux sortes de fonctions d'état primitives :

variables du programme : Une variable x du programme est une fonction d'état qui associe à tout état σ la valeur de x dans cet état.

prédicats sur le contrôle : Si δ est un point de contrôle dans le programme (i.e. une valeur possible du program counter) alors $at\,\delta$ dénote le prédicat qui est vrai dans un état si et seulement si le contrôle est au point δ dans cet état.

Nous pouvons construire des fonctions d'état plus complexes à partir de ces fonctions primitives. Par exemple, $at\,\delta \wedge x > 0$ est un prédicat qui est vrai dans un état si et seulement si le contrôle est au point δ et la variable x a une valeur positive dans cet état.

Un *prédicat d'action* est une fonction booléenne sur A. $\alpha \models Q$ dénote la valeur du prédicat d'action Q sur l'élément α de A.

Nous décrivons maintenant ce que signifie pour une assertion en logique temporelle d'être vraie à "l'instant i" dans l'exécution

$$\sigma_0 \xrightarrow{\alpha_1} \sigma_1 \xrightarrow{\alpha_2} \sigma_2 \cdots$$

i.e. à l'instant juste avant la $(i+1)$ème transition, quand le programme est dans l'état σ_i. La description suivante concerne cinq types d'assertions où P dénote un prédicat :

P :	vrai à l'instant i ssi P est vrai dans l'état σ_i
$\Box P$:	vrai à l'instant i ssi P est vrai à tous les instants $j \geq i$
$\Diamond P$:	vrai à l'instant i ssi P est vrai à un instant $j \geq i$
$\Diamond\Box P$:	vrai à l'instant i ssi P est vrai à tous les instants $k \geq j$ pour un $j \geq i$
$\Box\Diamond P$:	vrai à l'instant i ssi P est vrai à une infinité d'instants $j \geq i$

Toutes les propriétés de vivacité que nous utiliserons seront des combinaisons logiques de ces cinq types d'assertions.

Nous utilisons l'abréviation $P \rightsquigarrow Q$ pour $\Box(P \Rightarrow \Diamond Q)$. Et donc, la propriété $P \rightsquigarrow Q$ signifie que si P est vrai à un instant i, alors Q doit être vrai au même instant ou dans un instant futur $j \geq i$.

Plus précisément, les formules de logique temporelle sont construites à partir des prédicats d'état et des prédicats d'action en utilisant les opérateurs logiques usuels $\neg$ et $\vee$, la quantification sur V, et le connecteur binaire $\trianglelefteq$[1]. Nous définissons inductivement la relation $\pi \models P$, "P est valide pour la séquence π", pour chaque séquence

$$\pi = \sigma_0 \xrightarrow{\alpha_1} \sigma_1 \xrightarrow{\alpha_2} \sigma_2 \cdots$$

de Π et chaque formule de logique temporelle P :

$\pi \models P$	ssi $\sigma_0 \models P$ si P est un prédicat d'état
$\pi \models P$	ssi $\alpha_1 \models P$ si P est un prédicat d'action
$\pi \models \neg P$	ssi $\neg(\pi \models P)$
$\pi \models (P \vee Q)$	ssi $(\pi \models P) \vee (\pi \models Q)$
$\pi \models \forall x.P$	ssi $\forall v : v \in V.\ \pi \models P[x := v]$
$\pi \models P \trianglelefteq Q$	ssi $\forall n : n \geq 0.\ (\forall m.\ 0 \leq m \leq n \Rightarrow \pi^{+m} \models P) \Rightarrow \pi^{+n} \models Q$

Une formule P est dite *valide pour le programme* (S, A, Π), et on écrit $\models P$, si $\pi \models P$ est vrai pour tout π de Π.

[1] noté $\Box$ dans [Lam80b]

Les opérateurs $\Box$ et $\Diamond$ sont définis par

$$\Box P \equiv true \trianglelefteq P$$
$$\Diamond P \equiv \neg\Box\neg P$$

Nous définissons aussi l'opérateur binaire $\lhd$ par

$$P \lhd Q \equiv (P \vee \neg Q) \trianglelefteq Q$$

Il s'ensuit d'après la définition de $\pi \models P \trianglelefteq Q$:
$\pi \models P \lhd Q$ ssi $\forall n : n \geq 0.(\forall m.0 \leq m < n \Rightarrow \pi^{+m} \models P) \Rightarrow \pi^{+n} \models Q$

$P \trianglelefteq Q$ affirme que Q est vrai au moins aussi longtemps que P, et $P \lhd Q$ affirme que Q est vrai au moins un pas plus loin que P. Les deux opérateurs satisfont les relations de transitivité attendues (e.g. $P \trianglelefteq Q \wedge Q \lhd R \Rightarrow P \lhd R$).

Notons que $(\neg Q) \trianglelefteq P$ signifie que P est vrai jusqu'à ce que Q le soit. L'opérateur *(weak) until*, souvent utilisé, peut alors être défini par

$$P \; until \; Q \equiv (\neg Q) \trianglelefteq P$$

3.1.2 Spécifications

Pour spécifier un programme, nous devons spécifier l'ensemble de toutes ses exécutions. Une spécification consiste en une collection de conditions sur ces exécutions. Un programme satisfait la spécification si chacune de ses exécutions satisfait ces conditions. Nous décrivons informellement ici la sémantique des spécifications; une définition en termes de logique temporelle est donnée dans [Lam83].

Une spécification a la forme suivante :

Fonctions d'état : $f_1 : R_1, \ldots, f_n : R_n$
Conditions initiales : $I_1, \ldots, I_m$
Propriétés : $P_1, \ldots, P_q$

avec l'interprétation suivante :

IL EXISTE des fonctions d'état $f_1, \ldots, f_n$ TELLES QUE :
Le codomaine de f_i est R_i, pour chaque i, ET
SI l'état initial de l'exécution satisfait $I_1, \ldots, I_m$
ALORS les propriétés $P_1, \ldots, P_q$ sont satisfaites dans tous les états de l'exécution

Les valeurs possibles pour chaque fonction d'état f_i doivent appartenir à l'ensemble R_i.

Les conditions initiales I_j sont des prédicats. La spécification place des contraintes seulement sur toutes les exécutions pour lesquelles chaque I_j est vrai dans l'état initial σ_0.

Chaque propriété P_j exprime une contrainte sur la totalité de l'exécution

$$\sigma_0 \xrightarrow{\alpha_1} \sigma_1 \xrightarrow{\alpha_2} \sigma_2 \cdots$$

Nous utilisons deux types de propriétés : les propriétés de sûreté (safety) et les propriétés de vivacité (liveness).

3.1.3 Propriétés de sûreté

Une propriété de sûreté affirme que quelque chose ne peut jamais arriver.

Une propriété peut être un prédicat. Si dans une spécification, une propriété P_j est un prédicat, elle assure que dans chaque état d'une exécution (vérifiant initialement $I_1, \ldots, I_m$) le prédicat P_j est vrai. Une telle propriété est appelée un *invariant.*

Une autre forme de propriété de sûreté affirme qu'une fonction d'état ne peut changer quand elle est supposée ne pas pouvoir changer. Cela est exprimé par :

a **leaves unchanged** f **when** Q

où a est un ensemble d'actions, f est une fonction d'état, et Q est un prédicat. Son interprétation est la suivante :

Pour chaque transition $\sigma \xrightarrow{\alpha} \sigma'$ de l'exécution, *si*

(i) α est dans a et
(ii) Q est vrai dans l'état σ,

alors la valeur de f dans l'état σ' est égale à sa valeur dans σ.

En d'autre termes, cette propriété assure que n'importe quelle action de a laisse f inchangée quand elle agit dans un état de départ satisfaisant Q. Si a est l'ensemble de toutes les actions, alors "a **leaves**" peut être omis. De même si Q est le prédicat trivial *true*, alors "**when** Q" peut être omis.
Une définition de cette propriété en termes de logique temporelle est donnée dans [Lam83].

Plutôt qu'indiquer quand les fonctions d'état ne peuvent pas changer, nous pouvons établir quand elles peuvent changer en utilisant la construction suivante :

allowed changes to g_1 **when** Q_1,
$\vdots$
g_p **when** Q_p :

$a_1 : R_1 \rightarrow S_1$,
$\vdots$
$a_u : R_u \rightarrow S_u$

où les g_i sont des fonctions d'état, les Q_i et R_j sont des prédicats, les a_j sont des ensembles d'actions, et les S_j sont des fonctions booléennes sur des paires ordonnées d'états. Cette propriété affirme pour chaque i :

Pour chaque transition $\sigma \xrightarrow{\alpha} \sigma'$ de l'exécution, *si*

(i) Q_i est vrai dans l'état σ et
(ii) la valeur de g_i dans l'état σ' est différente de sa valeur dans σ,

alors il y a un j tel que

(i) α est dans a_j
(ii) R_j est vrai dans l'état σ
(iii) S_j est vrai pour la paire d'états (σ, σ')

Cette propriété contraint les changements des fonctions d'état g_i en décrivant exactement quelles transitions peuvent les changer. Chaque transition qui change g_i débutant dans un état où Q_i est vrai satisfait une des spécification de transition

$$a_j : R_j \rightarrow S_j$$

Dans cette spécification de transition, R_j est un prédicat indiquant ce qui doit être vrai dans l'état de départ (enabling predicate), et S_j spécifie une relation entre l'état de départ σ et l'état final σ'. Nous exigeons que si S_j est vrai pour la paire (σ, σ'), alors R_j doit être faux pour σ'. Cette hypothèse est nécessaire pour la définition de la propriété en termes de logique temporelle donnée dans [Lam83].

Les fonctions booléennes S_j sont décrites par des expressions comportant des fonctions d'état avec prime (e.g. f') ou sans prime (e.g. f). Lorsqu'on évalue une telle expression sur une paire d'états (σ, σ'), les fonctions d'état sans prime sont évaluées sur l'état de départ σ et celles avec prime sur l'état final σ'. Ainsi, $g_1 > f'$ est vrai pour la paire d'états (σ, σ') si et seulement si la valeur de la fonction d'état g_1 évaluée dans l'état σ est plus grande que la valeur de la fonction d'état f évaluée dans l'état σ'.

Les expressions S_j ont souvent la forme

$$g_i' = g_i \wedge \ldots,$$

indiquant que la transition n'est pas autorisée à changer la valeur de g_i. Nous faisons la convention que si une des fonctions d'état g_i n'apparaît pas avec prime dans l'expression S_j, alors S_j est supposée contenir une clause conjonctive non écrite de la forme $g_i' = g_i$. Ainsi, toute g_i qui n'apparaît pas dans une spécification de transition ne peut pas être changée par la transition.
Une autre convention est d'omettre "**when** Q_i" quand Q_i est le prédicat trivial *true*.

Nous écrivons parfois des propriétés "**allowed changes**" dans lesquelles il y a des spécifications de transition ayant la forme

for all $v : a : R \rightarrow S$,

où v peut apparaître comme une variable libre dans les expressions R et S. Cela signifie que pour n'importe quelle valeur de v, une transition satisfaisant la spécification de transition $a : R \rightarrow S$ pour cette valeur de v peut changer les valeurs des g_i.

3.1.4 Propriétés de vivacité

Les propriétés de sûreté énoncent que quelque chose peut ou ne peut pas arriver, mais ne garantissent pas que quelque chose arrivera. Par exemple, une propriété "**allowed changes**" spécifie les transitions d'état qui *peuvent* apparaître dans une exécution; elle ne spécifie pas qu'une de ces transitions apparaîtra. Les propriétés de vivacité établissent ce qui *doit* arriver. Elles sont spécifiées en utilisant la logique temporelle décrite précédemment.

Nous ne donnerons pas ici d'exemple de spécification. Une spécification des protocoles pour le rendez-vous est décrite dans le chapitre suivant et nous conseillons, pour une meilleure compréhension, de relire cette section après l'exposé de cette spécification. Pour d'autres exemples nous renvoyons à [Lam83] où sont spécifiés des protocoles "standards" (e.g. queue FIFO, protocole du bit alterné).

3.2 Construction des protocoles

La méthode de construction des protocoles que nous proposons consiste simplement à mettre en œuvre ces protocoles à partir d'un schéma donné.

Le corps d'un schéma est syntaxiquement identique à un protocole mais contient des identificateurs libres, appelés *entités abstraites*, à la place de certaines parties du code [BK88]. Le reste du schéma est constitué d'une spécification de ces entités abstraites.

Nous décrivons d'abord le langage utilisé pour écrire des schémas (Section 3.2.1). Puis nous définissons les prédicats sur le contrôle utilisés dans les spécifications des entités abstraites (Section 3.2.2). Enfin nous donnons un exemple de schéma pour le problème bien connu des philosophes buveurs [CM84] (Section 3.2.3).

3.2.1 Le langage de programmation

Nous utilisons un langage très simple. Il contient une instruction d'affectation atomique, les structures de contrôle usuelles : la séquence (;), les instructions **if** et **while**, plus l'instruction **cobegin**. Ces constructions sont illustrées par le programme de la Figure 3.1 (c'est d'ailleurs le seul intérêt de ce programme).

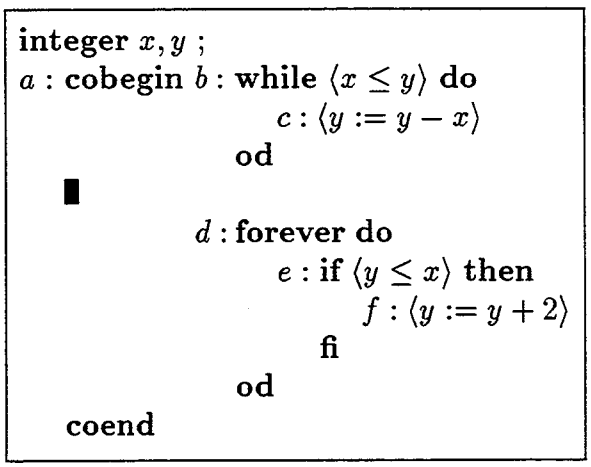


Figure 3.1: Un exemple de programme

Seule l'instruction **cobegin** n'est pas usuelle. Une instruction

cobegin S_1 ▮ ... ▮ S_n **coend**

permet d'exécuter les instructions S_1, ..., S_n en parallèle. Les S_i sont appelés *processus.*

Pour simplifier, nous supposerons que les déclarations des variables peuvent seulement apparaître au début du programme : toutes les variables sont globales au programme.

Nous indiquons les actions atomiques en les mettants entre crochets $\langle S \rangle$. Nous supposerons ici que toutes les affectations et tous les tests des instructions **if** et **while** sont atomiques[2].

Notons aussi que **forever** est une abréviation pour **while** $\langle true \rangle$.

[2]Les affectations et les tests non atomiques sont considérés dans [Lam80a].

En général, pour lever toute ambiguïté sur les instructions considérées, nous utiliserons leurs étiquettes dans le programme.

Nous définissons la sémantique d'un programme comme étant l'ensemble de toutes les exécutions possibles de ce programme. Un état σ du programme consiste en :

- l'affectation d'une valeur à chaque variable du programme,
- et une partie contrôle $readyact(\sigma)$ consistant en un ensemble d'actions atomiques. (Quand le programme est en cours d'exécution, l'unique prochaine action exécutée est choisie dans $readyact(\sigma)$.)

Un exemple d'état du programme de la Figure 3.1 est

$$[x = 2,\ y = 3;\ readyact = \{c, e\}].$$

Ici les variables x et y ont pour valeur 2 et 3 respectivement, et la partie contrôle indique qu'il y a deux possibilités pour la prochaine action à exécuter :

- l'affectation $c : \langle y := y - x \rangle$,
- et le test $e : \langle y \leq x \rangle$ du **if**.

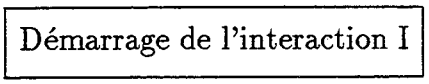
L'exemple suivant est un exemple de séquence d'exécution pour le programme de la Figure 3.1. L'exécution démarre avec les valeurs initiales 2 pour x et 3 pour y, et le contrôle au début du programme. (Pour simplifier nous omettons les noms des variables dans les états)

$$\begin{aligned}
\sigma_0 &= [2, 3; \{b, d\}];\\
\sigma_1 &= [2, 3; \{c, d\}];\\
\sigma_2 &= [2, 3; \{c, e\}];\\
\sigma_3 &= [2, 1; \{b, e\}];\\
\sigma_4 &= [2, 1; \{e\}];\\
\sigma_5 &= [2, 1; \{f\}];\\
\sigma_6 &= [2, 3; \{d\}];\\
&\vdots\\
\sigma_{2i+7} &= [2, 3; \{e\}];\ i = 0, 1, \ldots\\
\sigma_{2i+8} &= [2, 3; \{d\}];\ i = 0, 1, \ldots\\
&\vdots
\end{aligned}$$

Les exécutions d'un programme doivent satisfaire la propriété de Progrès minimal, i.e. si une action atomique a est dans $readyact(\sigma_i)$ alors il existe un instant $j \geq i$ tel que σ_j est obtenu en exécutant a.
On montre aisément que l'exécution décrite plus haut satisfait la propriété de Progrès minimal.

Le corps d'un schéma est syntaxiquement identique à un programme mais contient des identificateurs libres, appelés *entités abstraites*, à la place de certaines instructions.

Les entités abstraites sont signalées par des "boites". Par exemple



est une entité abstraite. Notons que l'étiquette de la boite n'est qu'indicative : seule la spécification associée au corps du schéma décrit précisément l'entité abstraite.

3.2.2 Les prédicats sur le contrôle

Le prédicat *at* est suffisant lorsqu'on considère seulement des instructions atomiques. Lorsqu'on utilise des instructions non-atomiques ou comme ici des entités abstraites, il devient nécessaire de pouvoir parler de points de contrôle situés quelque part dans l'instruction ou l'entité abstraite, ou situés immédiatement après une instruction ou une entité. Nous utilisons trois sortes de prédicats pour parler du contrôle : $at\,S$, $in\,S$ et $after\,S$ où S est une instruction du programme.

Le prédicat $at\,S$ est vrai dans tous les états où le contrôle est au début de S. Nous pouvons le définir ainsi :

$\sigma \models at\,S$ ssi
- SI S est c : $\langle x := e \rangle$ ALORS $c \in readyact(\sigma)$;
- SI S est c : **if** $\langle B \rangle$ **then** T **fi** ALORS $c \in readyact(\sigma)$;
- SI S est c : **while** $\langle B \rangle$ **do** T **od** ALORS $c \in readyact(\sigma)$;
- SI S est **cobegin** $T_1 \blacksquare \ldots \blacksquare T_n$ **coend** ALORS $(\sigma \models at\,T_1)$ et ... et $(\sigma \models at\,T_n)$;
- SI S est $T;U$ ALORS $\sigma \models at\,T$.

Par exemple, si on considère le programme de la Figure 3.1, $at\,c$ est vrai lorsque le contrôle est au début de $\langle y := y - x \rangle$, et $at\,e$ est vrai lorsque le contrôle est au début du test $\langle y \leq x \rangle$ du **if**.

