Méthodes de vérification de spécifications comportementales : étude et mise en œuvre

Laurent Mounier

THESE

présentée par

Laurent MOUNIER

pour obtenir le titre de **DOCTEUR** de

L'UNIVERSITE JOSEPH FOURIER - GRENOBLE I

(arrêté ministériel du 5 juillet 1984)

Spécialité : INFORMATIQUE

Méthodes de Vérification de Spécifications Comportementales : étude et mise en œuvre

---

Date de soutenance : 31 janvier 1992

Composition du jury :

Président :	J. VOIRON
Rapporteurs :	R. DE SIMONE
	P. WOLPER
Examinateurs :	J.C. FERNANDEZ
	C. JARD
	J. SIFAKIS

Thèse préparée au sein du Laboratoire de Génie Informatique

Remerciements

Je tiens à remercier :

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

Monsieur Jean-Claude Fernandez, *Maître de Conférences à l'Université Joseph Fourier, qui est le Directeur de cette thèse, et sans lequel ce travail n'aurait pu aboutir. Je le remercie en particulier pour le soutien constant qu'il m'a offert pendant toute la durée de cette thèse, et dont j'ai pu mesurer la valeur en de multiples occasions*

Monsieur Claude Jard, *Chargé de Recherche au C.N.R.S., qui a accepté de participer au jury. Je tiens à le remercier pour l'intérêt qu'il porte à mon travail et pour les nombreux éclaircissements qu'il m'a apporté sur la vérification "à la volée"*

Monsieur Joseph Sifakis, *Directeur de Recherche au C.N.R.S., qui m'a accueilli dans son équipe. Je lui suis particulièrement reconnaissant pour les motivations et les impulsions qu'il a toujours su donner à mon travail*

Monsieur Robert de Simone, *Directeur de Recherche à l'I.N.R.I.A., qui a bien voulu juger ce travail. Je souhaite le remercier pour ses nombreuses remarques et suggestions qui ont contribuées à enrichir ce document*

Monsieur Pierre Wolper, *Professeur à l'Université de Liège, qui a également accepté la charge de Rapporteur. Je suis très sensible aux encouragements qu'il m'a ainsi apporté*

Je tiens également à remercier l'ensemble des membres du projet SPECTRE, qui m'ont offert un cadre de travail privilégié et dont je garderais un excellent souvenir.

Je remercie en particulier :

Hubert Garavel, *dont les idées et la compétence ont été largement sollicitées lors de la réalisation de ce travail,*

Claire Loiseaux *et* Anne Rasse *qui ont accepté de relire et commenter les premières versions de ce document,*

Ahmed Bouajjani, Suzanne Graf, Alain Kerbrat, Florence Maraninchi, Xavier Nicollin, Christophe Ratel, Pascal Raymond, Fréderic Rocheteau *et* Carlos Rodriguez, *pour les conseils et l'aide efficace qu'ils m'ont si souvent apporté.*

Table des matières

Introduction

L'importance croissante du *parallélisme* et de la *répartition* dans les systèmes informatiques actuels a rendu nécessaire le développement de méthodes et de logiciels d'aide à la *conception* et à la *vérification* adaptés à ce type d'applications.

Parmi les différentes méthodes de vérification existantes, la *vérification formelle* a pour objectif de garantir le fonctionnement correct d'un programme vis-à-vis de l'ensemble de ses exécutions possibles. Plus précisément, elle consiste à *comparer* une description de ce programme, exprimée dans un formalisme approprié, avec une description de ses *spécifications*, c'est à dire de l'ensemble des propriétés qu'il doit satisfaire. Schématiquement, la vérification formelle nécessite donc la réunion de quatre éléments ([God90]) :

- une représentation $\mathcal{P}$ du programme à vérifier,
- une représentation $\mathcal{S}$ de ses spécifications,
- une *relation de satisfaction*, qui définit formellement la comparaison de $\mathcal{P}$ et $\mathcal{S}$,
- et, le cas échéant, un algorithme, ou *procédure de décision*, pour mettre en œuvre cette comparaison.

Notons que les notions de *programme* et de *spécifications* sont en fait assez relatives, et que, d'une façon générale, $\mathcal{P}$ et $\mathcal{S}$ peuvent être vues comme deux représentations formelles du même système, $\mathcal{P}$ désignant la plus détaillée et $\mathcal{S}$ la plus abstraite.

Concernant les systèmes parallèles, l'intérêt de cette approche vient du fait que, pour une large classe d'entre-eux au moins, les aspects essentiels de leur fonctionnement peuvent être représentés par un nombre *fini* d'états. Par conséquent, la comparaison entre le *modèle* $\mathcal{P}$ ainsi obtenu et les spécifications $\mathcal{S}$ peut être mise en œuvre en utilisant des procédures de décision efficaces qui reposent sur une *énumération exhaustive* de l'ensemble des états de $\mathcal{P}$.

Cette idée est à l'origine d'une méthode de vérification automatique, généralement désignée sous le terme de *méthode basée sur les modèles* (ou *model-based method*), que nous présentons plus en détail dans la suite.

La vérification à partir d'un modèle

En premier lieu, les programmes que l'on considère dans cette approche doivent être exprimés dans un langage qui possède une *sémantique opérationnelle* bien définie. Dans le cas des programmes parallèles, des exemples de langages fréquemment utilisés sont CSP [Hoa78], certaines *techniques de description formelles* normalisées (ou FDT, comme ESTELLE [ISO88] ou LOTOS [ISO87]), et, plus généralement, tous les langages basés sur les *algèbres de processus* (tels CCS [Mil80], MEIJE [Bou85b]).

A partir de la sémantique opérationnelle du langage, il est possible d'associer à tout programme donné un *système de transitions étiquetées* $\mathcal{P}$ (i.e., un ensemble d'états muni d'une relation de transition) qui modélise son *comportement* (i.e., l'ensemble de toutes ses exécutions possibles).

Selon le formalisme utilisé pour représenter les spécifications, on distingue alors deux méthodes de vérification :

spécifications logiques :
Elles permettent de caractériser des *propriétés globales* du programme telles que l'absence de blocage, l'exclusion mutuelle ou encore l'équité. Ce type de spécification est généralement exprimé par des formules d'une *logique temporelle* qui sont interprétées sur l'ensemble des systèmes de transitions étiquetées.

Une spécification $\mathcal{S}$ est donc représentée par une classe infinie de systèmes de transition étiquetées et la vérification consiste à s'assurer que le système particulier $\mathcal{P}$ appartient à cette classe (ou encore que $\mathcal{P}$ est un *modèle* pour $\mathcal{S}$, ce que l'on note $\mathcal{P} \models \mathcal{S}$). Toute procédure de décision pour la relation de satisfaction $\models$ définit alors une méthode pratique de vérification.

spécifications comportementales :
Elles consistent à décrire le *comportement attendu* du programme, observé à un certain niveau d'*abstraction*. Les spécifications sont donc représentées elles aussi par un systèmes de transitions étiquetées $\mathcal{S}$ qui peut éventuellement être issu d'un programme exprimé dans le même langage que le programme à vérifier.

La comparaison entre les deux système de transitions étiquetées $\mathcal{P}$ et $\mathcal{S}$ s'effectue alors au moyen d'une *relation d'équivalence ou de préordre* $\mathcal{R}$, dont le choix est fonction des *critères d'abstraction* que l'on souhaite prendre en compte. Toute procédure de décision pour la relation $\mathcal{R}$ définit donc une méthode pratique de vérification.

Ces deux approches peuvent êtres vues comme complémentaires : il apparaît en effet, que, parmi l'ensemble des propriétés que l'on est amené à vérifier en pratique, certaines d'entre elles peuvent facilement être représentées comme une abstraction du comportement attendu du programme, alors que d'autres s'expriment mieux dans un formalisme plus déclaratif, comme les logiques temporelles.