Le prédicat $in\,S$ est vrai dans tous les états où le contrôle est au début de S ou quelque part dans S. En d'autres termes, $\sigma \models in\,S$ si et seulement si soit $\sigma \models at\,S$ soit il existe une partie T de S telle que $\sigma \models at\,T$.

Par exemple, si on considère le programme de la Figure 3.1,

$$in\,d \equiv at\,d \vee at\,e \vee at\,f$$

Le prédicat $after\,S$ est vrai dans tous les états où le contrôle est immédiatement après l'instruction S. Pour définir $after\,S$ nous considérons l'instruction T qui contient "immédiatement" S :

$\sigma \models after\,S$ ssi
SI S est le programme entier ALORS $(readyact(\sigma) = \emptyset)$;
SI T est c : **if** $\langle B \rangle$ **then** S **fi** ALORS $\sigma \models after\,T$;
SI T est c : **while** $\langle B \rangle$ **do** S **od** ALORS $\sigma \models at\,T$;
SI T est **cobegin** ... ▮ S ▮ ... **coend**
ALORS $(\sigma \models after\,T)$ ou $[(\sigma \models in\,T) \wedge \neg(\sigma \models in\,S)]$;
SI T est $S;U$ ALORS $\sigma \models at\,U$;
SI T est $U;S$ ALORS $\sigma \models after\,T$.

Notons qu'être après le corps d'un **while** c'est être au début du test. Et être après un processus S dans un **cobegin** signifie que le **cobegin** est terminé ou bien qu'un de ses processus (mais pas S) est encore en cours d'exécution.

3.2.3 Exemple : un schéma pour le problème des philosophes buveurs

Nous donnons ici un exemple de schéma. Nous avons choisi de considérer le problème de la garantie de l'exclusion des accès des processus d'un système asynchrone à des ressources partagées. K. Chandy et J. Misra ont proposé un paradigme pour ce problème bien connu : *le problème des philosophes buveurs* (drinking philosophers) [CM84]. Notons que le schéma proposé ici nous servira dans le chapitre suivant au cours de la construction des schémas de protocoles pour le rendez-vous.

L'énoncé du problème est le suivant [CM84] :

> On considère un réseau de processus appelés *philosophes*. A chaque philosophe est associé un ensemble non vide de ressources appelées *boissons*. Lorsqu'un philosophe accède à une de ces ressources on dit qu'il *boit* cette boisson. Un philosophe peut boire plusieurs boissons à la fois. Ces boissons peuvent être différentes chaque fois qu'il boit. Une relation de conflit binaire symétrique irréflexive sur les philosophes est donnée. Elle est appelée *relation de voisinage*.
> *Le problème des philosophes buveurs (drinking philosophers) consiste à décrire un protocole qui garantisse que deux philosophes voisins ne boivent pas une même boisson en même temps.*

Figure 3.2: Schéma d'un philosophe buveur ph

Constantes :
$voisins = \ldots$ tel que $\forall ph_j.\forall ph_k.(voisins(ph_j, ph_k) = voisins(ph_k, ph_j))$

$Bset_{ph} = \ldots$ tels que $Bset_{ph} \subseteq Boissons$

Fonctions d'état :

B_{ph} : set of Boissons

Conditions initiales :

$B_{ph} = \emptyset$

$INIT_{ph}$: forever do
CH_{ph} : $B_{ph} := ?$;
IP_{ph} : Protocole d'entrée (B_{ph}) ;
CS_{ph} : Section critique (B_{ph}) ;
OP_{ph} : Protocole de sortie (B_{ph})
od

Propriétés de sûreté :

unchanged B_{ph} when $\neg inCH_{ph}$ (ph-CH1)
$\neg inCH_{ph} \Rightarrow B_{ph} \subseteq Bset_{ph}$ (ph-CH2)
for all $ph_i : voisins(ph, ph_i) \Rightarrow \neg(in\,CS_{ph} \wedge in\,CS_{ph_i} \wedge B_{ph} \cap B_{ph_i} \neq \emptyset)$ (ph-CS)

Propriétés de vivacité :

$in\,IP_{ph} \rightsquigarrow \exists ph_i.in\,CS_{ph_i}$ (ph-SBLOC)
$in\,IP_{ph} \rightsquigarrow in\,CS_{ph}$ (ph-SFAM)

En général on suppose que les philosophes sont décrits par le schéma[3] de la Figure 3.2 où un philosophe ph choisit les boissons qu'il désire boire dans la partie CH_{ph} (ph-CH1, ph-CH2) et où il les boit dans la partie CS_{ph} appelée *section critique.* Lorsqu'un philosophe exécute son *protocole d'entrée* IP_{ph} il est dit *assoiffé* (thirsty) et lorsqu'il exécute sa section critique CS_{ph} il est dit *buvant* (drinking). Sinon il est dit *tranquille* (tranquil) [CM84].

Le problème des philosophes buveurs est donc de trouver une mise en œuvre des protocoles d'entrée et de sortie telle que la propriété de sûreté (ph-CS) est toujours satisfaite.

Les philosophes sont dits *bloqués* (deadlocked) quand aucun philosophe n'exécute sa section critique et aucun des philosophes qui désirent l'exécuter ne peut y parvenir. Si la mise en œuvre des protocoles d'entrée et de sortie satisfait, en plus

[3]Le programme considéré consiste simplement en un **cobegin** dont les processus sont les philosophes.

de (ph-CS), la propriété de vivacité (ph-SBLOC) alors la solution du problème des philosophes buveurs est dite *sans blocage* (deadlock free).

Enfin si les conflits ne sont pas toujours résolus au détriment d'un philosophe particulier, c'est-à-dire si, en plus de (ph-CS), la propriété de vivacité (ph-SFAM) est satisfaite la solution du problème des philosophes buveurs est dite *sans famine* (starvation free). Il est clair qu'une solution sans famine est aussi sans blocage.

Remarque: Deux cas particuliers du problème des philosophes buveurs ont été particulièrement étudiés : lorsque les philosophes boivent toujours les mêmes boissons (B_{ph} ne change pas en cours d'exécution) le problème est appelé *problème des philosophes mangeurs* (dining philosophers) [Dij71, CM84] et si de plus la relation de voisinage est totale le problème est appelé *problème de l'exclusion mutuelle* [Dij65]. □

3.3 Correction des protocoles

Les propriétés de sûreté et les propriétés de vivacité se prouvent grâce à des techniques différentes. Nous faisons quelques rappels en Sections 3.3.1 et 3.3.2 sur ces techniques. L'utilisation de ces techniques pour vérifier la correction des schémas est discutée en Section 3.3.3.

3.3.1 Preuve des propriétés de sûreté

Nous nous intéressons essentiellement ici à la preuve d'invariants. Un invariant a la forme suivante où *Init* et *Etern* sont des prédicats :

$$Init \Rightarrow \Box Etern \qquad \text{(Inv)}$$

Si le programme débute dans un état satisfaisant *Init* alors tout état atteint durant l'exécution satisfait *Etern*.

La méthode générale pour prouver (Inv) est de trouver une prédicat I tel que :

S1. $Init \Rightarrow I$

S2. Si le programme débute dans un état satisfaisant I alors tout état atteint durant l'exécution satisfait I

S3. $I \Rightarrow Etern$

Il est clair que S1–S3 impliquent (Inv). Notons aussi que S2 est équivalente à prouver la formule de logique temporelle $I \Rightarrow \Box I$ pour le programme.

S1 et S3 sont en général facile à vérifier car ce sont des propriétés statiques des prédicats. La propriété S2 est le cœur de la preuve car c'est une propriété du comportement du programme. Elle est prouvée en montrant que chaque instruction atomique, si elle débute dans un état satisfaisant I, terminera dans un état où I est vrai. Par induction, cela implique que I est un invariant. L'intérêt de cette approche est qu'elle considère chaque instruction atomique isolement et de ce fait ignore l'histoire du calcul.

3.3.2 Preuve des propriétés de vivacité

Nous présentons maintenant une technique de preuve des propriétés de vivacité des protocoles. Cette présentation est rapide et pour plus de détails nous renvoyons à [OL82].

Graphes de preuve

Supposons que les trois assertions suivantes soient vraies :

1. $P \leadsto (R1 \vee R2)$
2. $R1 \leadsto Q$
3. $R2 \leadsto Q$

Ces assertions signifient :

1. Si P est vraie à l'instant i alors $R1 \vee R2$ sera vraie à un instant $j \geq i$.
2. Si $R1$ est vraie à l'instant j alors Q sera vraie à un instant $k \geq j$.
3. Si $R2$ est vraie à l'instant j alors Q sera vraie à un instant $l \geq j$.

Nous pouvons montrer aisément qu'elles impliquent que $P \leadsto Q$ est vraie.

Ce raisonnement peut être décrit simplement par le *graphe de preuve* (proof

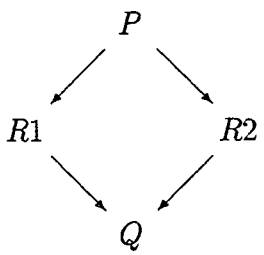


Figure 3.3: Graphe de preuve pour $P \leadsto Q$

lattice) de la Figure 3.3. Les deux flèches partant de P vers $R1$ et $R2$ dénotent l'assertion $P \rightsquigarrow R1 \vee R2$; La flèche de $R1$ vers Q dénote l'assertion $R1 \rightsquigarrow Q$; et la flèche de $R2$ vers Q dénote l'assertion $R2 \rightsquigarrow Q$.

Définition 3.1 (Graphe de preuve [OL82]) *Un graphe de preuve (proof lattice) pour un programme est un graphe fini acyclique dirigé dans lequel chaque nœud est étiqueté par une assertion tel que :*

1. *Il y a un seul nœud d'entrée (entry node) n'ayant pas d'arête entrante.*
2. *Il y a un seul nœud de sortie (exit node) n'ayant pas d'arête sortante.*
3. *Si un nœud étiqueté R a des arêtes sortantes vers des nœuds étiquetés R_1, R_2, ..., R_k alors $R \rightsquigarrow (R_1 \vee R_2 \vee \ldots \vee R_k)$ est vrai.*

La troisième condition signifie que si R est vrai à un instant alors un des R_i doit être vrai à un instant futur. En généralisant le raisonnement informel utilisé pour le graphe de la Figure 3.3, on voit aisément que si l'assertion du nœud d'entrée est vraie à un instant alors l'assertion du nœud de sortie doit être vraie à un instant futur. C'est ce qu'établi le théorème suivant (nous renvoyons à [OL82] pour sa preuve) :

Théorème 3.2 *Si il y a un graphe de preuve pour un programme avec un nœud d'entrée étiqueté P et un nœud de sortie étiqueté Q, alors $P \rightsquigarrow Q$ est vrai pour ce programme.*

Nous venons d'exposer une forme particulière de raisonnement utilisé pour prouver les propriétés de vivacité de la forme $P \rightsquigarrow Q$. Mais cela n'est pas suffisant pour prouver quelque chose sur un programme : nous avons besoin d'extraire des programmes leurs propriétés. Cela signifie qu'il nous faut donner une sémantique temporelle aux programmes, i.e. un ensemble de règles pour dériver des assertions vraies à propos du programme.

Progrès minimal

Les exécutions d'un programme respectent la propriété de Progrès minimal. Ce qui s'exprime ici par une seule règle : *les actions atomiques terminent toujours.* Cela signifie que si le programme atteint un état σ alors toute action atomique de $readyact(\sigma)$ sera exécutée au bout d'un temps fini. Et puisqu'il n'y a que trois sortes d'actions atomiques (les affectations et les tests du **while** et du **if**), le Progrès

minimal est exprimé par les trois axiomes[4] :

ATOMIC ASSIGNMENT AXIOM. Pour toute affectation atomique S :
$$at\,S \rightsquigarrow after\,S$$

if CONTROL FLOW AXIOM. Pour l'instruction c : **if** $\langle B \rangle$ **then** S **fi**
$$at\,c \rightsquigarrow (at\,S \vee after\,c)$$

while CONTROL FLOW AXIOM. Pour l'instruction c : **while** $\langle B \rangle$ **do** S **od**
$$at\,c \rightsquigarrow (at\,S \vee after\,c)$$

Règles sur le "control flow"

Avec une méthode pour prouver les propriétés de sûreté (Section 3.3.1), les trois axiomes précédents et les règles de la logique temporelle, nous pouvons dériver toutes les propriétés de vivacité que nous aimerions prouver pour le programme. Nous donnons maintenant une liste de règles d'inférence supplémentaires. Ces règles sont dérivées des axiomes précédents et de diverses propriétés de sûreté. Ces dérivations combinent essentiellement une propriété de sûreté "rien de mauvais ne peut arriver" avec les axiomes qui disent "quelque chose arrivera au bout d'un temps fini" pour conclure que "quelque chose de bon arrivera au bout d'un temps fini". La validité de ces règles est en général évidente et nous les présentons ici sans preuve. La dérivation de ces règles et la preuve de leur validité sont décrites dans [OL82] auquel nous renvoyons pour plus de détails.

Les propriétés de vivacité les plus simples parlent du *control flow* du programme : si le contrôle est à un point donné alors il doit atteindre au bout d'un temps fini un autre point donné. Les axiomes précédents ont cette forme : si le contrôle est au début d'une action atomique alors au bout d'un temps fini il sera après cette action. Nous dérivons de ces axiomes les règles sur le control flow suivantes :

CONCATENATION CONTROL FLOW RULE. Pour l'instruction $S;T$
$$\frac{at\,S \rightsquigarrow after\,S, \quad at\,T \rightsquigarrow after\,T}{at\,S \rightsquigarrow after\,T}$$

[4]Nous avons gardé les noms des axiomes et des règles de [OL82].

cobegin CONTROL FLOW RULE. Pour l'instruction c : **cobegin** $S \blacksquare T$ **coend**

$$\frac{at\,S \rightsquigarrow after\,S,\quad at\,T \rightsquigarrow after\,T}{at\,c \rightsquigarrow after\,c}$$

SINGLE EXIT RULE. Pour toute instruction S

$$in\,S \;\Rightarrow\; (\Box in\,S \vee \Diamond after\,S)$$

Notons que cette dernière règle n'est valide que parce que notre langage de programmation ne comporte pas d'instruction **goto**. Et donc, si le contrôle est dans S $(in\,S)$ il ne peut quitter S sans passer par le point de contrôle $after\,S$.

Nous avons aussi besoin de règles qui décrivent les relations entre le control flow et les valeurs des variables.

Si S est une instruction et P et Q sont des prédicats d'état alors la formule $\{P\}\ S\ \{Q\}$ a la signification suivante :

> Si l'exécution débute quelque part dans S avec P vrai alors l'exécution de la prochaine action atomique de S produira un nouvel état dans lequel
>
> - soit le contrôle est encore dans S et P est vrai,
> - soit le contrôle est après S et Q est vrai.

Nous déduisons des axiomes la règle :

ATOMIC STATEMENT RULE. Pour toute instruction atomique $\langle S \rangle$

$$\frac{\{P\}\ \langle S \rangle\ \{Q\},\quad \Box(at\,\langle S \rangle \;\Rightarrow\; P)}{at\,\langle S \rangle \rightsquigarrow (after\,\langle S \rangle \wedge Q)}$$

Nous étendons cette règle aux instructions non atomiques :

GENERAL STATEMENT RULE.

$$\frac{\{P\}\ S\ \{Q\},\quad \Box(in\,S \;\Rightarrow\; P),\quad in\,S \rightsquigarrow after\,S}{in\,S \rightsquigarrow (after\,S \wedge Q)}$$

Enfin nous pouvons dériver les trois règles suivantes traitant des tests atomiques des **if** et **while** :

if TEST RULE. Pour l'instruction c : **if** $\langle B \rangle$ **then** S **fi**

$$at\, c \rightsquigarrow ((at\, S \wedge B) \vee (after\, c \wedge \neg B))$$

while TEST RULE. Pour l'instruction c : **while** $\langle B \rangle$ **do** S **od**

$$at\, c \rightsquigarrow ((at\, S \wedge B) \vee (after\, c \wedge \neg B))$$

while EXIT RULE. Pour l'instruction c : **while** $\langle B \rangle$ **do** S **od**

$$(at\, c \wedge \Box(at\, c \Rightarrow B)) \rightsquigarrow at\, S$$
$$(at\, c \wedge \Box(at\, c \Rightarrow \neg B)) \rightsquigarrow after\, c$$

3.3.3 Correction des protocoles

La correction d'un schéma de protocoles se montre en utilisant les techniques décrites dans les sections précédentes. Les fonctions d'état de la spécification des entités abstraites sont utilisées dans la preuve comme des variables ordinaires d'un programme, et les transitions jouent le rôle d'instructions atomiques d'un programme. On construit les prédicats avec les variables du corps du schéma et les fonctions d'état de la spécification des entités. Les invariants sont prouvés en montrant que chaque instruction atomique du corps et chaque transition de la spécification du schéma garde vrai le prédicat. Le propriétés de vivacité sont prouvées à partir des règles de vivacité des instructions du corps du schéma et des propriétés de vivacité (axiomes) de la spécification du schéma.

Pour vérifier la correction d'un protocole qui implémente une spécification on doit définir les fonctions d'état de la spécification des entités abstraites comme des fonctions de l'état du protocole. C'est en général la partie la plus difficile de la preuve de correction. Ensuite la vérification est simple. Les propriétés de sûreté de la spécification sont vérifiées en montrant que chaque instruction atomique du protocole laisse inchangées toutes les fonctions d'état, ou sinon effectue une des transitions spécifiées (i.e. on doit montrer que les valeurs des fonctions d'état dans l'ancien et le nouvel état satisfont la spécification de la transition). Les propriétés de vivacité sont prouvées en utilisant les techniques de la Section 3.3.2.

3.4 Conclusions

Notre choix de la méthode Transition-Axiom pour spécifier les protocoles a été essentiellement motivé par sa simplicité. De même nous avons choisi d'utiliser la méthode Owicki-Lamport pour prouver la correction des protocoles car elle permet de développer des preuves informelles suffisamment claires et rigoureuses.

Nous n'avons fait qu'une présentation rapide de ces méthodes. Nous renvoyons à la lecture de [LS85, Lam89] et à [OL82] pour une présentation plus détaillée.

Chapitre 4

Construction des protocoles pour le rendez-vous et l'équité

Le plan de ce chapitre est le suivant : nous donnons en Section 4.1 une spécification des protocoles. Puis, en Sections 4.2 et 4.3, nous dérivons, de cette spécification, des schémas garantissant respectivement $\mathcal{WIF}$ et $\mathcal{SPF}$ (rendez-vous binaire), et nous discutons la mise en œuvre de protocoles à partir de ces schémas.

G. Buckley et A. Silberschatz [BS83] ont énoncé quatre critères permettant de juger la qualité d'un protocole pour le rendez-vous. Nous rappelons et discutons ces critères en Section 4.4.

4.1 Spécification des protocoles

Nous supposons un *système distribué* constitué d'un ensemble de *processus* p_1, p_2, ..., p_n, et d'un *scheduler distribué* DS chargé de contrôler les activités des processus afin d'assurer la cohérence du comportement global du système (Figure 4.1). La fonction essentielle du scheduler distribué est en fait de déterminer quelles sont les interactions qui peuvent être exécutées, compte tenu des contraintes de synchronisation, d'exclusion et d'équité, et de démarrer leur exécution. Nous supposons que le comportement des processus et celui du scheduler distribué sont

asynchrones.

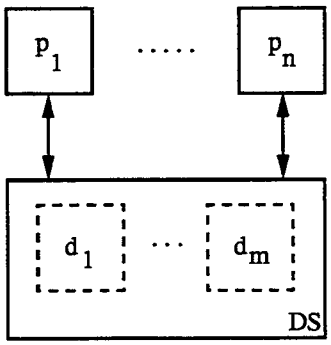


Figure 4.1: Processus et scheduler distribué

Définir un scheduler n'est pas en général une tâche aisée. La difficulté provient essentiellement du fait que le scheduler travaille avec une représentation approchée (locale) de l'état global du système. En général on suppose que le scheduler distribué est composé de *schedulers élémentaires* d_1, d_2, ..., d_m asynchrones, chargés chacun de contrôler un sous-ensemble des interactions : leur $Iset$. Le scheduler distribué devant contrôler l'exécution de *toutes* les interactions, l'union des Isets est bien sûr l'ensemble des interactions. Ou en d'autres termes, pour chaque interaction I il existe au moins un scheduler élémentaire d tel que $I \in Iset_d$.

Dans la suite de cette section nous spécifions le comportement des processus (Section 4.1.1) et celui des schedulers élémentaires (Section 4.1.2). Nous spécifions aussi les échanges d'informations nécessaires des schedulers vers les processus. Ces échanges d'informations sont effectués par l'*environnement* (Section 4.1.3).

4.1.1 Les processus

La spécification d'un processus p est donnée en Figure 4.2.

Nous supposons un ensemble fini de noms d'interactions noté Act. Lorsque nous parlerons d'une interaction I il s'agira d'une occurrence d'une interaction de nom I.

Nous notons $Proc$ l'ensemble des processus, et $p \in I$ le fait que p est un participant de l'interaction I.

Nous notons $\alpha[p]$ l'ensemble des actions atomiques qui peuvent être effectuées par le processus p.

p, p_i dénoteront des processus, d, d_i des schedulers et I, J, K des interactions.

Un processus est *actif* ($state = active$), ou bien est *prêt* ($state = idle$), ou bien participe à une interaction I ($state = commit^I$) (e.g. exécution du corps local

d'une interaction en IP). Lorsqu'il est actif, un processus exécute du code qui lui est local. Un processus prêt désire participer à une interaction parmi un ensemble d'interactions bien défini (*guard*). Cet ensemble d'interactions peut changer quand le processus est actif (p-S1) mais il est constant tant que le processus est prêt (p-S2 à p-S4). Le processus peut participer à I seulement si le scheduler distribué a démarré son exécution : le processus *sait* que le scheduler l'a fait si $Kstart[round, I]$ (p-S3).

Fonctions d'état :
$state_p$: { active, idle, commit^I | $I \in$ Act }
$round_p$: integer
$guard_p$: array [integer, Act] of boolean
$Kstart_p$: array [integer, Act] of boolean

Conditions initiales :
$state_p = active$
$round_p = 0$
for all r, I : $\neg guard_p[r, I]$
for all r, I : $\neg Kstart_p[r, I]$

Abréviations :
$active_p \equiv (state_p = active)$
$idle_p \equiv (state_p = idle)$
$idle_p^I \equiv (idle_p \wedge guard_p[round_p, I])$
$commit_p^I \equiv (state_p = commit^I)$

Propriétés de sûreté :
allowed changes to $state_p$
$round_p$
$guard_p$

for all $I : \alpha[p] : active_p \wedge p \in I \wedge \neg guard_p[round_p, I] \rightarrow guard_p[round_p, I]'$ (p-S1)

$\alpha[p] : active_p \rightarrow state'_p = idle$ (p-S2)

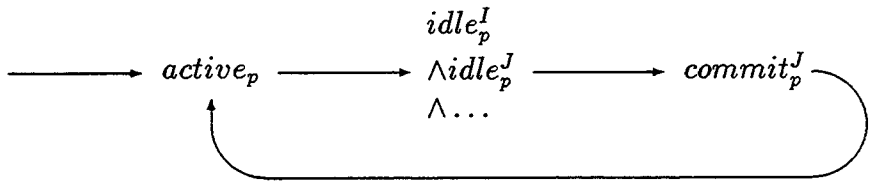
for all $I : \alpha[p] : idle_p^I \wedge Kstart_p[round_p, I] \rightarrow state'_p = commit^I$ (p-S3)

for all $I : \alpha[p] : commit_p^I \rightarrow state'_p = active \wedge round'_p = round_p + 1$ (p-S4)

Propriété de vivacité :
for all I : $idle_p^I \wedge Kstart_p[round_p, I] \rightsquigarrow commit_p^I$ (p-L1)

Figure 4.2: Spécification d'un processus p

Initialement tous les processus sont actifs ($active_p$). Un processus actif effectue de manière autonome (indépendante) la transition vers l'état prêt ($idle_p$) (p-S2). Un processus prêt reste prêt jusqu'à ce qu'il s'engage dans une interaction pour laquelle il est prêt ($commit_p^I$) (p-S3). Lorsqu'un processus est prêt pour une interaction et qu'il sait qu'elle est démarrée, il s'engage dans cette interaction au bout d'un temps fini (p-L1). Quand sa participation à une interaction I se termine (si elle se termine), le processus redevient actif (p-S4).



round est le nombre de cycles *active* → *idle* → *commit* → *active* effectués par le processus. Ce compteur permet de distinguer les différentes fois où le processus a été prêt pour une même interaction, et ainsi permet de distinguer les différentes fois où cette interaction a été exécutable ou exécutée.

4.1.2 Les schedulers élémentaires

Constantes :
$Iset_d = \ldots$ tels que $Act = \bigcup_d Iset_d$

Fonctions d'état :
$start_d$: array [Proc, integer, Act] of boolean

Conditions initiales :
for all $p, r, I : \neg start_d[p, r, I]$

Abréviations :
$enabled^I \equiv \forall p : p \in I.idle_p^I$
$ready^p \equiv \exists I : p \in I.enabled^I$

Propriétés de sûreté :
allowed changes to $start_d$
for all $I : \alpha[d] : I \in Iset_d \wedge enabled^I \wedge \exists p : p \in I.\neg start_d[p, round_p, I]$
$\rightarrow \forall p : p \in I.start_d[p, round_p, I]'$ (SYNC)
for all d_i, d_j, p, I, J, r : $\neg(p \in I \cap J \ \wedge I \neq J \wedge start_d[p, r, I] \wedge start_{d_j}[p, r, J])$ (EXCL)

Propriétés de vivacité :
$\forall I.(enabled^I \rightsquigarrow \neg enabled^I)$ (WIF)
ou
$\forall p.(\Box\Diamond ready^p \Rightarrow \Box\Diamond\exists I : p \in I.commit_p^I)$ (SPF)

Figure 4.3: Spécification d'un scheduler élémentaire d

A chaque scheduler est associé l'ensemble des interactions ($Iset$) qu'il est chargé de contrôler. Ces Isets sont tels que pour chaque interaction I il existe au moins un scheduler d tel que $I \in Iset_d$.

Un scheduler démarre une interaction I en mettant à *true* la partie de $start_d$ correspondante. Un scheduler ne peut démarrer une interaction que si tous ses participants sont prêts pour elle ($enabled^I$) (SYNC). Un scheduler ne démarre que les

interactions de son Iset (SYNC).

Un processus prêt désire participer à une seule des interactions pour lesquelles il est prêt. Les schedulers doivent donc coordonner leurs décisions de façon à ne pas démarrer des interactions différentes ayant un même participant (EXCL).

Nous voulons des schedulers qui garantissent que les exécutions du système vérifient $\mathcal{WIF}$ (WIF) ou $\mathcal{SPF}$ (SPF).

4.1.3 L'environnement

Nous attribuons à l'environnement tout ce qui concerne les échanges d'informations entre les processus et les schedulers, et les échanges des schedulers entre eux. La spécification de la Figure 4.4 décrit une seule propriété de sûreté nécessaire à la correction du système. (e-S1) garantit que les processus ne peuvent pas démarrer par eux-mêmes une interaction.

Figure 4.4: Spécification de l'environnement

Propriété de sûreté :
allowed changes to $Kstart_p$
for all $d, r, I : \alpha[env] : start_d[p, r, I] \wedge \neg Kstart_p[r, I] \rightarrow Kstart_p[r, I]' = true$ (e-S1)

4.1.4 Quelques invariants

Nous établissons dans cette section quelques invariants de cette spécification. Ces invariants permettent de nous convaincre que la spécification est "correcte", i.e. celle que nous voulons.

Invariant 1 *Un processus ne peut être prêt que pour les interactions auxquelles il peut participer.*

$$idle_p^I \Rightarrow p \in I$$

Idée de la preuve On montre grâce à (p-S1 à p-S4), (SYNC), (EXCL) et (e-S1) les invariants :

1. $idle_p^I \Rightarrow guard_p[round_p, I]$
2. $guard_p[round_p, I] \Rightarrow p \in I$

□

Invariant 2 *Un processus ne peut participer qu'à une interaction qui lui est attribué.*

$$commit_p^I \Rightarrow p \in I$$

Idée de la preuve On montre grâce à (p-S1 à p-S4), (SYNC), (EXCL) et (e-S1) les invariants :

1. $commit_p^I \Rightarrow guard_p[round_p, I]$
2. $guard_p[round_p, I] \Rightarrow p \in I$

□

Invariant 3 $Kstart_p[r, I] \Rightarrow guard_p[r, I]$

Idée de la preuve On montre grâce à (p-S1 à p-S4), (SYNC), (EXCL) et (e-S1) les invariants :

1. $Kstart_p[r, I] \Rightarrow \exists d.start_d[p, r, I]$
2. $start_d[p, r, I] \Rightarrow guard_p[r, I]$

□

4.2 Construction des protocoles pour $\mathcal{WIF}$

Dans cette section nous dérivons de la spécification précédente un schéma pour les schedulers qui garantit $\mathcal{WIF}$ (Section 4.2.1). Puis nous discutons la mise en œuvre de protocoles à partir de ce schéma (Section 4.2.2)

4.2.1 Dérivation d'un schéma

La dérivation du schéma se fait en plusieurs étapes. Nous proposons tout d'abord un schéma qui respecte la règle de synchronisation (SYNC). Ce schéma est ensuite transformé pour garantir, en plus de la synchronisation, l'exclusion (EXCL). Une troisième transformation permet de garantir $\mathcal{WIF}$. Enfin une dernière transformation tient compte de l'asynchronisme des processus et des schedulers.

Garantir la synchronisation

allowed changes to $start_d$
for all $I : \alpha[d] : I \in Iset_d \wedge enabled^I \wedge \exists p : p \in I.\neg start_d[p, round_p, I]$
$\rightarrow \forall p : p \in I.start_d[p, round_p, I]'$ (SYNC)

Figure 4.5: Schéma d'un scheduler d garantissant (SYNC)

Constantes :
$Iset_d = \ldots$ tels que $Act = \bigcup_d Iset_d$

Fonctions d'état :
I_d : $Act \cup \{NULL\}$
$start_d$: array [Proc, integer, Act] of boolean

Conditions initiales :
$I_d = NULL$
for all p, r, I : $\neg start_d[p, r, I]$

Abréviations :
$enabled^I \equiv \forall p : p \in I.idle_p^I$

$INIT_d$: forever do
CH_d : $I_d := ?$;
EX_d : Démarre exécution (I_d)
od

Propriétés de sûreté :
unchanged I_d when $\neg in\, CH_d$ (d-CH1)

$\neg in\, CH_d \Rightarrow I_d \in Iset_d$ (d-CH2)

allowed changes to $start_d$
for all $I : \alpha[d] : in\, EX_d \wedge I = I_d \wedge enabled^I \wedge \exists p : p \in I.\neg start_d[p, round_p, I]$
$\rightarrow after\, EX_d \wedge \forall p : p \in I.start_d[p, round_p, I]'$ (d-EX1)

Pour garantir (SYNC) il suffit à un scheduler de s'assurer avant de démarrer l'exécution de l'une des interactions de son Iset que tous les processus participants à l'interaction sont prêts pour elle.

Le schéma décrit en Figure 4.5 est un des schémas les plus simples qui puisse être proposé pour résoudre le problème de la synchronisation. Les schedulers répètent indéfiniment les actions suivantes :

1. choisir une interaction du Iset
2. si les participants à l'interaction sont tous prêts pour elle, démarrer son exécution

Le choix de l'interaction traitée, appelée I_d, se fait dans la partie CH_d et le démarrage de son exécution se fait dans EX_d. (d-CH1) spécifie que la valeur de I_d est constante en dehors de CH_d, et (d-CH2) spécifie que I_d est choisie parmi les interactions du Iset du scheduler d. Enfin (d-EX1) assure que I_d est démarrée seulement si tous ses participants sont prêts pour elle.

On montre aisément grâce à (d-CH2) et (d-EX1) que (SYNC) est respectée.

Garantir l'exclusion

Nous devons transformer le schéma précédent (Figure 4.5) pour que la règle d'exclusion (EXCL) soit respectée.

$$\text{for all } d_i, d_j, p, I, J, r: \ \neg(p \in I \cap J \ \wedge I \neq J \wedge start_d[p, r, I] \wedge start_{d_j}[p, r, J]) \quad \text{(EXCL)}$$

Pour assurer l'exclusion un scheduler doit, avant de démarrer une interaction I, être sûr qu'aucun autre scheduler n'a démarré (ou ne va démarrer en même temps que lui) une interaction différente ayant un participant en commun avec I.

Une première solution est d'interdire à tout scheduler d de démarrer l'exécution de I_d si une interaction différente, ayant un participant en commun avec I_d, a déjà été démarrée, c'est-à-dire de remplacer (d-EX1) par :

$executable^I \equiv (enabled^I \wedge \forall J : I \neq J \wedge I \cap J \neq \emptyset.\ \forall p : p \in I \cap J.\ \forall d.\ \neg start_d[p, round_p, J])$

allowed changes to $start_d$

for all $I : \alpha[d] : in\ EX_d \wedge I = I_d \wedge executable^I \wedge \exists p : p \in I.\neg start_d[p, round_p, I]$
$\rightarrow after\ EX_d \wedge \forall p : p \in I.start_d[p, round_p, I]'$ (d-EX2)

Mais (d-EX2) n'interdit pas que deux schedulers démarrent en même temps (dans la même transition) deux interactions différentes ayant un même participant. Or un démarrage se fait en exécutant la partie EX. Il faut donc aussi garantir que pour toute paire de schedulers (d_j, d_k) on a toujours :

$$\neg(in\ EX_{d_j} \wedge in\ EX_{d_k} \wedge I_{d_j} \neq I_{d_k} \wedge I_{d_j} \cap I_{d_k} \neq \emptyset)$$

Ce problème de l'exclusion des exécutions des blocs de code EX est clairement un cas particulier du problème des philosophes buveurs présenté en Section 3.2.3. Ici les schedulers sont les philosophes, les processus sont les boissons partagées, les interactions sont les ensembles de boissons, et EX est en section critique.

La relation de voisinage devra être telle que si deux schedulers d_j, d_k ont dans leurs Isets respectifs deux interactions différentes J et K ayant un participant commun ($J \in Iset_{d_j} \ \wedge \ K \in Iset_{d_k} \ \wedge \ J \neq K \ \wedge \ J \cap K \neq \emptyset$) alors ces deux schedulers sont voisins.

Le nouveau schéma des schedulers est décrit en Figure 4.6.

Figure 4.6: Schéma d'un scheduler d garantissant (SYNC) et (EXCL)

Constantes :
$voisins = \ldots$ tel que $\forall d_j.\forall d_k.(voisins(d_j, d_k) = voisins(d_k, d_j)$
$\wedge\ (\exists J.\exists K.J \in Iset_{d_j} \wedge K \in Iset_{d_k} \wedge J \neq K \wedge J \cap K \neq \emptyset) \Rightarrow voisins(d_j, d_k))$

$Iset_d = \ldots$ tels que $Act = \bigcup_d Iset_d$

Fonctions d'état :

$I_d : Act \cup \{NULL\}$
$start_d$: array [Proc, integer, Act] of boolean

Conditions initiales :

$I_d = NULL$
for all $p, r, I : \neg start_d[p, r, I]$

Abréviations :

$enabled^I \equiv \forall p : p \in I.idle_p^I$

$INIT_d$: forever do
CH_d : $I_d := ?$;
IP_d : Protocole d'entrée (I_d) ;
EX_d : Démarre exécution (I_d) ;
OP_d : Protocole de sortie (I_d)
od

$executable^I \equiv (enabled^I \wedge \forall J : I \neq J \wedge I \cap J \neq \emptyset.\ \forall p : p \in I \cap J.\ \forall d.\ \neg start_d[p, round_p, J])$

Propriétés de sûreté :
unchanged I_d when $\neg in\, CH_d$ (d-CH1)
$\neg in\, CH_d \Rightarrow I_d \in Iset_d$ (d-CH2)
for all $d_i : voisins(d, d_i) \Rightarrow \neg(in\, EX_d \wedge in EX_{d_i} \wedge I_d \cap I_{d_i} \neq \emptyset)$ (d-CS)
allowed changes to $start_d$
for all $I : \alpha[d] : in\, EX_d \wedge I = I_d \wedge executable^I \wedge \exists p : p \in I.\neg start_d[p, round_p, I]$
$\rightarrow after\, EX_d \wedge \forall p : p \in I.start_d[p, round_p, I]'$ (d-EX2)

On peut aisément montrer que (SYNC) est toujours satisfaite ((d-EX2) est plus restrictive que (d-EX1)). De plus (EXCL) est respectée par ce schéma. En effet, considérons un processus p. (d-EX2) spécifie :

1. qu'aucun scheduler ne démarre une interaction I à laquelle participe p si :
$$\exists d.\exists J.\ p \in I \cap J \wedge I \neq J \wedge start_d[p, round_p, J]$$

2. et on a pour toute transition $\sigma \xrightarrow{\alpha} \sigma'$:
$(\forall d.\neg(start_d[p, round_p, I] \vee start_d[p, round_p, J])$
$\wedge\ \exists d_i.\exists d_j.\ d_i \neq d_j \wedge start_{d_i}[p, round_p, I]' \wedge start_{d_j}[p, round_p, J]')$
$\Rightarrow\ \exists d_i.\exists d_j.\ d_i \neq d_j \wedge in\, EX_{d_i} \wedge in\, EX_{d_j} \wedge p \in I_{d_i} \cap I_{d_j}$
ce qui est contredit par l'invariant :

Invariant 4 $\forall d_j.\forall d_k.\neg(in\,EX_{d_j} \wedge in\,EX_{d_k} \wedge I_{d_j} \neq I_{d_k} \wedge I_{d_j} \cap I_{d_k} \neq \emptyset)$

Idée de la preuve Avec (d-CS) et la définition de *voisins* on montre :
$\forall d_j.\forall d_k.((\exists J.\exists K.\ J \in Iset_{d_j} \wedge K \in Iset_{d_k} \wedge J \neq K \wedge J \cap K \neq \emptyset)$
$\Rightarrow \neg(in\,EX_{d_j} \wedge in\,EX_{d_k} \wedge I_{d_j} \cap I_{d_k} \neq \emptyset))$
et on termine la preuve avec (d-CH2). □

Garantir $\mathcal{WIF}$

Les propriétés de sûreté sont respectées par le schéma de la Figure 4.6. Nous devons le transformer afin que (WIF) soit elle aussi garantie.

$$\boxed{\forall I.\ (enabled^I \leadsto \neg enabled^I) \qquad \text{(WIF)}}$$

(WIF) assure que si tous les participants à une interaction I sont prêts pour elle alors au bout d'un temps fini au moins un de ces processus participera à une interaction.