Enfin, notons également que les relations d'équivalence définies pour la vérification de spécifications comportementales peuvent aussi jouer un rôle dans le cadre de la vérification de spécifications logiques. En effet, certaines logiques temporelles sont *adéquates* pour une relation d'équivalence $\mathcal{R}$ dans le sens où deux systèmes de transitions étiquetées satisfont les mêmes formules de cette logique si et seulement si ils sont équivalents par $\mathcal{R}$. Par suite, pour vérifier que $\mathcal{P}$ satisfait une formule $\mathcal{S}$, il suffit de s'assurer que n'importe quel système de transitions étiquetées $\mathcal{P}'$ équivalent à $\mathcal{P}$ modulo $\mathcal{R}$ satisfait $\mathcal{S}$. L'intérêt d'une telle démarche est alors directement lié à l'obtention d'un système $\mathcal{P}'$ dont le nombre d'états est inférieur à celui de $\mathcal{P}$.

Dans ce travail, nous nous sommes plus particulièrement intéressés à la vérification de spécifications comportementales, et, plus précisément, aux méthodes pratiques de vérification qui sont associées à cette approche.

La vérification de spécifications comportementales

Parmi l'ensemble des relations d'équivalence entre systèmes de transitions étiquetées dédiées à la vérification de spécifications comportementales, la *bisimulation forte* [Mil80, Par81] occupe une place prépondérante. En effet, cette relation est tout d'abord importante du point de vue théorique dans le sens où elle sert de support à la définition de toute une *famille* de relations — les *relations de bisimulation* — qui correspondent chacune à des critères d'abstraction différents. Mais, il existe

également un second intérêt, plus pratique, dû au fait qu'il lui est associée une procédure de décision efficace, qui peut alors être étendue à chaque élément de cette famille.

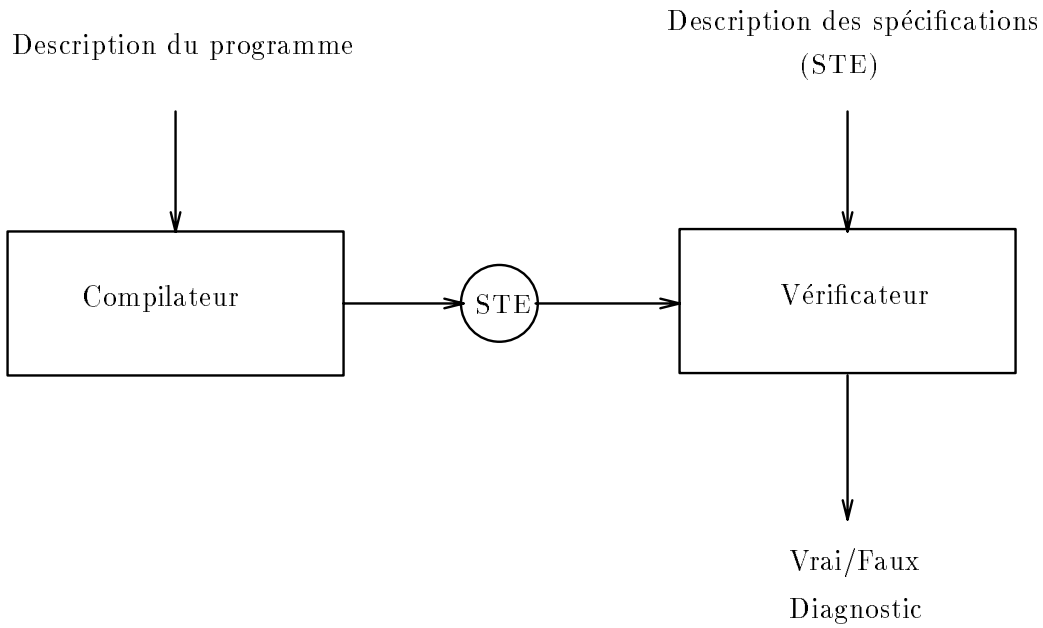
La procédure de décision "classique" associée à la bisimulation forte est basée sur un algorithme de *minimisation* qui permet de calculer le *quotient* d'un système de transitions étiquetées vis-à-vis de cette relation (i.e., le plus petit système de transitions étiquetées qui lui soit équivalent). Plus précisément, ce quotient est construit en procédant par *raffinements* successifs d'une *partition* initiale de l'ensemble des états du système de transitions étiquetées jusqu'à la rendre *compatible* avec sa relation de transition (i.e., deux états sont dans une même classe si et seulement si ils peuvent évoluer via la relation de transition vers deux états dans une même classe). La partition finale ainsi obtenue permet alors de déterminer les états du quotient. Par suite, comparer deux systèmes de transitions étiquetées revient à vérifier si leurs quotients sont ou non identiques. Plusieurs algorithmes de raffinements ont été proposés, parmi lesquels l'algorithme de Paige et Tarjan [PT87] dont la complexité en temps est de $O(m.\log n)$, où n dénote le nombre d'états du système et m le cardinal de sa relation de transition.