Pour garantir (WIF) il faut d'abords garantir que lorsqu'un scheduler a démarré une interaction, ses participants connaissent son démarrage au bout d'un temps fini, c'est-à-dire :

$$\boxed{\text{for all } p, r, I : \exists d.\ start_d[p, r, I] \leadsto Kstart_p[r, I] \qquad \text{(e-L1)}}$$

Avec cette propriété de vivacité et (p-L1) on montre :
$$\text{for all } I : \neg executable^I \leadsto \neg enabled^I$$
c'est-à-dire que lorsque un scheduler a démarré une interaction ses participants ne restent pas indéfiniment prêts pour elle.

Avant de démarrer une interaction de son Iset, un scheduler doit la choisir. Pour garantir qu'une interaction I ne restera pas indéfiniment "enabled", la solution la plus simple semble être de garantir que :

1. on ne peut avoir une interaction continuellement exécutable sans qu'elle soit choisie par un scheduler au bout d'un temps fini :

   (A) for all I : $executable^I \leadsto \neg executable^I \vee \exists d.(after\,CH_d \wedge I_d = I)$

2. puis démarrée au bout d'un temps fini :

   (B) for all d : $after\, CH_d \rightsquigarrow at\, EX_d$

   (C) for all d, I : $at\, EX_d \wedge I_d = I \wedge executable^I \rightsquigarrow after\, EX_d \wedge \neg executable^I$

Une manière sûre d'obtenir (B) est de choisir pour protocoles d'entrée (IP) et de sortie (OP) des solutions *sans famine* du problème des philosophes buveurs.

Une manière sûre d'obtenir (A) est de garantir que chaque scheduler effectuera, au bout d'un temps fini, sa partie CH

(D) for all d : $\neg in\, CH_d \rightsquigarrow in\, CH_d$

et qu'il ne pourra repousser indéfiniment le choix d'une interaction exécutable

for all d, I : $in\, CH_d \wedge I \in Iset_d \wedge executable^I$
$\rightsquigarrow \neg executable^I \vee (after\, CH_d \wedge I_d = I)$

IP/OP terminant au bout d'un temps fini (solution sans famine), il suffit maintenant, pour garantir (D), que l'exécution de la partie EX des schedulers termine toujours. C'est le cas si nous raffinons EX en :

EX_d : if T_d : $< executable^{I_d} >$
then
ST_d : Démarre exécution (I_d) ;
fi

avec (d-EX2) remplacée par (d-T1) et (d-EX3) :

Fonctions d'état :
r_d : array [Proc] of integer

Conditions initiales :
for all p : $r_d[p] = 0$

Propriétés de sûreté :
allowed changes to r_d
for all I : $\alpha[d]$: $in\, T_d \wedge I = I_d \rightarrow after\, T_d \wedge \forall p : p \in I.r_d[p]' = round_p$ (d-T1)

allowed changes to $start_d$
for all I : $\alpha[d]$: $in\, ST_d \wedge I = I_d \rightarrow after\, ST_d \wedge \forall p : p \in I.start_d[p, r_d[p], I]'$ (d-EX3)

Propriétés de vivacité :
for all I : $in\, ST_d \wedge I = I_d \rightsquigarrow after\, ST_d \wedge \forall p : p \in I.start_d[p, r_d[p], I]'$ (d-LST)

La fonction d'état r_d sert ici à mémoriser les valeurs des $round_p$ qui ont servi lors du test T_d. Elle est nécessaire car la valeur de $round_p$ peut changer entre le test d'exécutabilité et le démarrage de l'interaction (I_d peut aussi être démarrée par un

autre scheduler).

Il est clair que (d-EX2) est respectée et (C) est garantie. De plus EX termine toujours puisque T et ST terminent toujours (**if** control flow axiom).

Un schéma respectant les propriétés de sûreté et garantissant (WIF) est décrit en Figure 4.7.

Figure 4.7: Un schéma garantissant $\mathcal{WIF}$

Constantes :
$voisins = \ldots$ tel que $\forall d_j.\forall d_k.(voisins(d_j,d_k) = voisins(d_k,d_j)$
$\wedge\ (\exists J.\exists K.J \in Iset_{d_j} \wedge K \in Iset_{d_k} \wedge J \neq K \wedge J \cap K \neq \emptyset) \Rightarrow voisins(d_j,d_k))$
$Iset_d = \ldots$ tels que $Act = \bigcup_d Iset_d$

Fonctions d'état :
I_d : $Act \cup \{NULL\}$
$start_d$: array [Proc, integer, Act] of boolean
r_d : array [Proc] of integer

Conditions initiales :
$I_d = NULL$
for all p,r,I : $\neg start_d[p,r,I]$
for all p : $r_d[p] = 0$

Abréviations :
$enabled^I \equiv \forall p : p \in I.idle_p^I$
$executable^I \equiv (enabled^I \wedge \forall J : I \neq J \wedge I \cap J \neq \emptyset.\ \forall p : p \in I \cap J.\ \forall d.\ \neg start_d[p, round_p, J])$

Propriétés de sûreté :
unchanged I_d when $\neg in\,CH_d$ (d-CH1)
$\neg in\,CH_d \Rightarrow I_d \in Iset_d$ (d-CH2)
for all $d_i : voisins(d,d_i) \Rightarrow \neg(in\,EX_d \wedge inEX_{d_i} \wedge I_d \cap I_{d_i} \neq \emptyset)$ (d-CS)
allowed changes to r_d
for all $I : \alpha[d] : in\,T_d \wedge I = I_d \rightarrow after\,T_d \wedge\ \forall p : p \in I.r_d[p]' = round_p$ (d-T1)
allowed changes to $start_d$
for all $I : \alpha[d] : in\,ST_d \wedge I = I_d \rightarrow after\,ST_d \wedge\ \forall p : p \in I.start_d[p, r_d[p], I]'$ (d-EX3)

Propriétés de vivacité :
for all $I : in\,CH_d \wedge I \in Iset_d \wedge executable^I \leadsto \neg executable^I \vee (after\,CH_d \wedge I_d = I)$ (d-LCH)
$in\,IP_d \leadsto at\,EX_d$ (d-LIP)
for all $I : in\,ST_d \wedge I = I_d \leadsto after\,ST_d \wedge \forall p : p \in I.start_d[p, r_d[p], I]'$ (d-LST)
$in\,OP_d \leadsto after\,OP_d$ (d-LOP)
for all $p,r,I : \exists d.\ start_d[p,r,I] \leadsto Kstart_p[r,I]$ (e-L1)

Le graphe de preuve pour (WIF) est décrit en Figure 4.8 où :

1. $\dfrac{enabled^I \leadsto \neg enabled^I \vee \Box enabled^I}{\text{c'est une instance de la tautologie } P \leadsto \neg P \vee \Box P}$

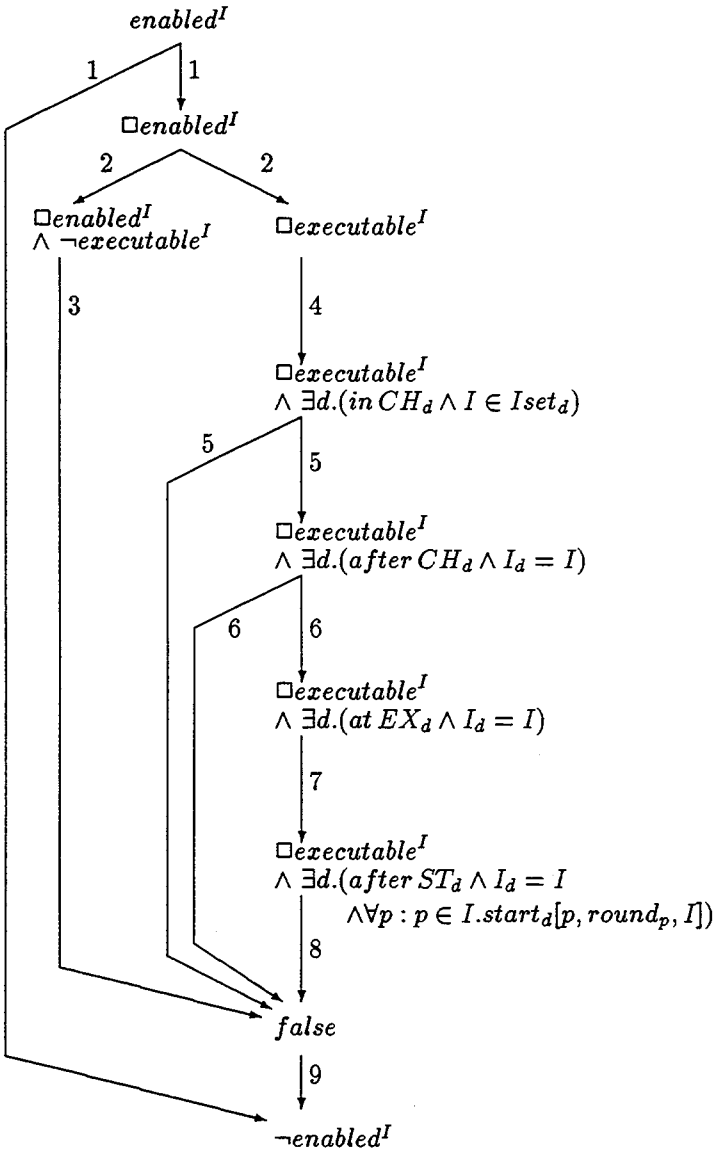
Figure 4.7: Un schéma garantissant $\mathcal{WIF}$

$INIT_d$: forever do
CH_d : $\boxed{I_d := ?}$;
IP_d : $\boxed{\text{Protocole d'entrée } (I_d)}$;
EX_d : if $T_d : < executable^{I_d} >$
then
ST_d : $\boxed{\text{Démarre exécution } (I_d)}$;
fi ;
OP_d : $\boxed{\text{Protocole de sortie } (I_d)}$
od

2. $\dfrac{\Box enabled^I \rightsquigarrow (\Box enabled^I \wedge \neg executable^I) \vee \Box executable^I}{\Box enabled^I \rightsquigarrow (\Box enabled^I \wedge \neg executable^I) \vee \Box(enabled^I \wedge executable^I)}$
est une instance de la tautologie $P \rightsquigarrow P \wedge (\neg Q \vee \Box Q)$
et par définition
$enabled^I \wedge executable^I \Leftrightarrow executable^I$

3. $\dfrac{\Box enabled^I \wedge \neg executable^I \rightsquigarrow false}{}$
par définition
$\neg executable^I \equiv \neg enabled^I$
$\vee \neg(\forall J : I \neq J \wedge I \cap J \neq \emptyset.\ \forall p : p \in I \cap J.\ \forall d.\ \neg start_d[p, round_p, J])$
trivialement
$\neg(\forall J : I \neq J \wedge I \cap J \neq \emptyset.\ \forall p : p \in I \cap J.\ \forall d.\ \neg start_d[p, round_p, J])$
$\Rightarrow\ \exists p : p \in I.\exists d.start_d[p, round_p, I]$
on montre grâce à (e-L1)
$\exists p : p \in I.\exists d.start_d[p, round_p, I] \rightsquigarrow \exists p : p \in I.Kstart_p[round_p, I]$
on montre grâce à (p-L1)
$\exists p : p \in I.Kstart_p[round_p, I] \rightsquigarrow \exists p : p \in I.commit_p^I$
et enfin
$\exists p : p \in I.commit_p^I \ \Rightarrow\ \neg enabled^I$

Figure 4.8: Graphe de preuve pour $\mathcal{WIF}$



4. $\Box executable^I \rightsquigarrow \Box executable^I \wedge \exists d.(in\,CH_d \wedge I \in Iset_d)$

on montre grâce à (d-LIP), (d-LST), (d-LOP) et (**if** control flow axiom)

$\neg in\,CH_d \rightsquigarrow in\,CH_d$

et par définition des Isets

$\forall I.\exists d.\ I \in Iset_d$

ces deux résultats permettent de prouver

$true \rightsquigarrow \exists d.(in\,CH_d \wedge I \in Iset_d)$

5. $\Box executable^I \wedge \exists d.(in\,CH_d \wedge I \in Iset_d) \rightsquigarrow false \vee (\Box executable^I \wedge \exists d.(after\,CH_d \wedge I_d = I))$
   se prouve grâce à (d-LCH) et
   $\neg executable^I \rightsquigarrow \neg enabled^I$ (voir 3)

6. $\Box executable^I \wedge \exists d.(after\,CH_d \wedge I_d = I) \rightsquigarrow false \vee (\Box executable^I \wedge \exists d.(at\,EX_d \wedge I_d = I))$
   se prouve grâce à (d-LIP) et
   $\neg executable^I \rightsquigarrow \neg enabled^I$ (voir 3)

7. $\Box executable^I \wedge \exists d.(at\,EX_d \wedge I_d = I) \rightsquigarrow \Box executable^I \wedge \exists d.(after\,ST_d \wedge I_d = I \wedge \forall p : p \in I.start_d[p, round_p, I])$
   se prouve grâce à (d-T1), (d-LST) et (**if** control flow axiom)

8. $\Box executable^I \wedge \exists d.(after\,ST_d \wedge I_d = I \wedge \forall p : p \in I.start_d[p, round_p, I]) \rightsquigarrow false$
   se prouve grâce à (e-L1) et (p-L1)

9. $false \rightsquigarrow \neg enabled^I$
   trivialement

Un schéma pour des schedulers asynchrones

Le schéma des schedulers spécifié en Figure 4.7 garantit $\mathcal{WIF}$. Mais nous pouvons remarquer que le test $< executable^{I_d} >$ nécessite de la part des schedulers une connaissance précise et immédiate de l'état des processus et des autres schedulers. Or les comportements des processus et des schedulers sont asynchrones. Nous modifions le schéma pour prendre en compte cet asynchronisme.

Par définition

$$executable^I \equiv \forall p : p \in I.(idle_p \wedge guard_p[round_p, I]) \wedge \forall J : I \neq J \wedge I \cap J \neq \emptyset.\ \forall p : p \in I \cap J.\ \forall d.\ \neg start_d[p, round_p, J]$$

Cette écriture met en évidence les informations nécessaires au scheduler sur l'état des processus ($idle_p$ et $guard_p[round_p]$) et sur l'état des autres schedulers ($start$). Nous devons faire apparaître dans le schéma le fait que le scheduler n'a qu'une connaissance locale et différée de ces informations. Nous traitons d'abord le cas des informations sur l'état des processus.

Nous supposerons que les fonctions d'état $Knumidle_d$ et $Kguard_d$ décrivent la connaissance du scheduler d sur l'état des processus. Ces fonctions d'état sont spécifiées ainsi :

Fonctions d'état :
$Knumidle_d$: array [Proc] of integer
$Kguard_d$: array [Proc, integer, Act] of boolean

Conditions initiales :
for all p : $Knumidle_d[p] = 0$
for all p, r, I : $\neg Kguard_d[p, r, I]$

Propriétés de sûreté :
allowed changes to $Knumidle_d$
for all p : $\alpha[env]$: $idle_p \wedge Knumidle_d[p] < round_p$
$\rightarrow Knumidle_d[p]' = Knumidle_d[p] + 1$ (e-S2)
for all p, I : $Knumidle_d[p] = round_p \Rightarrow Kguard_d[p, round_p, I] = guard_p[round_p, I]$ (e-S3)

$Knumidle_d[p]$ est le nombre de fois (connues par le scheduler d) où le processus p est devenu prêt. (e-S2) garantit que le scheduler n'oublie ni n'invente aucun de ces changements d'état des processus. $Knumidle_d[p]$ contient des informations sur à la fois $state_p$ et $round_p$ ainsi que le montre l'invariant suivant :

Invariant 5 $Knumidle_d[p] = numstart^p + 1 \Rightarrow idle_p \wedge Knumidle_d[p] = round_p$
où $numstart^p \equiv max\{r \mid \exists d. \exists I. start_d[p, r, I]\}$

Idée de la preuve Grâce à (e-S1), (e-S2) et (p-S1 à p-S4) on montre
$Knumidle_d[p] \leq round_p \leq numstart^p + 1$
et en examinant les changements d'état des processus
$Knumidle_d[p] = numstart^p + 1 \Rightarrow idle_p$ □

$Kguard_d[p]$ décrit ce que d sait à propos de $guard_p$. (e-S3) garantit que lorsque $Knumidle_d$ est à jour $Kguard_d$ l'est aussi.

Chaque scheduler a aussi besoin de connaître (partiellement) l'état (*start*) des autres schedulers. Nous supposerons que la fonction d'état $Kstart_d$ décrit cette connaissance. Cette fonction d'état est spécifiée ainsi :

Fonctions d'état :
$Kstart_d$: array [Proc, integer, Act] of boolean

Conditions initiales :
for all $p, r, I : \neg Kstart_d[p, r, I]$

Abréviations :
$Knumstart_d^p \equiv max\{r \mid \exists I.Kstart_d[p, r, I]\}$
$Kenabled_d^I \equiv \forall p : p \in I.(Knumidle_d[p] = Knumstart_d^p + 1 \wedge Kguard_d[p, Knumidle_d[p], I])$
$Kexecutable_d^I \equiv (Kenabled_d^I \wedge \forall J : I \neq J \wedge I \cap J \neq \emptyset.\forall p : p \in I \cap J.\neg Kstart_d[p, Knumidle_d[p], J])$

Propriétés de sûreté :
allowed changes to $Kstart_d$
for all $p, r, I : \alpha[env] : \exists d_i.start_{d_i}[p, r, I] \wedge \neg Kstart_d[p, r, I] \rightarrow Kstart_d[p, r, I]$ (e-S4)
for all $I : (in\,EX_d \wedge I_d = I) \Rightarrow$
$\forall J : I \cap J \neq \emptyset.\ \forall p : p \in I \cap J.\ (\exists d_i.start_{d_i}[p, round_p, J] \Rightarrow Kstart_d[p, round_p, J])$ (e-S5)

(e-S4) garantit que le scheduler n'oublie ni n'invente aucun démarrage d'interaction. Et (e-S5) garantit que, lorsque le scheduler effectue sa partie EX (qui contient le test d'exécutabilité), sa connaissance sur les démarrages de toutes les interactions ayant des participants communs avec I_d est à jour.

Invariant 6 *for all* $I : (in\,EX_d \wedge I_d = I) \Rightarrow (Kexecutable_d^I \Rightarrow executable^I)$

Idée de la preuve Grâce à (e-S5) on montre :
for all $I : (in\,EX_d \wedge I_d = I) \Rightarrow \forall p : p \in I.(Knumstart_d^p = numstart^p)$
ce qui nous permet de prouver
for all $I : in\,EX_d \wedge I_d = I \Rightarrow (Kenabled_d^I \Rightarrow enabled^I)$
puis l'invariant. □

Il ne faut pas qu'une interaction du Iset d'un scheduler d reste continuellement exécutable sans que le scheduler soit prévenu que ses participants sont prêts pour elle. C'est ce qu'exprime la propriété de vivacité suivante :

for all $I : executable^I \wedge I \in Iset_d$
$\rightsquigarrow \neg executable^I \vee \forall p : p \in I.(Knumidle_d[p] = round_p)$ (e-L2)

L'invariant et la propriété de vivacité précédents nous permettent de remplacer dans le schéma le test $< executable^{I_d} >$ par $< Kexecutable_d^{I_d} >$ qui n'utilise que les connaissances locales et différées du scheduler sur l'état des processus et des autres schedulers. Ainsi nous pouvons transformer le schéma précédent en un schéma qui garantit toujours $\mathcal{WIF}$ mais qui de plus tient compte de l'asynchronisme des processus et des schedulers. Ce nouveau schéma est décrit en Figure 4.9.

Figure 4.9: Schéma d'un scheduler d garantissant $\mathcal{WIF}$

Constantes :

$voisins = \ldots$ tel que $\forall d_j.\forall d_k.(voisins(d_j,d_k) = voisins(d_k,d_j)$
$\wedge\ (\exists J.\exists K.J \in Iset_{d_j} \wedge K \in Iset_{d_k} \wedge J \neq K \wedge J \cap K \neq \emptyset) \Rightarrow voisins(d_j,d_k))$

$Iset_d = \ldots$ tels que $Act = \bigcup_d Iset_d$

Fonctions d'état :

I_d : $Act \cup \{NULL\}$
$start_d$: array [Proc, integer, Act] of boolean
r_d : array [Proc] of integer
$Kstart_d$: array [Proc, integer, Act] of boolean
$Knumidle_d$: array [Proc] of integer
$Kguard_d$: array [Proc, integer, Act] of boolean

Conditions initiales :

$I_d = NULL$
for all p,r,I : $\neg start_d[p,r,I]$
for all p : $r_d[p] = 0$
for all p,r,I : $\neg Kstart_d[p,r,I]$
for all p : $Knumidle_d[p] = 0$
for all p,r,I : $\neg Kguard_d[p,r,I]$

Abréviations :

$enabled^I \equiv \forall p : p \in I.idle_p^I$
$executable^I \equiv (enabled^I \wedge \forall J : I \neq J \wedge I \cap J \neq \emptyset.\ \forall p : p \in I \cap J.\ \forall d.\ \neg start_d[p, round_p, J])$
$Knumstart_d^p \equiv max\{r \mid \exists I.Kstart_d[p,r,I]\}$
$Kenabled_d^I \equiv \forall p : p \in I.(Knumidle_d[p] = Knumstart_d^p + 1 \wedge Kguard_d[p, Knumidle_d[p], I])$
$Kexecutable_d^I \equiv (Kenabled_d^I \wedge$
$\forall J : I \neq J \wedge I \cap J \neq \emptyset.\forall p : p \in I \cap J.\neg Kstart_d[p, Knumidle_d[p], J])$

Propriétés de sûreté :

unchanged I_d when $\neg in\,CH_d$ (d-CH1)
$\neg in\,CH_d \Rightarrow I_d \in Iset_d$ (d-CH2)
for all d_i : $voisins(d,d_i) \Rightarrow \neg(in\,EX_d \wedge inEX_{d_i} \wedge I_d \cap I_{d_i} \neq \emptyset)$ (d-CS)
allowed changes to r_d
for all $I : \alpha[d] : in\,T_d \wedge I = I_d \rightarrow after\,T_d \wedge\ \forall p : p \in I.r_d[p]' = Knumidle_d[p]$ (d-T2)
allowed changes to $start_d$
for all $I : \alpha[d] : in\,ST_d \wedge I = I_d \rightarrow after\,ST_d \wedge\ \forall p : p \in I.start_d[p, r_d[p], I]'$ (d-EX3)

Propriétés de vivacité :

for all $I : in\,CH_d \wedge I \in Iset_d \wedge executable^I \rightsquigarrow \neg executable^I \vee (after\,CH_d \wedge I_d = I)$ (d-LCH)
$in\,IP_d \rightsquigarrow at\,EX_d$ (d-LIP)
for all $I : in\,ST_d \wedge I = I_d \rightsquigarrow after\,ST_d \wedge \forall p : p \in I.start_d[p, r_d[p], I]'$ (d-LST)
$in\,OP_d \rightsquigarrow after\,OP_d$ (d-LOP)

Figure 4.9: Schéma d'un scheduler d garantissant $\mathcal{WIF}$

$INIT_d$: forever do

CH_d : $\boxed{I_d := ?}$;

IP_d : $\boxed{\text{Protocole d'entrée } (I_d)}$;

EX_d : if $T_d : < Kexecutable_d^{I_d} >$
then

ST_d : $\boxed{\text{Démarre exécution } (I_d)}$;

fi ;

OP_d : $\boxed{\text{Protocole de sortie } (I_d)}$

od

Figure 4.9: Schéma d'un scheduler d garantissant $\mathcal{WIF}$

Propriétés de sûreté :

allowed changes to $Knumidle_d$

for all $p : \alpha[env] : idle_p \wedge Knumidle_d[p] < round_p \rightarrow Knumidle_d[p]' = Knumidle_d[p] + 1$ (e-S2)

for all $p, I : Knumidle_d[p] = round_p \Rightarrow Kguard_d[p, round_p, I] = guard_p[round_p, I]$ (e-S3)

allowed changes to $Kstart_d$

for all $p, r, I : \alpha[env] : \exists d_i.start_{d_i}[p, r, I] \wedge \neg Kstart_d[p, r, I] \rightarrow Kstart_d[p, r, I]$ (e-S4)

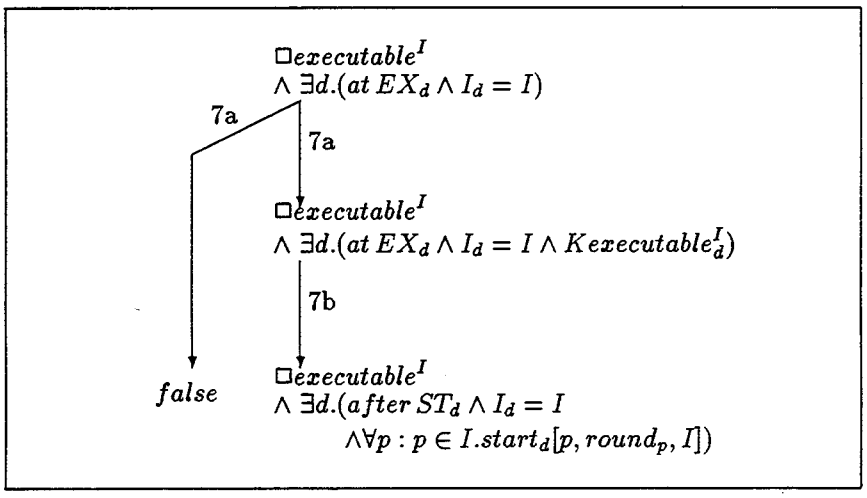
for all $I : (in\, EX_d \wedge I_d = I) \Rightarrow$
$\forall J : I \cap J \neq \emptyset.\ \forall p : p \in I \cap J.\ (\exists d_i.start_{d_i}[p, round_p, J] \Rightarrow Kstart_d[p, round_p, J])$ (e-S5)

Propriétés de vivacité :

for all $p, r, I : \exists d.\ start_d[p, r, I] \leadsto Kstart_p[r, I]$ (e-L1)

for all $I : executable^I \wedge I \in Iset_d \leadsto \neg executable^I \vee \forall p : p \in I.(Knumidle_d[p] = round_p)$ (e-L2)

On montre aisément que les propriété de sûreté (SYNC) et (EXCL) sont toujours respectées par le schéma de la Figure 4.9. Le graphe de preuve pour (WIF) est similaire à celui décrit en Figure 4.8 où seule la branche 7 est remplacée par :



7a. $\Box executable^I \wedge \exists d.(at\,EX_d \wedge I_d = I)$
$\rightsquigarrow false \vee \Box executable^I \wedge \exists d.(at\,EX_d \wedge I_d = I \wedge Kexecutable_d^I)$

Grâce à (e-S3) et (e-L2) on montre
$\Box executable^I \wedge \exists d.(at\,EX_d \wedge I_d = I \wedge \forall p : p \in I.(Knumidle_d[p] = round_p$
$\wedge\ Kguard_d[p, round_p, I] = guard_p[round_p, I]))$
puis grâce à l'invariant $executable^I \Rightarrow \forall p : p \in I.round_p = numstart^p + 1$
et à (e-S5) on montre
$\Box executable^I \wedge \exists d.(at\,EX_d \wedge I_d = I$
$\wedge \forall p : p \in I.(Kguard_d[p, round_p, I] = guard_p[round_p, I]$
$\wedge\ Knumidle_d[p] = round_p = numstart^p + 1 = Knumstart_d^p + 1))$
et enfin on trouve le résultat grâce à l'invariant
$executable^I \Rightarrow \forall p : p \in I.\neg Kstart_d[p, round_p, I]$

7b. $\Box executable^I \wedge \exists d.(at\,EX_d \wedge I_d = I \wedge Kexecutable_d^I)$
$\rightsquigarrow \Box executable^I \wedge \exists d.(after\,ST_d \wedge I_d = I \wedge \forall p : p \in I.start_d[p, round_p, I])$

se prouve d'une manière similaire à 7.

4.2.2 Mise en œuvre des protocoles

Pour mettre en œuvre un protocole à partir du schéma de la Figure 4.9, il suffit de remplacer les parties abstraites du schéma par du code qui respecte les propriétés de sûreté et de vivacité exigées. Nous discutons dans cette section le choix de ce code en donnant des exemples de techniques employées dans divers algorithmes publiés.

- **Choix des Isets :**

La répartition des interactions entre les schedulers constitue la première étape de construction d'un protocole. La seule contrainte imposée sur cette répartition est que chaque interaction soit associée à au moins un scheduler.

Cette répartition est simplifiée lorsqu'il n'y a qu'un seul scheduler élémentaire. L'utilisation d'un seul scheduler permet d'obtenir un *contrôle centralisé* des exécutions des interactions. C'est pourquoi une telle solution est dite *centralisée* [Bag89a]. Plusieurs auteurs ont proposé des solutions centralisées (e.g. [CM88], [Lev88], [Bag89a]). Elles ont l'avantage de pouvoir être implémentées aisément (e.g. pas besoin de protocoles d'entrée et sortie) mais en général leurs performances sont faibles.

L'étude des protocoles publiés (voir Bibliographie) utilisant un *contrôle distribué* (plusieurs schedulers élémentaires) montre qu'ils peuvent être quasiment tous classés en deux groupes en fonction du choix des Isets.

Les protocoles constituants le premier groupe sont caractérisés par le fait que les Isets des schedulers sont des singletons, c'est-à-dire que les schedulers sont en charge d'une et une seule interaction (e.g. [CM88]). Nous appelons *schedulers d'interaction* les schedulers élémentaires de ces protocoles.

Le second groupe est constitué de protocoles qui utilisent des *schedulers de processus* (e.g. [Ram87a]). A chaque scheduler est associé un processus. Le Iset d'un scheduler est constitué par l'ensemble des interactions auxquelles peut participer son processus associé. Les schedulers sont donc chargés, lorsque leur processus associé est prêt, de décider la participation du processus à une interaction.

D'autres manières de répartir les interactions ont été proposées : par exemple on peut associer à chaque scheduler un Iset composé uniquement d'interactions qui ont deux à deux des participants communs (e.g. [Bag89a]).

Nous discutons dans la suite de cette section, pour chacune des parties du protocole, des avantages et des inconvénients de ces différentes répartitions.

- **Choix de l'interaction traitée :**

Le code de la partie CH doit être tel que l'interaction traitée (I_d) est choisie parmi les interactions du Iset (d-CH2), et si une interaction du Iset est continuellement exécutable alors elle sera choisie au bout d'un temps fini (d-LCH).

Ce choix est simplifié lorsque les schedulers sont des schedulers d'interaction : il n'y a qu'une seule interaction dans le Iset. Certains protocoles supposent que les schedulers d'interaction essaient systématiquement d'entrer dans leur section critique (pas de code pour CH) (e.g. algorithme 2a de [Lev88]). D'autres protocoles supposent que les schedulers attendent que tous les processus participant à leur interaction associée soient prêts pour elle (à la connaissance du scheduler) avant d'effectuer le protocole d'entrée (e.g. [CM88]).

En général les protocoles à base de schedulers de processus supposent que les schedulers attendent que leur processus associé soit prêt puis choisissent I_d parmi les interactions auxquelles le processus désire participer (e.g. [Ram87a]).

- **Protocoles d'entrée et de sortie :**

Les protocoles d'entrée et de sortie sont choisis parmi les solutions sans famine du problème des philosophes buveurs (d-CS, d-LIP, d-LOP).

La relation de voisinage sur les schedulers doit être telle que :
$\forall d_j.\forall d_k.((\exists J.\exists K.J \in Iset_{d_j} \wedge K \in Iset_{d_k} \wedge J \neq K \wedge J \cap K \neq \emptyset) \Rightarrow voisins(d_j, d_k))$

Cette contrainte donne seulement une borne inférieure pour cette relation de voisinage. On peut donc utiliser sans difficulté des protocoles solution du problème des philosophes mangeurs (e.g. [CM88]) ou du problème de l'exclusion mutuelle (e.g. [Bag89a]).

En particulier lorsque les schedulers sont des schedulers d'interaction on pourra utiliser des protocoles solution du problème des philosophes mangeurs avec une relation de voisinage telle que :
$\forall d_j.\forall d_k.((\exists J.\exists K.J \in Iset_{d_j} \wedge K \in Iset_{d_k} \wedge J \neq K \wedge J \cap K \neq \emptyset) \Leftrightarrow voisins(d_j, d_k))$
sans restreindre la concurrence des démarrages des interactions qui n'ont pas de participant commun.

De même lorsque toutes les interactions du système ont deux à deux des participants communs on peut utiliser, sans restreindre la concurrence des démarrages, des protocoles solution du problème de l'exclusion mutuelle car la relation de voisinage est nécessairement totale.

Notons que l'utilisation de solutions du problème de l'exclusion mutuelle permet au scheduler en section critique d'avoir un accès exclusif à tous les processus du système. Le choix de l'interaction traitée peut alors se faire en section critique. Cela est même nécessaire lorsqu'on utilise certains algorithmes pour l'exclusion mutuelle exigeant que la décision d'entrer en section critique soit systématique (pas de code avant IP). C'est le cas des algorithmes à la [Dij74] basés sur la circulation d'un privilège sur un anneau : le scheduler ne peut ni réclamer le privilège ni le refuser (le premier protocole pour le rendez-vous de [Bag89a] utilise un tel algorithme).

Remarque: Le protocole décrit dans [BHKS85] n'utilise pas explicitement de protocoles d'entrée et de sortie. Le support de communication (niveau machine) utilisé dans ce protocole est la diffusion synchrone (synchronous broadcast) de messages. L'exclusion est réalisée en utilisant le fait que l'accès au canal (channel) de diffusion est exclusif : lorsque plusieurs schedulers veulent en même temps utiliser le canal un seul scheduler à la fois peut le faire. □

- **Démarrage de l'interaction :**

  Le codage du démarrage de l'interaction ne pose pas de problèmes particuliers.

- **L'environnement :**

  Au cours d'une exécution du système les informations sur les changements d'état (actif ↔ prêt) et les démarrages d'interaction sont les seules informations qui ont besoin d'être échangées entre les processus et les schedulers, et les informations sur les accès à leur section critique et les démarrages d'interaction sont les seules informations qui ont besoin d'être échangées par les schedulers entre eux.

  Les échanges entre les schedulers et les processus doivent respecter (e-S1), (e-S2), (e-S3), (e-L1) et (e-L2). Les échanges des schedulers entre eux doivent respecter (e-S4) et (e-S5).

  Les techniques employées pour réaliser ces échanges d'informations dépendent essentiellement du support (niveau machine) de la communication : variables propres[1] (e.g. [Sch78]), échange (point-à-point) synchrone/asynchrone de messages (e.g. [Bor86], [Bag89a]), diffusion (multicast) de messages (e.g. [BHKS85]) ...

  D'un point de vue plus abstrait on peut distinguer deux modes d'échange d'informations :

  le mode observation : Chaque information est diffusée aux différents processus ou schedulers intéressés. Les schedulers mémorisent les informations qu'ils reçoivent : ces informations constituent leur connaissance de l'état global du système.

  Exemple: le premier algorithme de [Bag89a] suppose que chaque fois qu'un processus devient prêt il envoie un message *ready* aux schedulers concernés. Les schedulers s'informent mutuellement des démarrages d'interaction via un jeton qui passe par tous les schedulers, et informent les processus participants par un message *execut*. □

  le mode interrogation (polling) : Dans ce mode les informations ne sont pas diffusées. Elles sont au contraire mémorisées par ceux qui en ont connaissance

[1] Une *variable propre* à un processus est une variable qui peut être modifiée seulement par ce processus mais qui peut être lue par tous les processus du réseau [Ray84].

et diffusées à la demande. Par exemple supposons qu'un processus actif devienne prêt. Il faut qu'un scheduler demande au processus (question) quel est son état pour que ce scheduler ait connaissance (réponse) du changement d'état.