Pour étendre cette procédure de décision aux autres relations de bisimulation, les algorithmes usuels reposent alors sur deux phases : on effectue dans un premier temps un certain nombre de *transformations* sur les relations de transition des deux systèmes, puis on compare modulo la bisimulation forte les nouveaux systèmes ainsi obtenus. Ces transformations, qui dépendent de la relation d'équivalence considérée, sont en général basées sur des calculs de *fermeture transitive*.

Sur le plan pratique, la vérification de spécifications comportementales nécessite donc deux types d'outils :

- un *compilateur*, dont le rôle est de traduire le programme à vérifier en un système de transitions étiquetées (STE) qui modélise l'ensemble de ses exécutions possibles,
- et un *vérificateur*, qui implémente un certain nombre de procédures de décision associées à des relations d'équivalence ou de préordre, et qui permet ainsi de comparer le système de transitions étiquetées représentant le programme à celui représentant ses spécifications. Notons qu'en pratique le rôle du vérificateur doit être également de fournir des éléments de diagnostic lorsque le programme à vérifier est incorrect.

D'une manière générale, ces deux outils s'organisent alors selon l'architecture "classique" suivante :



De nombreux logiciels pour la vérification de spécifications comportementales basés sur cette architecture ont été réalisés à ce jour (tels ALDÉBARAN [Fer88], AUTO [dSV89], TAV [GLZ89], CONCURRENCY WORKBENCH [CPS89], BRANCHING TOOL [GV90], *etc* ...). On trouve une présentation et une comparaison détaillée de la plupart de ces logiciels dans [Kor91b] et [IP91].

Toutefois, en dépit de l'efficacité des algorithmes mis en œuvre, cette approche souffre d'une restriction importante due au fait qu'elle nécessite de *construire* et *mémoriser* l'ensemble du système de transitions étiquetées qui modélise le programme à vérifier (c'est à dire l'ensemble de ses états et de sa relation de transition). Par conséquent, la taille des systèmes qui peuvent être traités reste limitée et il s'avère en pratique que cette limite est rapidement atteinte lorsque l'on s'intéresse à des programmes réalistes. Ce problème, souvent désigné sous le terme de *problème de l'explosion d'états* — qui est commun à toutes les méthodes de vérification basées sur les modèles — fait l'objet d'un nombre important de recherches depuis quelques années.

Dans ce contexte, les objectifs de notre travail sont l'étude de nouvelles procédures de décision pour les relations de bisimulation qui doivent permettre de comparer un programme à ses spécifications comportementales sans construire au préalable l'ensemble du système de transitions étiquetées qui lui correspond. D'un point de vue pratique, cette étude s'intègre dans le cadre de la réalisation d'une boîte à outils [FGM+91] pour la vérification de programme LOTOS, articulée autour du compilateur CÆSAR [Gar89, GS90a] et du vérificateur ALDÉBARAN [Fer88, FM91b]. L'intérêt de considérer un langage de description normalisé comme LOTOS est alors de pouvoir confronter les algorithmes obtenus à de "vrais" exemples.

Réduire la taille du modèle

Pour remédier au problème de l'explosion d'états, une première solution consiste à effectuer la vérification non plus à partir d'un système de transitions étiquetées représentant l'ensemble des exécutions du programme mais à partir d'un espace d'états plus réduit, qui peut éventuellement dépendre de la propriété à vérifier. Nous présentons ici un certain nombre d'approches qui ont été envisagées pour générer ce "modèle réduit" :