Exemple: l'algorithme de [Ram87a] suppose que chaque fois qu'un processus devient prêt le scheduler de processus associé envoi des requêtes aux autres schedulers, les réponses reçues décrivant l'état des processus associés aux autres schedulers. □

L'utilisation du mode interrogation peut donner lieu à des réalisations qui dans le pire des cas produisent un nombre non borné de messages question/réponse. C'est le cas lorsque le code de CH ou de IP/OP comporte une condition d'attente active durant laquelle le scheduler s'informe sur l'état des processus ou des autres schedulers (e.g. [Sch78]).

4.3 Construction des protocoles pour $\mathcal{SPF}$ (rendez-vous binaire)

Dans cette section nous dérivons de la spécification (Section 4.1) un schéma pour les schedulers qui garantit $\mathcal{SPF}$ dans le cas du rendez-vous binaire (Section 4.3.1). Puis nous discutons la mise en œuvre de protocoles à partir de ce schéma (Section 4.3.2)

4.3.1 Dérivation d'un schéma

Un schéma garantissant $\mathcal{SPF}$ garantira aussi $\mathcal{WIF}$. Nous venons de montrer que $\mathcal{WIF}$ est garantie par le schéma de la Figure 4.9. Nous le transformons, dans cette section, pour que $\mathcal{SPF}$ soit elle aussi respectée.

$$\forall p.\ (\Box\Diamond ready^p \Rightarrow \Box\Diamond\exists I : p \in I.\ commit_p^I) \qquad \text{(SPF)}$$

(SPF) assure que, pour chaque processus p, si p est infiniment souvent prêt à interagir et si cela est infiniment souvent possible (i.e. si infiniment souvent il existe une interaction de p exécutable) alors p interagira infiniment souvent.

Pour garantir (SPF) il faut donc faire en sorte que le choix par les schedulers des interactions à démarrer ne se fassent pas toujours au détriment d'un processus particulier. Pour qu'un scheduler choisisse de démarrer une interaction exécutable ou de reporter ce démarrage il lui faut un critère de choix. Nous supposerons que le choix du scheduler d est dirigé par la valeur de la fonction d'état booléenne $postponed_d$: le scheduler choisira de ne pas démarrer immédiatement l'interaction I exécutable si $postponed_d[I]$. Cette fonction d'état est spécifiée ainsi :

Fonction d'état :
$postponed_d$: array [Act] of boolean

Conditions initiales :
for all I : $postponed_d[I] \in \{true, false\}$

Propriétés de vivacité :
for all I : $\Diamond\Box Kexecutable_d^I \Rightarrow \Diamond\Box(\neg(at\, EX_d \wedge I_d = I) \vee \neg postponed_d[I])$ (d-SP1)

for all p, I : $\Diamond\Box idle_p^I \wedge \Box\Diamond enabled^I \wedge \Box\Diamond\neg enabled^I \Rightarrow$
$\forall J : I \cap J = \{\tilde{p}\} \wedge p \neq \tilde{p}.\ \Diamond\Box(\neg(at\, EX_d \wedge I_d = J) \vee \neg Kexecutable_d^J \vee postponed_d[J])$ (d-SP2)

La valeur de $postponed_d$ dans l'état initial peut être quelconque. (d-SP1) garantit que $\mathcal{WIF}$ est encore respectée. Et c'est (d-SP2) qui permet de garantir $\mathcal{SPF}$. En effet (d-SP2) assure que si un processus p est continuellement prêt alors ses partenaires possibles ne peuvent pas être à la fois infiniment souvent prêts à interagir avec lui et infiniment souvent interagir avec un autre processus.

Le nouveau schéma obtenu est décrit en Figure 4.10.

Figure 4.10: Schéma d'un scheduler d garantissant $\mathcal{SPF}$ (rendez-vous binaire)

$INIT_d$: forever do
CH_d : [$I_d := ?$] ;
IP_d : [Protocole d'entrée (I_d)] ;
EX_d : if $T_d : < Kexecutable_d^{I_d} \wedge \neg postponed_d[I_d] >$
then
ST_d : [Démarre exécution (I_d)] ;
fi ;
OP_d : [Protocole de sortie (I_d)]
od

Figure 4.10: Schéma d'un scheduler d garantissant $\mathcal{SPF}$ (rendez-vous binaire)

Constantes :

$voisins = \ldots$ tel que $\forall d_j.\forall d_k.(voisins(d_j,d_k) = voisins(d_k,d_j)$
$\wedge\ (\exists J.\exists K.J \in Iset_{d_j} \wedge K \in Iset_{d_k} \wedge J \neq K \wedge J \cap K \neq \emptyset) \Rightarrow voisins(d_j,d_k))$

$Iset_d = \ldots$ tels que $Act = \bigcup_d Iset_d$

Fonctions d'état :

I_d : $Act \cup \{NULL\}$
$start_d$: array [Proc, integer, Act] of boolean
r_d : array [Proc] of integer
$Kstart_d$: array [Proc, integer, Act] of boolean
$Knumidle_d$: array [Proc] of integer
$Kguard_d$: array [Proc, integer, Act] of boolean
$postponed_d$: array [Act] of boolean

Conditions initiales :

$I_d = NULL$
for all p,r,I : $\neg start_d[p,r,I]$
for all p : $r_d[p] = 0$
for all p,r,I : $\neg Kstart_d[p,r,I]$
for all p : $Knumidle_d[p] = 0$
for all p,r,I : $\neg Kguard_d[p,r,I]$
for all I : $postponed_d[I] \in \{true, false\}$

Abréviations :

$enabled^I \equiv \forall p : p \in I.idle_p^I$
$executable^I \equiv (enabled^I \wedge \forall J : I \neq J \wedge I \cap J \neq \emptyset.\ \forall p : p \in I \cap J.\ \forall d.\ \neg start_d[p, round_p, J])$
$Knumstart_d^p \equiv max\{r \mid \exists I.Kstart_d[p,r,I]\}$
$Kenabled_d^I \equiv \forall p : p \in I.(Knumidle_d[p] = Knumstart_d^p + 1 \wedge Kguard_d[p, Knumidle_d[p], I])$
$Kexecutable_d^I \equiv (Kenabled_d^I \wedge$
$\forall J : I \neq J \wedge I \cap J \neq \emptyset.\forall p : p \in I \cap J.\neg Kstart_d[p, Knumidle_d[p], J])$

Propriétés de sûreté :

unchanged I_d when $\neg in\,CH_d$ (d-CH1)
$\neg in\,CH_d \Rightarrow I_d \in Iset_d$ (d-CH2)
for all d_i : $voisins(d,d_i) \Rightarrow \neg(in\,EX_d \wedge inEX_{d_i} \wedge I_d \cap I_{d_i} \neq \emptyset)$ (d-CS)
allowed changes to r_d
for all I : $\alpha[d] : in\,T_d \wedge I = I_d \rightarrow after\,T_d \wedge\ \forall p : p \in I.r_d[p]' = Knumidle_d[p]$ (d-T2)
allowed changes to $start_d$
for all I : $\alpha[d] : in\,ST_d \wedge I = I_d \rightarrow after\,ST_d \wedge\ \forall p : p \in I.start_d[p, r_d[p], I]'$ (d-EX3)

Propriétés de vivacité :

for all I : $in\,CH_d \wedge I \in Iset_d \wedge executable^I \rightsquigarrow \neg executable^I \vee (after\,CH_d \wedge I_d = I)$ (d-LCH)
$in\,IP_d \rightsquigarrow at\,EX_d$ (d-LIP)
for all I : $in\,ST_d \wedge I = I_d \rightsquigarrow after\,ST_d \wedge \forall p : p \in I.start_d[p, r_d[p], I]'$ (d-LST)
$in\,OP_d \rightsquigarrow after\,OP_d$ (d-LOP)
for all I : $\Diamond\Box Kexecutable_d^I \Rightarrow \Diamond\Box(\neg(at\,EX_d \wedge I_d = I) \vee \neg postponed_d[I])$ (d-SP1)
for all p,I : $\Diamond\Box idle_p^I \wedge \Box\Diamond enabled^I \wedge \Box\Diamond\neg enabled^I \Rightarrow$
$\forall J : I \cap J = \{\tilde{p}\} \wedge p \neq \tilde{p}.\ \Diamond\Box(\neg(at\,EX_d \wedge I_d = J) \vee \neg Kexecutable_d^J \vee postponed_d[J])$ (d-SP2)

Figure 4.10: Schéma d'un scheduler d garantissant $\mathcal{SPF}$ (rendez-vous binaire)

Propriétés de sûreté :

allowed changes to $Knumidle_d$

for all $p : \alpha[env] : idle_p \wedge Knumidle_d[p] < round_p$

$$\rightarrow Knumidle_d[p]' = Knumidle_d[p] + 1 \quad \text{(e-S2)}$$

for all $p, I : Knumidle_d[p] = round_p \Rightarrow Kguard_d[p, round_p, I] = guard_p[round_p, I]$ (e-S3)

allowed changes to $Kstart_d$

for all $p, r, I : \alpha[env] : \exists d_i.start_{d_i}[p, r, I] \wedge \neg Kstart_d[p, r, I] \rightarrow Kstart_d[p, r, I]$ (e-S4)

for all $I : (in\, EX_d \wedge I_d = I) \Rightarrow$

$$\forall J : I \cap J \neq \emptyset.\ \forall p : p \in I \cap J.\ (\exists d_i.start_{d_i}[p, round_p, J] \Rightarrow Kstart_d[p, round_p, J]) \quad \text{(e-S5)}$$

Propriétés de vivacité :

for all $p, r, I : \exists d.\ start_d[p, r, I] \leadsto Kstart_p[r, I]$ (e-L1)

for all $I : executable^I \wedge I \in Iset_d \leadsto \neg executable^I \vee \forall p : p \in I.(Knumidle_d[p] = round_p)$ (e-L2)

On montre aisément que les propriété de sûreté (SYNC) et (EXCL) sont toujours respectées par le schéma de la Figure 4.10. Le graphe de preuve pour (SPF) est décrit en Figure 4.11. Notons que dans ce graphe de preuve toutes les flèches sont des implications.

1. $\Box\Diamond ready^p \Rightarrow (\Diamond\Box(active_p \vee idle_p) \wedge \Box\Diamond ready^p) \vee \Box\Diamond\exists I : p \in I.\ commit_p^I$
   c'est une instance de la tautologie $P \Rightarrow P \wedge (Q \vee \neg Q)$

2. $\Diamond\Box(active_p \vee idle_p) \wedge \Box\Diamond ready^p \Rightarrow \exists I : p \in I.\ (\Diamond\Box idle_p^I \wedge \Box\Diamond enabled^I)$
   grâce aux règles de changements d'état des processus (p-S1 à p-S4) on montre
   $\forall p.\ \forall I.\ \Diamond\Box(active_p \vee idle_p) \wedge idle_p^I \Rightarrow \Diamond\Box idle_p^I$
   puis on utilise la définition de $ready^p$

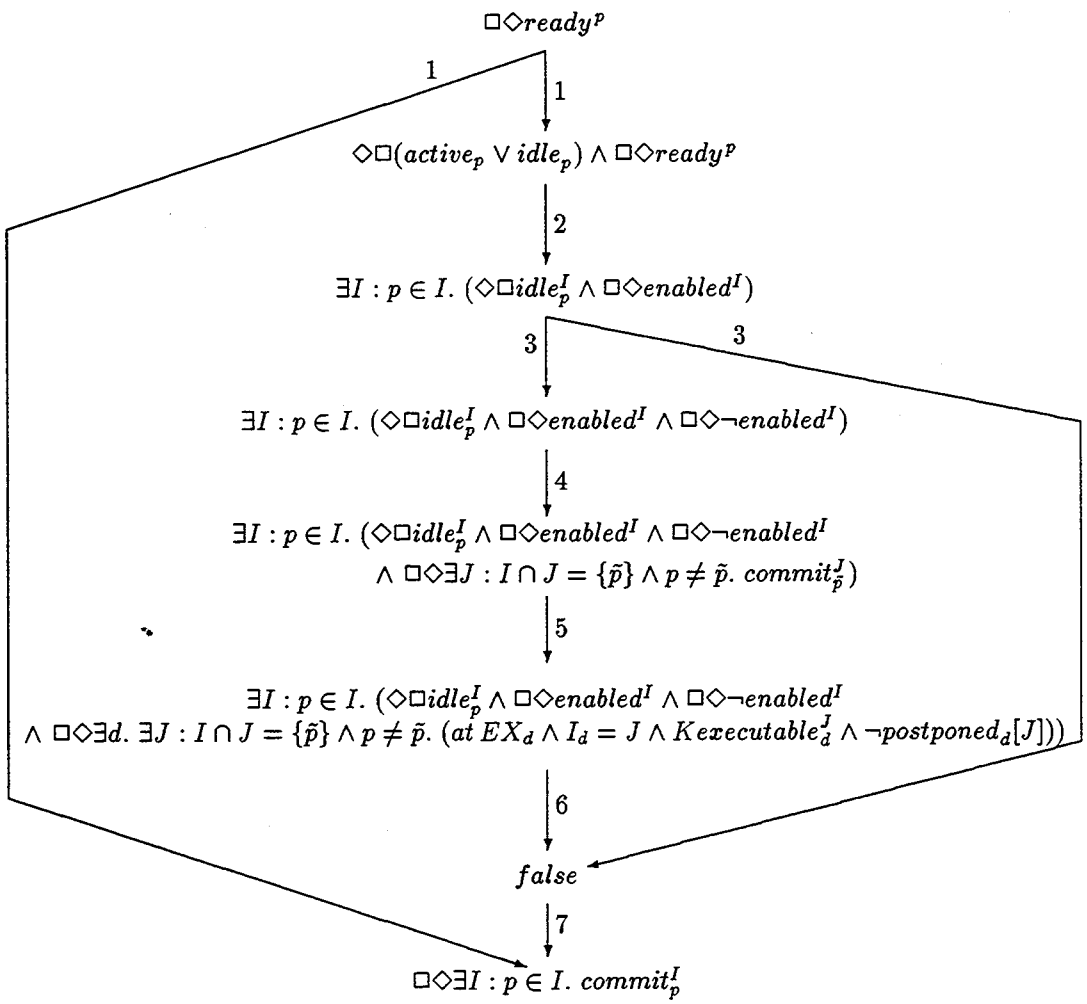
3. $\exists I : p \in I.\ (\Diamond\Box idle_p^I \wedge \Box\Diamond enabled^I) \Rightarrow false \vee \exists I : p \in I.\ (\Diamond\Box idle_p^I \wedge \Box\Diamond enabled^I \wedge \Box\Diamond\neg enabled^I)$
   trivialement
   $\Box\Diamond enabled^I \Rightarrow \Diamond\Box enabled^I \vee (\Box\Diamond enabled^I \wedge \Box\Diamond\neg enabled^I)$
   $\Diamond\Box enabled^I \Rightarrow \forall p : p \in I.\ \Diamond\Box idle_p^I$
   et on montre $\exists I : p \in I.\ \Diamond\Box enabled^I \Rightarrow false$ (c'est-à-dire $\mathcal{WIF}$) grâce à :

   3.1 $\exists I : p \in I.\ \Diamond\Box enabled^I \Rightarrow false \vee \exists I : p \in I.\ \Diamond\Box executable^I$
   c'est une instance de la tautologie $P \Rightarrow P \wedge (Q \vee \neg Q)$
   et par définition
   $\neg executable^I \equiv \neg enabled^I$
   $\vee\neg(\forall J : I \neq J \wedge I \cap J \neq \emptyset.\ \forall p : p \in I \cap J.\ \forall d.\ \neg start_d[p, round_p, J])$
   on montre alors grâce à (e-L1), (p-L1)
   $\Box\Diamond\neg executable^I \leadsto \neg enabled^I$

et donc

$$\exists I : p \in I.\ \Diamond\Box enabled^I \wedge \Box\Diamond\neg executable^I \Rightarrow false$$

Figure 4.11: Graphe de preuve pour $\mathcal{SPF}$ (rendez-vous binaire)



3.2 $\dfrac{\exists I : p \in I.\ \Diamond\Box executable^I \Rightarrow}{\exists I : p \in I.\ (\Diamond\Box executable^I \wedge \Diamond\exists d.(in\,CH_d \wedge I \in Iset_d))}$

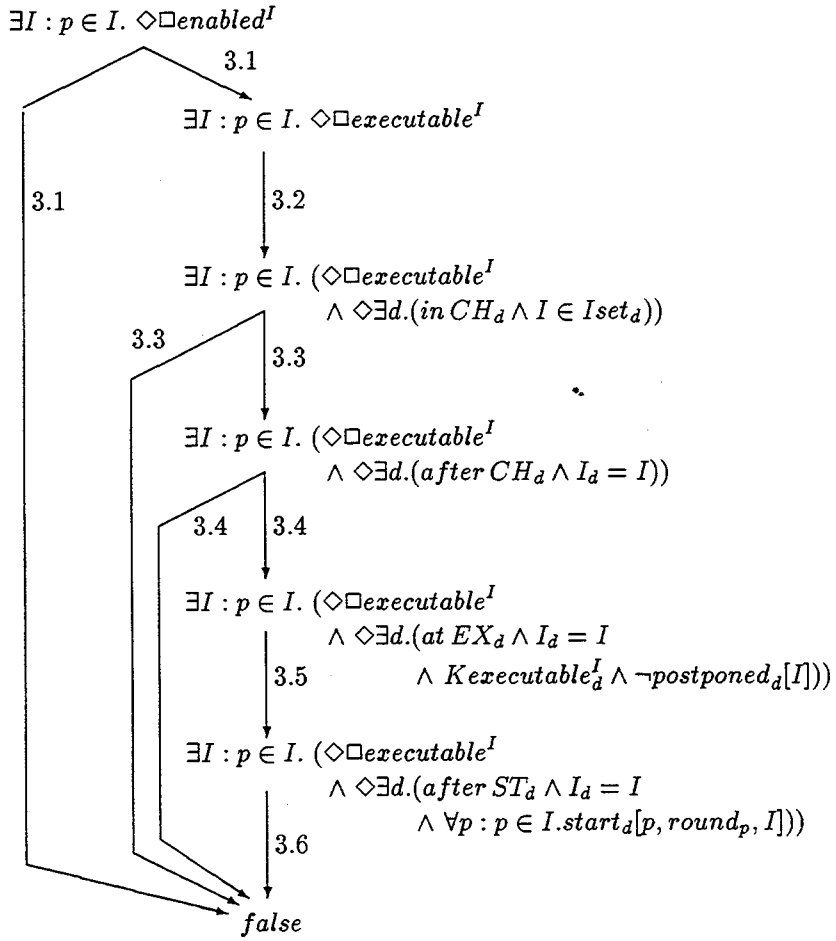
on montre grâce à (d-LIP), (d-LST), (d-LOP) et (**if** control flow axiom)

$$\neg in\,CH_d \rightsquigarrow in\,CH_d$$

et par définition des Isets

$\forall I.\exists d.\ I \in Iset_d$
ces deux résultats permettent de prouver
$true \rightsquigarrow \exists d.(in\,CH_d \wedge I \in Iset_d)$

Figure 4.11: Graphe de preuve pour $\mathcal{SPF}$ (rendez-vous binaire)



3.3 $\dfrac{\exists I : p \in I.\ (\Diamond\Box executable^I \wedge \Diamond\exists d.(in\,CH_d \wedge I \in Iset_d)) \Rightarrow false \vee}{\exists I : p \in I.\ (\Diamond\Box executable^I \wedge \Diamond\exists d.(after\,CH_d \wedge I_d = I))}$
se prouve grâce à (d-LCH)

3.4 $\dfrac{\exists I : p \in I.\ (\Diamond\Box executable^I \wedge \Diamond\exists d.(after\,CH_d \wedge I_d = I)) \Rightarrow false \vee}{\exists I : p \in I.\ (\Diamond\Box executable^I \wedge \Diamond\exists d.(at\,EX_d \wedge I_d = I \wedge Kexecutable_d^I}$

$\wedge\neg postponed_d[I]))$

on montre grâce à (e-S3) et (e-L2)

$\Diamond\Box executable^I \Rightarrow \Diamond\Box K executable_d^I$

puis on utilise (d-LIP) et (d-SP1)

3.5 $\exists I : p \in I.\ (\Diamond\Box executable^I \wedge \Diamond\exists d.(at\,EX_d \wedge I_d = I \wedge K executable_d^I \wedge\neg postponed_d[I])) \Rightarrow \exists I : p \in I.\ (\Diamond\Box executable^I \wedge \Diamond\exists d.(after\,ST_d \wedge I_d = I \wedge \forall p : p \in I.start_d[p, round_p, I])$

se prouve grâce à (d-T2), (d-LST) et (**if** control flow axiom)

3.6 $\exists I : p \in I.\ (\Diamond\Box executable^I \wedge \Diamond\exists d.(after\,ST_d \wedge I_d = I \wedge \forall p : p \in I.start_d[p, round_p, I]) \Rightarrow false$

se prouve grâce à (e-L1) et (p-L1)

4. $\exists I : p \in I.\ (\Diamond\Box idle_p^I \wedge \Box\Diamond enabled^I \wedge \Box\Diamond\neg enabled^I) \Rightarrow \exists I : p \in I.\ (\Diamond\Box idle_p^I \wedge \Box\Diamond enabled^I \wedge \Box\Diamond\neg enabled^I \wedge\Box\Diamond\exists J : I \cap J = \{\tilde{p}\} \wedge p \neq \tilde{p}.\ commit_{\tilde{p}}^J)$

se prouve grâce :

(a) à la définition de $enabled^I$

(b) aux règles de changement d'état des processus (p-S1 à p-S4)

(c) au fait que le rendez-vous est binaire c'est-à-dire que p a un seul partenaire $\tilde{p}$ dans I

5. $\exists I : p \in I.\ (\Diamond\Box idle_p^I \wedge \Box\Diamond enabled^I \wedge \Box\Diamond\neg enabled^I \wedge\Box\Diamond\exists J : I \cap J = \{\tilde{p}\} \wedge p \neq \tilde{p}.\ commit_{\tilde{p}}^J) \Rightarrow \exists I : p \in I.\ (\Diamond\Box idle_p^I \wedge \Box\Diamond enabled^I \wedge \Box\Diamond\neg enabled^I \wedge\Box\Diamond\exists J : I \cap J = \{\tilde{p}\} \wedge p \neq \tilde{p}.\ (at\,EX_d \wedge I_d = J \wedge K executable_d^J \wedge\neg postponed_d[J]))$

se prouve grâce aux règles sur le démarrage des interactions (e-S4), (e-L1), (p-S3) et (p-L1) et à la structure du schéma (**if** control flow axiom)

6. $\exists I : p \in I.\ (\Diamond\Box idle_p^I \wedge \Box\Diamond enabled^I \wedge \Box\Diamond\neg enabled^I \wedge\Box\Diamond\exists J : I \cap J = \{\tilde{p}\} \wedge p \neq \tilde{p}.\ (at\,EX_d \wedge I_d = J \wedge K executable_d^J \wedge\neg postponed_d[J])) \Rightarrow false$

se prouve grâce à (d-SP2)

7. $false \Rightarrow \Box\Diamond\exists I : p \in I.\ commit_p^I$

trivialement

4.3.2 Mise en œuvre des protocoles

Le schéma garantissant $\mathcal{SPF}$ est peu différent de celui proposé pour $\mathcal{WIF}$. De ce fait, dans cette section, nous discutons uniquement du choix du code pour la fonction d'état $postponed_d$. Le choix du code pour les autres parties a déjà été examiné en Section 4.2.2.

- **La fonction d'état** $postponed_d$ **:**

[BT91] est à notre connaissance le seul article publié décrivant un algorithme pour $\mathcal{SPF}$. Cet algorithme utilise des schedulers de processus. Il suppose qu'à chaque interaction est associé un unique jeton. Pour qu'un scheduler puisse démarrer une interaction il faut qu'il possède son jeton. L'utilisation des jetons sert donc à garantir l'exclusion mais aussi à garantir la synchronisation grâce à un protocole d'échange des jetons entre les schedulers associés aux partenaires dans une interaction. Et c'est le mécanisme de gestion locale des jetons qui implémente la fonction *postponed*. En effet chaque scheduler stocke ses jetons dans une queue FIFO et on a $\neg postponed_d[I]$ seulement si le jeton associé à I est le premier dans la queue.

Afin de compléter cet unique exemple de technique employée dans un algorithme, nous proposons maintenant une implémentation originale et nous prouvons que cette implémentation satisfait bien (d-SP1) et (d-SP2).

Exemple : Considérons une implémentation du schéma de la Figure 4.10 qui garantit, en sus des propriétés exigées, les propriétés suivantes :

Constante :
$Maxdiff = \ldots$ tel que $Maxdiff > 0$

Abréviations :
$numstart^p \equiv max\{r \mid \exists d.\exists I.start_d[p,r,I]\}$
$W^I \equiv min\{numstart^p \mid p \in I\}$
$W_d^I \equiv min\{Knumstart_d^p \mid p \in I\}$

Propriété de sûreté :
for all $I : (in\,EX_d \wedge I_d = I) \Rightarrow$
$\quad \forall J : I \cap J = \{q\}.\ \forall p : p \in J.\ \forall K.\ (\exists d_i.start_{d_i}[p,r,K] \Rightarrow Kstart_d[p,r,K])$ (I-S1)

Propriété de vivacité :
for all $p : \Diamond\Box idle_p \Rightarrow \Diamond\Box(Knumidle_d[p] = round_p)$ (I-L1)

(I-L1) assure simplement que si un processus reste continuellement prêt alors le scheduler le saura au bout d'un temps fini. Et (I-S1) garantit que lorsqu'un scheduler teste une interaction I il sait exactement combien de fois chaque processus participant à J a participé à une interaction et cela pour chaque interaction J qui a un seul participant en commun avec I.

$numstart^p$ est le nombre d'interactions auxquelles a participé le processus p depuis le début de l'exécution. Considérons les deux nombres de participations à des interactions pour les participants à l'interaction I : W^I est le plus petit de ces nombres, et W_d^I est la connaissance qu'en a le scheduler d. En fait (I-S1) permet d'assurer que

for all $I : (in\, EX_d \wedge I_d = I) \Rightarrow \forall J : I \cap J = \{p\}.\ W_d^J = W^J$

L'implémentation de $postponed_d$ que nous proposons est la suivante :

$postponed_d[I] \equiv$	SI ($at\, EX_d \wedge I_d = I$) ALORS $\exists J : I \cap J = \{p\}.\ Kexecutable_d^J \wedge W_d^I - W_d^J \geq Maxdiff$ SINON $false$

En d'autres termes, un scheduler reportera le démarrage d'une interaction exécutable I si il existe une interaction exécutable J dont un des participants a participé à une interaction "moins souvent" que les participants à I. Ici "moins souvent" signifie que la différence entre les nombres de participations a dépassée une borne fixée par avance ($Maxdiff$). L'interaction J devra avoir un et un seul participant en commun avec I (reporter le démarrage des interactions n'ayant aucun participant commun ou ayant exactement les mêmes participants ne présente aucun intérêt du point de vue de $\mathcal{SPF}$).

Bien que cette implémentation nécessite que les schedulers aient un peu plus d'informations sur l'état des processus (I-L1) et sur les démarrages des interactions (I-S1), elle se réalise très simplement.

Proposition 4.1 (Correction de l'implémentation)
L'implémentation proposée respecte (d-SP1) et (d-SP2).

Idée de la preuve

• L'implémentation proposée respecte (d-SP1)

Supposons que (d-SP1) ne soit pas respectée c'est-à-dire que l'on ait

$\Diamond\Box Kexecutable_d^I \wedge \Box\Diamond(at\, EX_d \wedge I_d = I \wedge \exists J : I \cap J = \{p\}.\ (Kexecutable_d^J \wedge W_d^I - W_d^J \geq Maxdiff))$

Puisque Act est fini cela implique

$\Diamond\Box Kexecutable_d^I \wedge \exists J : I \cap J = \{p\}.\ \Box\Diamond(at\, EX_d \wedge I_d = I \wedge Kexecutable_d^J \wedge W_d^I - W_d^J \geq Maxdiff)$

I étant continuellement exécutable, W_d^I est constant. Maxdiff étant aussi constant, W_d^J sera constant au bout d'un temps fini. En d'autres termes J sera continuellement exécutable. Les situations de I et J sont exactement symétriques, c'est-à-dire qu'au bout d'un temps fini on devra avoir à la fois $W_d^I - W_d^J \geq Maxdiff$ et $W_d^J - W_d^I \geq Maxdiff$, ce qui est impossible.

• **L'implémentation proposée respecte (d-SP2)**

On montre aisément que

$\Diamond\Box idle_p^I \wedge \Box\Diamond enabled^I \wedge \Box\Diamond\neg enabled^I$

implique

$\Diamond\Box idle_p^I \wedge \Box\Diamond\exists J : I \cap J = \{\tilde{p}\} \wedge p \neq \tilde{p}.\ commit_{\tilde{p}}^J$

p est continuellement prêt et $\tilde{p}$, son partenaire dans I, participe infiniment souvent à des interactions. Cela implique

$\Diamond\Box\forall J : I \cap J = \{\tilde{p}\} \wedge p \neq \tilde{p}.\ (W^J - W^I \geq Maxdiff) \vee \Box\neg commit_{\tilde{p}}^J$

Grâce à (I-S1) on montre

for all I : $(in\,EX_d \wedge I_d = I) \Rightarrow \forall J : I \cap J = \{p\}.\ W_d^J = W^J$

Grâce à (I-L1) on montre

$\Diamond\Box idle_p^I \Rightarrow \forall J : I \cap J = \{\tilde{p}\} \wedge p \neq \tilde{p}.\ \Diamond\Box(Kexecutable_d^J \Rightarrow Kexecutable_d^I)$

En utilisant la tautologie

$\forall J : I \cap J = \{\tilde{p}\} \wedge p \neq \tilde{p}.\ \Diamond\Box(\neg(at\,EX_d \wedge I_d = J \wedge Kexecutable_d^J) \vee (at\,EX_d \wedge I_d = J \wedge Kexecutable_d^J))$

et les résultats prouvés grâce à (I-L1) et (I-S1) on montre que (d-SP2) est respectée. □

4.4 Les critères de qualité de Buckley et Silberschatz

Pour pouvoir estimer la qualité d'un protocole ou comparer les protocoles obtenus nous avons besoin de critères qualitatifs et quantitatifs. G. Buckley et A. Silberschatz [BS83] ont proposé dans leur article quatre critères (appelés depuis *les quatre conditions de Buckley et Silberschatz*) que doit vérifier pour être acceptable tout protocole implémentant le rendez-vous binaire. Ces critères ont été généralisés pour être appliqués aux protocoles pour le rendez-vous multiprocessus. La réalisation de $\mathcal{WIF}$ est un de ces critères. Les trois autres sont la *concurrence*, *l'économie* et *l'efficacité*. Dans cette section nous définissons ces trois critères et nous discutons brièvement leur satisfaction.

Concurrence : *La quantité d'informations externes (globales) nécessaires à chacun des schedulers pour démarrer une interaction doit être faible.*
Pour prendre une décision, un scheduler a seulement besoin de connaître l'état des participants de ses interactions associées (Iset). L'état des autres processus du système n'a pas d'influence directe sur la décision du scheduler. Cela

permet une plus grande concurrence dans l'implémentation car les schedulers qui peuvent démarrer une interaction à un instant donné n'ont pas besoin d'attendre que l'état global du système soit déterminé pour prendre une décision.

Pour savoir si une interaction est exécutable un scheduler a seulement besoin de tester l'état des participants de l'interaction. Il est clair que toute implémentation du test d'exécutabilité qui tient compte de l'état d'autres processus du système réduit la concurrence du protocole obtenu.

Des protocoles solution du problème des philosophes buveurs sont utilisés pour assurer l'exclusion. L'utilisation de protocoles solution du problème des philosophes mangeurs conserve la concurrence lorsque pour toute paire d'interactions n'ayant pas de participants communs il existe une paire de schedulers différents, auxquels elles sont respectivement associées, qui ne sont pas voisins (i.e. rien n'empêche a priori ces schedulers de démarrer les deux interactions en même temps).

Economie : *Seuls un petit nombre de processus et schedulers doivent être impliqués dans le démarrage d'une interaction.*
Le nombre minimum de processus impliqués est bien sûr le nombre de participants de l'interaction.

Les protocoles construits suivant la méthode que nous proposons supposent que les schedulers sont impliqués (en sus des participants) dans le démarrage d'une interaction. Plus précisément, sont impliqués un scheduler qui démarre effectivement l'interaction, mais aussi ses voisins avec lesquels il a dû négocier pour pouvoir entrer en section critique. Et donc moins les protocoles utilisent de schedulers et plus petite est la relation de voisinage, plus ces protocoles sont économiques.

Nous avons supposé que la relation de voisinage doit satisfaire la contrainte :

$$\forall d_j.\forall d_k.(\exists J.\exists K.J \in Iset_{d_j} \wedge K \in Iset_{d_k} \wedge J \neq K \wedge J \cap K \neq \emptyset) \Rightarrow voisins(d_j, d_k))$$

En fait cette contrainte peut être affaiblie en

$$\forall d_j.\forall d_k.(\exists J.\exists K.J \in Iset_{d_j} \wedge K \in Iset_{d_k} \wedge J \neq K \wedge conflict(J,K)) \Rightarrow voisins(d_j, d_k))$$

où $conflict(J,K)$ signifie que les interactions J et K ont un participant commun ($conflict(J,K) \Rightarrow J \cap K \neq \emptyset$) et qu'il existe un état atteignable du programme considéré dans lequel elles sont toutes deux exécutables [FF90].
Différents algorithmes de calcul statique des points de synchronisation d'un programme ont été proposés (e.g. [Mer90]), en particulier pour déterminer les

états bloquants d'un programme. Ces algorithmes peuvent être utilisés pour calculer la *relation de conflit* ou une approximation de cette relation.

Efficacité : *Le protocole pour le rendez-vous doit être efficace par rapport au nombre de messages échangés*[2].
Pour estimer l'efficacité d'un protocole on calcule les deux mesures suivantes ([Bag89a]) :

1. Le nombre total de messages échangés par rapport au nombre d'interactions exécutées.
2. Le nombre de messages échangés nécessaires pour démarrer une interaction (temps de réponse) c'est-à-dire mesuré à partir de l'instant où une interaction devient exécutable jusqu'à l'instant où elle est exécutée. Ce nombre de messages est en fait déterminé à partir du nombre de messages nécessaires pour déterminer si une interaction est exécutable (temps de synchronisation), et le nombre de messages nécessaires pour choisir une interaction parmi celles qui sont exécutables (temps de sélection).

Ces deux mesures sont, bien sûr, toujours estimées pour les cas les plus défavorables.

La majorité des auteurs de protocole ont estimé l'efficacité de leurs protocoles et l'ont comparé à celles d'autres protocoles publiés.

Il est très souvent possible de transformer a posteriori les protocoles construits afin d'améliorer leur efficacité (optimisation globale). La comparaison de la structure des schémas que nous proposons avec la structure des protocoles publiés permet quelquefois de retrouver ces transformations. C'est le cas par exemple du premier protocole proposé dans [Bag89a]. Ses protocoles d'entrée et de sortie (IP/OP) sont basés sur la circulation d'un privilège sur un anneau. Les schedulers ne peuvent ni réclamer ni refuser le privilège : ils peuvent donc choisir une interaction seulement quand ils sont en section critique. Afin d'améliorer l'efficacité du protocole les schedulers considèrent, quand ils sont en section critique, successivement toutes les interactions de leur Iset et non pas une seule de ces interactions.

4.5 Conclusions

Diverses extensions à ce travail peuvent être envisagées en fonction des particularités des langages considérés. En particulier nous avons considéré lors de la spécification des protocoles que l'ensemble des processus et l'ensemble des interactions sont fixés

[2] L'expression "échange de message" signifie ici aussi bien envoi/réception d'un message que diffusion d'un message ou écriture/lecture d'une variable propre ...

c'est-à-dire qu'on connaît a priori tous leurs éléments. Pour pouvoir implémenter des langages qui permettent la création dynamique de processus (par exemple LOTOS) et/ou leur terminaison, il faut considérer le cas où ces ensembles peuvent varier au cours de l'exécution du programme. L'étude de [vBGW89] et [Sjö90] traitant ce cas nous laisse penser qu'il suffit de créer et/ou détruire dynamiquement les schedulers en même temps que les interactions, et de ne pas changer les schémas proposés mais d'utiliser dans les protocoles d'entrée et de sortie des techniques bien adaptées à la création et l'insertion dynamique de processus dans un réseau (e.g. [KM90]). Par exemple, l'algorithme de [vBGW89] utilise avantageusement le fait qu'il est facile d'insérer un processus dans un réseau en anneau et qu'il est facile dans un tel réseau de diffuser (ou collecter) une information à ses voisins sans en connaître le nombre exact.

L'approche que nous avons adopté permet aisément de considérer d'autres propriétés d'équité : il suffit de construire un nouveau schéma en transformant le schéma satisfaisant la synchronisation et l'exclusion (Section 4.2.1).

Proposer un schéma pour la U-équité [AFL90] nécessite un peu plus de travail. En effet la définition de la U-équité suppose un modèle où toutes les actions des processus sont gardées, i.e. les processus n'ont pas d'état *active*. Mais l'étude du protocole proposé dans [AFL90] nous laisse penser qu'un schéma peut être construit en reprenant les idées directrices qui nous ont servi ici (e.g. utilisation des algorithmes solutions du problème des philosophes buveurs).