- En premier lieu, il est souvent possible de tirer parti du caractère *compositionnel* du langage utilisé pour décrire le programme. Plus précisément, dans le cas des algèbres de processus, un programme $\mathcal{P}$ est généralement représenté comme la *composition parallèle* d'un ensemble de termes, chaque terme pouvant lui-même être modélisé par un système de transitions étiquetées $\mathcal{P}_i$. Par suite, plutôt que de générer directement le système de transitions étiquetées correspondant au programme complet, une solution possible consiste à effectuer simultanément sur l'ensemble des $\mathcal{P}_i$ des opérations de *composition* et de *minimisation* modulo une relation d'équivalence donnée. Toutefois, en l'absence de précautions supplémentaires dans l'ordonnancement de ces deux opérations, et du fait que chaque composition effectuée soit *locale* à quelques uns des $\mathcal{P}_i$, il est possible que la taille des systèmes intermédiaires engendrés soit supérieure à celle qu'aurait eu le système résultat obtenu de manière classique.

  Pour remédier à ce problème, des *stratégies* de réduction sont proposés dans [Fer88] et [dSV89], et une méthode qui permet de prendre en compte l'*environnement* lors des compositions est décrite dans [GS90b]. Des approches similaires sont également présentés dans [CLM89] pour la vérification de spécifications logiques, et dans [VT91], pour la vérification de spécifications comportementales modulo une relation d'équivalence particulière.

- Une seconde approche consiste à remettre en cause la solution habituellement retenue pour transformer une description *parallèle* du programme en un modèle purement *séquentiel* — le système de transitions étiquetées — et qui est basée sur la technique de l'*entrelacement* (ou *interleaving*) : le fait que deux événements (ou actions) puissent avoir lieu de façon simultanée est représenté par l'ensemble des entrelacements possibles de ces deux événements, *dans n'importe quel ordre*. Pour réduire la taille du modèle, on peut vouloir de ne pas construire explicitement cet ensemble de séquences, mais uniquement *un seul* de ses éléments. Les séquences "manquantes" peuvent alors être retrouvées à partir de ce représentant unique grâce à une *relation de dépendance* qui définit toutes les permutations possibles sur l'ensemble de ses actions. De plus, l'efficacité de la méthode de vérification mise en œuvre sur ce modèle réduit est souvent améliorée en tirant parti du fait que la propriété que l'on vérifie peut être indépendante de l'ordre dans lequel ont lieu une partie de l'ensemble des actions du système.

  Cette solution a été appliquée au problème de la vérification de spécifications logiques [Val90, GW91a], à celui de de la vérification des *propriétés de sûreté* des systèmes [GW91b], ou encore à celui de la vérification de spécification comportementales modulo une relation particulière [God90]. Néanmoins, l'extension possible de cette approche au cas des relations de bisimulation ne parait pas immédiate.

- Enfin, dans le cas particulier où le programme à vérifier est constitué d'un ensemble de N composants identiques, il est possible de mettre en œuvre des méthodes de vérification *par induction* comme celles décrites par exemple dans [CG87] ou [WL89] qui permettent de ne pas avoir à générer un système de transitions étiquetées représentant le comportement de ces N composants.

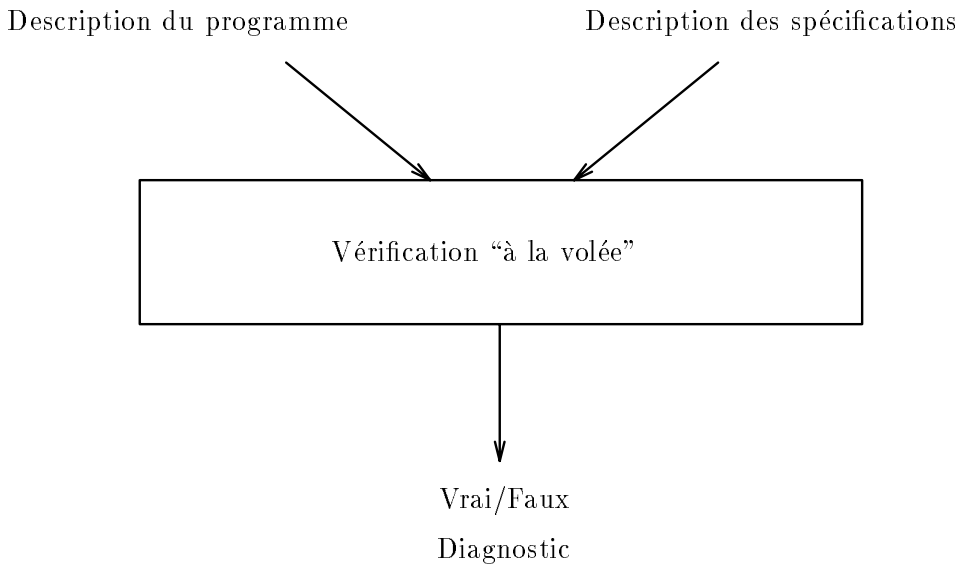
Toujours dans le but de réduire la taille du modèle, une autre méthode, radicalement différente, fait actuellement l'objet de nombreux travaux. Contrairement aux solutions précédentes, elle ne consiste pas à diminuer le nombre des états du modèle, mais plutôt à en adopter une *représentation symbolique* qui soit plus efficace en mémoire, et qui évite d'avoir à énumérer de manière exhaustive cet ensemble d'états lors de la vérification. En pratique, la solution la plus fréquemment utilisée consiste à modéliser le programme à l'aide d'un *ensemble de fonctions booléennes*, implémentées au moyen de *graphes de décision binaires* (ou BDD [Bry86]). Les algorithmes de vérification reposent alors sur des opérations symboliques entre les éléments de cet ensemble.

La vérification symbolique a été étudiée aussi bien dans le cas de spécifications exprimées à l'aide d'une logique temporelle [CMB90, BCM+90, RHR91], que dans le cas de spécifications comportementales [BFH90a, BCM+90, EFT91]. Toutefois, l'efficacité de cette approche dépend encore largement de la façon dont sont obtenues les fonctions booléennes qui modélisent les programmes, et, en particulier, son application à des langages qui permettent de définir des valeurs sur des domaines quelconques ne semble pas évidente à l'heure actuelle.

Vérifier à la "volée"

Parallèlement à ces premières solutions qui avaient pour but de générer directement un système de transitions étiquetées réduit, il existe également une seconde approche pour résoudre le problème de l'explosion d'états qui consiste à effectuer la vérification *au fur et à mesure* de la génération de ce système, et surtout sans avoir à le mémoriser dans son intégralité. Cette approche, peut être moins ambitieuse que les précédentes, est généralement désignée sous le terme de vérification "à la volée".

Cette approche peut donc être représentée par l'architecture suivante, dans laquelle le compilateur et le vérificateur sont en fait intégrés dans un même outil :



Compte-tenu de ce principe, pour qu'une propriété puisse être vérifiée "à la volée", il est nécessaire qu'il lui soit associé une procédure de décision qui repose sur un algorithme de parcours du système de transitions étiquetées qui soit "compatible" avec celui mis en œuvre au moment de la génération.

A l'heure actuelle, cette condition a déjà pu être obtenue pour différentes classes de propriétés :

- Les premières applications de la vérification "à la volée" ont été consacrés au problème de l'*analyse d'accessibilité* [Hol85, Hol87], qui consiste à s'assurer qu'un programme ne contient pas de *blocage*.
- Par la suite, cette méthode a alors été étendue aux propriétés de *sûreté*, spécifiées à partir de formules de logiques temporelles [JJ89, BFH90b].
- Enfin, plus récemment, des algorithmes de parcours plus sophistiqués ont permis de vérifier "à la volée" une classe de propriétés plus générales qui peuvent être spécifiées à l'aide d'automates

de Büchi [CVWY90, JJ91].

Vis-à-vis des objectifs que nous nous sommes fixés, la vérification "à la volée" présente un certain nombre d'avantages :

- Du fait que la vérification est basée sur une *énumération* de l'ensemble des états du système de transitions étiquetées qui modélise le programme, sa *relation de transition* n'a jamais a être mémorisée. De plus, il existe des algorithmes qui permettent de mettre efficacement en œuvre ce type d'énumération sans même mémoriser dans son intégralité l'ensemble des *états* de ce système [Hol89, JJ89].