Nous avons énoncé des critères de qualité que doit satisfaire un protocole pour être acceptable (Section 4.4). Mais nous n'avons pu donner aucune indication sur les relations existant entre ces différents critères et les propriétés d'équité. En particulier il serait intéressant de savoir, étant donnée une propriété d'équité et étant donnée une configuration du système (schedulers d'interaction, schedulers de processus ...), quelles sont la concurrence et l'efficacité maximales (lower bounds) qui peuvent être obtenues. A. Sistla [Sis84] donne une borne inférieure (lower bound) pour l'efficacité (en temps) dans le cas du rendez-vous binaire avec $\mathcal{WIF}$ et des protocoles à base de schedulers de processus. Mais il suppose aussi que tout processus devient prêt au bout d'un temps fini. L'établissement de ces bornes pour la concurrence et le nombre de messages échangés est un problème ouvert.

Bibliographie

Les références suivies de * décrivent un protocole pour le rendez-vous binaire; celles suivies de ** décrivent un protocole pour le rendez-vous multiprocessus; celles suivies de † traitent d'une notion d'équité et du rendez-vous; et enfin celles suivies de ‡ décrivent un (langage comportant un) mécanisme de communication basé sur le rendez-vous.

[AB83]‡ S. Abramsky et R. Bornat. Pascal-m : A language for loosely coupled distributed systems. In Y. Paker et J.-P. Verjus, éditeurs, *Distributed Computing Systems*, pages 163–189. Academic Press, 1983.

[AFG90]† P.C. Attie, N. Francez, et O. Grümberg. Fairness and hyperfairness in multiparty interactions. In *Proc. 17th ACM Symp. Principles of Programming Languages, San Francisco, California*, Janvier 1990.

[AFK87]† K.R. Apt, N. Francez, et S. Katz. Appraising fairness in languages for distributed programming. In *Proc. 14th ACM Symp. Principles of Programming Languages, Munich*, pages 189–198, Janvier 1987. Version longue dans Rapport CWI, CS-R8811, Mars 1988.

[AFL90]**† P. Attie, I.R. Forman, et E. Levy. On fairness as an abstraction for the design of distributed systems. In *Proc. of 10th Int. Conf. on Distributed Computing Systems, Paris, France*, pages 150–157, Mai 1990.

[AS83] G.R. Andrews et F.B. Schneider. Concepts and notations for concurrent programming. *ACM Computing Surveys*, 15(1):3–43, Mars 1983.

[Bag89a]** R. Bagrodia. Process synchronization: Design and performance evaluation of distributed algorithms. *IEEE Transactions on Software Engineering*, 15(9):1053–1065, Septembre 1989.

[Bag89b]* R. Bagrodia. Synchronization of asynchronous processes in CSP. *ACM Transactions on Programming Languages and Systems*, 11(4):585–597, Octobre 1989.

[BB87]‡ T. Bolognesi et E. Brinksma. Introduction to the ISO specification language LOTOS. *Computer Networks and ISDN Systems*, 14:25–59, 1987.

[BEKS84]* R.J.R. Back, P. Eklund, et R. Kurki-Suonio. A fair and efficient implementation of CSP with output guards. Rapport Ser. A, 38, Dept. of Computer Science, Abo Akademi, Finland, 1984.

[Ber80]* A.J. Bernstein. Output guards and nondeterminism in 'Communicating Sequential Processes'. *ACM Transactions on Programming Languages and Systems*, 2(2):234–238, Avril 1980.

[BH87]‡ P. Brinch Hansen. Joyce - A programming language for distributed systems. *Software - Practice and Experience*, 17(1):29–50, Janvier 1987.

[BHKS85]** R.J.R. Back, E. Hartikainen, et R. Kurki-Suonio. Multiprocess handshaking on broadcasting networks. Rapport Ser. A, 42, Dept. of Computer Science, Abo Akademi, Finland, 1985.

[BK88] O. Baruch et S. Katz. Partially interpreted schemas for CSP programming. *Science of Computer Programming*, 10:1–18, 1988.

[BKS84]** R.J.R. Back et R. Kurki-Suonio. Cooperation in distributed system using symmetric multiprocess handshaking. Rapport Ser. A, 34, Dept. of Computer Science, Abo Akademi, Finland, 1984.

[BKS88a]**† R.J.R. Back et R. Kurki-Suonio. Distributed cooperation with Action Systems. *ACM Transactions on Programming Languages and Systems*, 10(4):513–554, 1988.

[BKS88b]† R.J.R. Back et R. Kurki-Suonio. Serializability in distributed systems with handshaking. In *Proc. 15th ICALP, Tampere, LNCS 317*, pages 52–66. Springer-Verlag, Juillet 1988.

[BKZ79]* J.S. Banino, C. Kaiser, et H. Zimmermann. Synchronization for distributed systems using a single broadcast channel. In *Proc. of 1st Int. Conf. on Distributed Computing Systems*, pages 330–338, Octobre 1979.

[Bor83]‡ R. Bornat. Programming styles in Pascal-m. CSL report, Queen Mary College, Londres, Août 1983.

[Bor86]* R. Bornat. A protocol for generalized Occam. *Software - Practice and Experience*, 16(9):783–799, Septembre 1986.

[Bou88] L. Bougé. On the existence of symmetric algorithms to find leaders in networks of communicating sequential processes. *Acta Informatica*, 25:179–201, 1988.

[BR89]‡ G. Bruns et C. Richter. Rada - an ADA-based language for distributed systems design. Rapport STP-251-89, Microelectronics and Computer Technology Corp., Austin, Texas, Octobre 1989.

[BS83]* G.N. Buckley et A. Silberschatz. An effective implementation for the generalized input-output construct of CSP. *ACM Transactions on Programming Languages and Systems*, 5(2):223–235, Avril 1983.

[BT91]** R. Bagrodia et Y.-K. Tsay. An efficient algorithm for fair interprocess synchronization. Soumis pour publication, 1991.

[Car84]* L. Cardelli. An implementation model of rendezvous communication. In *Seminar on Concurrency, Pittsburgh, LNCS 197*, pages 449–457. Springer-Verlag, Juillet 1984.

[Cha87]‡ A. Charlesworth. The multiway rendezvous. *ACM Transactions on Programming Languages and Systems*, 9(3):350–366, Juillet 1987.

[CM84] K.M. Chandy et J. Misra. The drinking philosophers problem. *ACM Transactions on Programming Languages and Systems*, 6(4):632–646, Octobre 1984.

[CM86] K.M. Chandy et J. Misra. How processes learn. *Distributed Computing*, 1:40–52, 1986.

[CM88]** K.M. Chandy et J. Misra. *Parallel Program Design : A Foundation.* Addison-Wesley, 1988.

[CPR91] A. Couvert, R. Pedrono, et M. Raynal. Implementation and evaluation of distributed synchronization on a distributed memory parallel machine. In *Proc. 2nd European Conf. EDMCC2, Munich, LNCS 487*, pages 304–314. Springer-Verlag, Avril 1991.

[dBP91] F.S. de Boer et C. Palamidessi. Embedding as a tool for language comparison: on the CSP hierarchy. In J.C.M. Baeten et J.F. Groote, éditeurs, *Proc. CONCUR 91, Amsterdam, LNCS 527*, pages 127–141. Springer-Verlag, Août 1991.

[Dij65] E.W. Dijkstra. Solution of a problem in concurrent programming control. *Communications of the ACM*, 8(9), Septembre 1965.

[Dij71] E.W. Dijkstra. Hierarchical ordering of sequential processes. *Acta Informatica*, 1(2):115–138, 1971.

[Dij74] E.W. Dijkstra. Self-stabilizing systems in spite of distributed control. *Communications of the ACM*, 17(11):643–644, Novembre 1974.

[Dij75] E.W. Dijkstra. Guarded commands, nondeterminacy, and formal derivation of programs. *Communications of the ACM*, 18(8):453–457, Août 1975.

[EFK89]‡ M. Evangelist, N. Francez, et S. Katz. Multiparty interactions for interprocess communication and synchronization. *IEEE Transactions on Software Engineering*, 15(11):1417–1426, Novembre 1989.

[Ekl84]** P. Eklund. Synchronizing multiple processes in common handshakes. Rapport Ser A., 39, Abo Akademi, Dept. of Computer Science, Finland, Novembre 1984.

[FF90] N. Francez et I.R. Forman. Conflict propagation. In *Proc. of 1990 Int. Conf. on Computer Languages, New-Orleans, Louisiana*, pages 155–168, Mars 1990.

[FH83]‡ N. Francez et B. Hailpern. Script: a communication abstraction mechanism. In *Proc. of 2nd ACM Symp. Principles of Distributed Computing, Montreal , Canada*, pages 213–227, Août 1983.

[For86]‡ I.R. Forman. On the design of large distributed systems. In *Proc. of IEEE 1986 Int. Conf. on Computer Languages, Miami, Florida*, pages 84–95, Octobre 1986.

[FR80]* N. Francez et M. Rodeh. A distributed abstract data type implemented by a probabilistic communication scheme. In *Proc. of 21st Symp. on Foundations of Computer Science*, pages 373–379, Octobre 1980.

[Fra86] N. Francez. *Fairness.* Springer-Verlag, 1986.

[Fra89] N. Francez. Cooperating proofs for distributed programs with multiparty interactions. *Information Processing Letters*, 32(5):235–242, Septembre 1989.

[GFK84] O. Grümberg, N. Francez, et S. Katz. Fair termination of communicating processes. In *Proc. of 3rd ACM Symp. Principles of Distributed Computing, Vancouver, B.C., Canada*, Août 1984.

[GFMdR85] O. Grümberg, N. Francez, J. Makowsky, et W.P. de Roever. A proof rule for fair termination of guarded commands. *Information and Control*, 66(1):83–102, Juillet 1985.

[GY91]* R. Govindarajan et S. Yu. Data flow implementation of generalized guarded commands. In *Proc. PARLE 91, vol. I: Parallel Architectures and Algorithms, Eindhoven, LNCS 505*, pages 372–389. Springer-Verlag, Juin 1991.

[Hoa78]‡ C.A.R. Hoare. Communicating Sequential Processes. *Communications of the ACM*, 21(8):666–677, Août 1978.

[JS91]‡ Y.-J. Joung et S.A. Smolka. Coordinating first-order multiparty interactions. In *Proc. 18th ACM Symp. Principles of Programming Languages, Orlanda, Florida*, pages 209–220, Janvier 1991.

[KM90] J. Kramer et J. Magee. The evolving philosophers problem : dynamic change management. *IEEE Transactions on Software Engineering*, 16(11):1293–1306, Novembre 1990.

[KPB+90] F. Kurfess, X. Pandolfi, Z. Belmesk, W. Ertel, R. Letz, et J. Schumann. PARTHEO and FP2 : Design of a parallel inference machine. In P.C. Treleaven, éditeur, *Parallel Computers : Object-Oriented, Functional, Logic*, chapitre 9, pages 259–297. Wiley, 1990.

[KS79]* R.B. Kieburtz et A. Silberschatz. Comments on 'Communicating Sequential Processes'. *ACM Transactions on Programming Languages and Systems*, 1(2):218–225, Octobre 1979.

[Kum90]** D. Kumar. An implementation of n-party synchronization using tokens. In *Proc. 10th Int. Conf. on Distributed Computing Systems, Paris, France*, pages 320–327, Mai 1990.

[Lam78] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, Juillet 1978.

[Lam80a] L. Lamport. The "Hoare logic" of concurrent programs. *Acta Informatica*, 14(1):21–37, 1980.

[Lam80b] L. Lamport. "Sometimes" is sometimes "Not Never". In *Proc. 7th ACM Symp. Principles of Programming Languages, Las Vegas*, pages 174–185, Janvier 1980.

[Lam83] L. Lamport. Specifying concurrent program modules. *ACM Transactions on Programming Languages and Systems*, 5(2):190–222, Avril 1983.

[Lam89] L. Lamport. A simple approach to specifying concurrent systems. *Communications of the ACM*, 32(1):32–45, Janvier 1989.

[Lev88]** E. Levy. A survey of distributed coordination algorithms. Rapport Technique STP-271-88, Microelectronics and Computer Technology Corp., Austin, Texas, Septembre 1988.

[LL90]‡ P. Labrèche et L. Lamarche. Interactors: A real-time executive with multiparty interactions in C++. *SIGPLAN Notices*, 25(4):20–32, 1990.

[LS85] L. Lamport et F.B. Schneider. Formal foundations for specification and verification. In *Distributed Systems. Methods and Tools for Specification, LNCS 190*, pages 203–285. Springer-Verlag, 1985.

[May83]‡ D. May. OCCAM. *SIGPLAN Notices*, 13(4), Avril 1983.

[Mer90] N. Mercouroff. *Analyse sémantique des communications entre processus de programmes parallèles.* Thèse de doctorat, Ecole Polytechnique, Septembre 1990.

[Mil80]‡ R. Milner. *A Calculus of Communicating Systems, LNCS 92.* Springer-Verlag, 1980.

[Mit86]* K. Mitchell. *Implementations of process synchronisation and their analysis.* PhD thesis, Univ. of Edinburgh, Dept. of Computer Science, Juillet 1986.

[Mor90]* C. Morin. An efficient implementation of the rendezvous atomicity property. In D.J. Evans et al., éditeurs, *Parallel Computing 89*, pages 603–608. North-Holland, 1990.

[MR87]‡ S.L. Mehndiratta et S. Ramesh. A methodology for developing distributed programs. *IEEE Transactions on Software Engineering*, 13(8):967–976, Août 1987.

[Nat86]* N. Natarajan. A distributed synchronisation scheme for communicating processes. *The Computer Journal*, 29(2):109–117, 1986.

[OA88] E.-R. Olderog et K.R. Apt. Fairness in parallel programs: the transformational approach. *ACM Transactions on Programming Languages and Systems*, 10(3):420–455, Juillet 1988.

[OL82] S.S. Owicki et L. Lamport. Proving liveness properties of concurrent programs. *ACM Transactions on Programming Languages and Systems*, 4(3):455–495, Juillet 1982.

[PK90]**† M.H. Park et M. Kim. A distributed synchronization scheme for fair multi-process handshakes. *Information Processing Letters*, 34(3):131–138, Avril 1990.

[Ram87a]** S. Ramesh. A new and efficient implementation of multiprocess synchronization. In *Proc. PARLE 87, vol. II: Parallel Languages, Eindhoven, LNCS 259*, pages 387–401. Springer-Verlag, Juin 1987.

[Ram87b]* S. Ramesh. A new implementation of CSP with output guards. In *Proc. 7th Int. Conf. on Distributed Computing Systems*, pages 266–273, 1987.

[Ray84] M. Raynal. *Algorithmique du parallélisme : le problème de l'Exclusion Mutuelle.* Dunod, 1984.

[RD84]‡ G.-C. Roman et M.S. Day. Multifaceted distributed systems specification using processes and event synchronization. In *Proc. 7th Int. Conf. on Software Engineering, Orlando, Florida*, pages 44–55, Mars 1984.

[RS84]* J.H. Reif et P.G. Spirakis. Real-time synchronization of interprocess communication. *ACM Transactions on Programming Languages and Systems*, 6(2):215–238, Avril 1984.

[Sch78]* J.S. Schwartz. Distributed synchronization of communicating sequential processes. Rapport DAI TR 56, Dept. of A.I., University of Edinburgh, 1978.

[Sch82]* F.B. Schneider. Synchronization in distributed programs. *ACM Transactions on Programming Languages and Systems*, 4(2):179–195, Avril 1982.

[Sil79]* A. Silberschatz. Communication and synchronization in distributed systems. *IEEE Transactions on Software Engineering*, 5(6):542–546, Novembre 1979.

[Sis84]*† A.P. Sistla. Distributed algorithms for ensuring fair interprocess communications. In *Proc. of 3rd ACM Symp. Principles of Distributed Computing, Vancouver, B.C., Canada*, pages 266–277, Août 1984.

[SJ89]‡ Ph. Schnoebelen et Ph. Jorrand. Principles of FP2. Term algebras for specification of parallel machines. In J. W. de Bakker, éditeur, *Languages for Parallel Architectures: Design, Semantics, Implementation Models*, chapitre 5, pages 223–273. Wiley, 1989.

[Sjö90]** P. Sjödin. Implementing LOTOS as asynchronously communicating processes. Rapport SICS/R-90/90014, Swedish Institute of Computer Science, Novembre 1990.

[TB91]** Y.-K. Tsay et R. Bagrodia. Some impossibility results in interprocess synchronization. Soumis pour publication, 1991.

[Tho88] R. Thoraval. *Les principes et mécanismes fondamentaux de l'exclusion mutuelle équitable dans les systèmes distribués.* Thèse de doctorat, Université de Rennes I, Mars 1988.

[vBGW89]** G. von Bochmann, Q. Gao, et C. Wu. On the distributed implementation of LOTOS. In *Proc 2nd Int. Conf. on Formal Description Techniques for Distributed Systems and Communications Protocols : FORTE'89, Vancouver B.C., Canada*, Décembre 1989.

[VdS81]* J.L.A. Van de Snepscheut. Synchronous communication between asynchronous components. *Information Processing Letters*, 13(3):127–130, Décembre 1981.

[Yan91]‡ J.T. Yantchev. Communication abstraction and refinement. In *Proc. PARLE 91, vol. II: Parallel Languages, Eindhoven, LNCS 506*, pages 148–165, Juin 1991.

Résumé

Nous étudions la mise en œuvre du rendez-vous multiprocessus et de l'équité dans les langages du type "CSP généralisé".

Nous proposons une méthode de construction des protocoles. Cette méthode consiste à mettre en œuvre ces protocoles à partir d'un schéma de protocole, c'est-à-dire un protocole comportant des parties abstraites. La mise en œuvre effective d'un protocole se fait en remplaçant les parties abstraites par du code. Nous discutons le choix de ce code à l'aide d'exemples tirés de la littérature.

Nous étudions six notions d'équité dites classiques. Nous montrons que parmi ces six notions d'équité seules les notions d'équité dites fortes accroissent la vivacité. Nous montrons aussi qu'en général il est impossible de construire un protocole réalisant une de ces notions d'équité fortes : seule la "Strong Process Fairness" peut l'être, et cela seulement lorsque le rendez-vous est binaire. Nous étudions la construction des protocoles réalisant la "Weak Interaction Fairness" et la "Strong Process Fairness" (avec rendez-vous binaire). Nous dérivons, à partir d'une spécification, un schéma pour chacune de ces notions d'équité.

Abstract

We study the implementation of multiparty rendezvous and fairness for "generalized CSP-like" languages.

A method for constructing protocols is proposed. It is based on the use of protocol schemas, i.e. protocols containing abstract sections. A schema may be instantiated for obtaining a correct protocol by appropriately substituting abstract sections by effective code. Examples are given of such code for classical protocols.

We study six commonly used notions of fairness. We show that from these six notions, only strong notions of fairness are liveness enhancing. We also show that, among these strong notions of fairness, only "Strong Process Fairness" with binary rendezvous can have a distributed implementation. We study the design of protocols for "Weak Interaction Fairness" and "Strong Process Fairness" (binary rendezvous). We derive from a specification a schema for each of these notions of fairness.