- Cette approche est entièrement indépendante du langage choisi pour décrire le programme, dans le sens où elle ne repose pas sur une propriété quelconque de ce langage. Elle peut donc être appliquée à n'importe quel formalisme — à la condition toutefois que celui-ci permette de générer un système de transitions étiquetées fini — et elle ne nécessite pas de remettre profondément en cause l'architecture du compilateur.

- Enfin, signalons également que certaines méthodes de réduction qui permettent de générer directement un système de transitions étiquetées réduit restent compatible avec une méthode de vérification "à la volée" : la vérification s'effectue alors simplement au fur et à mesure de la génération du modèle réduit.

Par conséquent, il nous a paru intéressant de pouvoir appliquer cette approche à la vérification de spécifications comportementales. Nous présentons donc dans ce travail de nouvelles procédures de décision pour diverses relations de bisimulation qui peuvent être mises en œuvre lors d'un parcours simultané des deux systèmes de transitions étiquetées à comparer, et qui ont été implémentées au sein du logiciel ALDÉBARAN.

Organisation du document

Ce document est composé de trois parties.

La **première partie** présente les deux familles de relations dédiées à la vérification de spécifications comportementales auxquelles nous nous sommes plus particulièrement intéressés, qui sont respectivement les relations de simulation et de bisimulation.

Le chapitre 1 introduit un certain nombre de définitions générales concernant les systèmes de transitions étiquetées et les relations d'équivalence qui leurs sont associées.

Le chapitre 2 est consacré aux préordres de simulation et équivalences de bisimulation. Nous présentons tout d'abord de manière intuitive ces deux familles de relations à travers la notion de *comportement observable*, puis nous en donnons une définition formelle, paramétrée par un ensemble de *langages d'actions* Λ. A chaque valeur de ce paramètre sera associée une relation particulière, qui correspond à un certain *critère d'abstraction*.

Tout au long de ce travail, nous nous sommes attachés à un petit nombre de relations de bisimulation et de simulation qui sont respectivement la bisimulation forte, l'équivalence observationnelle, la $\tau^* a$-bisimulation, la delay bisimulation et la bisimulation de branchement, ainsi que l'équivalence et le préordre de sûreté. Nous précisons les valeurs du paramètre Λ qui permettent de définir chacune de ces relations, en essayant à chaque fois de montrer leur intérêt en pratique.

Enfin, nous terminons ce chapitre en comparant ces relations entre elles vis-à-vis de la relation d'inclusion, et en précisant leurs positions relatives dans le treillis des partitions. Cette com-

paraison est d'abord effectuée dans le cas général puis en considérant certains cas particuliers qui nous semblent intéressants du point de vue de la vérification.

Les deux autres parties sont consacrées aux méthodes de vérification de spécifications comportementales, et, plus précisément, à la comparaison de deux systèmes de transitions étiquetées à l'aide d'une relation d'équivalence définie sur l'ensemble de leurs états. En effet, partant du constat qu'une relation d'équivalence peut être caractérisée soit par une partition (i.e., l'ensemble de ses classes d'équivalence), soit par une partie d'un produit cartésien, deux approches sont envisageables.

La **deuxième partie** présente l'approche "classique" dans le cas d'une relation de bisimulation, qui consiste à raffiner une partition initiale de l'ensemble des états des deux systèmes jusqu'à obtenir les classes d'équivalence pour cette relation. Les algorithmes mis en œuvre reposent donc sur des calculs de plus grand points fixes d'opérateurs de raffinements définis sur le treillis des partitions.

Le chapitre 3 décrit les procédures de décision usuelles qui sont associées aux relations de bisimulation et de simulation, et en particulier celles implémentées dans le logiciel ALDÉBARAN. Nous donnons tout d'abord les principes généraux des algorithmes de minimisation et de comparaison en rappelant la notion de *compatibilité* d'une partition par rapport à une relation de transition, ainsi que celle de *quotient* d'un système de transitions étiquetées modulo une relation de bisimulation. La définition paramétrée d'équivalence de bisimulation donnée au premier chapitre permet d'obtenir une présentation générale de ces algorithmes, valide pour toute relation de bisimulation.

Nous décrivons alors plus précisément les procédures de décision utilisées en pratique pour les relations que nous avons choisi de considérer. Nous rappelons donc le principe de l'algorithme de raffinement proposé par Paige & Tarjan dans le cas de la bisimulation forte, puis nous présentons les algorithmes de *transformation* de la relation de transition qui sont utilisées dans le cas des autres relations d'équivalence. Là encore, nous avons essayé d'adopter une démarche qui soit la plus générale possible en utilisant la notion de *forme pre-normale* introduite dans [Fer89]. Nous précisons également pour chaque relation le coût des algorithmes obtenus.

Le chapitre 4 est consacré à un problème assez peu étudié mais qui nous semble important en pratique et qui est le calcul d'un *diagnostic* lorsque les deux systèmes de transitions étiquetées que l'on compare ne sont pas identifiés par la relation d'équivalence (ou de préordre) considérée. Nous présentons un formalisme qui nous parait approprié pour exprimer la *non-équivalence* et qui est basé sur la notion de *séquence diagnostique* : il s'agit d'un couple de *séquences d'exécutions* de ces deux systèmes, qui, à partir de leurs états initiaux, conduisent via des actions observables identiques, à deux états où la non-équivalence apparaît clairement (i.e., des états *erreurs*). Nous définissons formellement ces séquences et nous justifions leur existence.

Dans le but d'offrir à l'utilisateur un diagnostic plus facile à exploiter, nous nous intéressons alors au problème de la recherche de *séquences diagnostiques minimales*. Nous proposons donc trois relations d'ordre sur l'ensemble des séquences d'exécution d'un système de transitions étiquetées, ainsi que des algorithmes qui permettent de calculer les séquences diagnostiques minimales associées à chacune de ces relations.

La **troisième partie** présente la méthode que nous avons développée pour comparer des systèmes de transitions étiquetées "à la volée". Son principe consiste à construire un sous-ensemble de la relation d'équivalence (ou de préordre) que l'on considère, en examinant uniquement des couples d'états accessibles à partir des états initiaux des deux systèmes. L'algorithme proposé repose donc sur le calcul du plus grand point fixe d'un opérateur défini sur le treillis des relations binaires.

Le chapitre 5 décrit cette approche de façon détaillée dans le cas d'une relation de bisimulation ou de simulation quelconque. Plus précisément, on montre tout d'abord comment l'existence d'une telle relation entre deux systèmes peut être exprimée comme une propriété sur l'ensemble

des séquences d'exécution élémentaires d'un *produit synchrone* défini entre ces systèmes. A partir de ce critère d'équivalence on déduit un premier algorithme dont la complexité en temps est exponentielle en fonction de la taille de ce produit synchrone.

Nous présentons alors deux améliorations pour cet algorithme qui permettent de ramener cette complexité à $O(m.n)$ dans le cas général et à $O(m)$ lorsque l'un des deux systèmes est *déterministe*, avec n et m désignant respectivement le nombre d'états et de transitions du produit synchrone. Nous justifions formellement la correction de ces algorithmes.

Le chapitre 6 décrit la mise en œuvre des algorithmes de comparaison proposés au chapitre précédent, l'objectif étant de montrer la faisabilité d'une vérification "à la volée" à partir de cette approche. Nous précisons donc d'abord comment ces algorithmes généraux peuvent être adaptés efficacement à la plupart des relations définies au chapitre 1, puis nous présentons les structures de données que nous avons retenues pour implémenter le parcours du produit synchrone. Nous discutons alors de l'amélioration possible de cette implémentation en adoptant des techniques plus efficaces en mémoire pour mettre en œuvre ce parcours ([Hol89] et [JJ89]).

Nous terminons ce chapitre en décrivant plus spécifiquement les nouvelles fonctionnalités que nous avons ainsi apporté au logiciel ALDÉBARAN, puis nous comparons à partir de différents exemples les nouvelles procédures de décision ainsi obtenues au procédures de décision "classiques" qui étaient déjà implémentées dans ce logiciel.

Enfin, nous présentons en **Annexes** un exemple de vérification d'un *protocole de diffusion atomique* [BM91].

Partie I

Relations d'équivalence comportementales

Chapitre 1

Définitions préliminaires

Nous introduisons dans cette partie un certain nombre de définitions préliminaires concernant les systèmes de transitions étiquetées et les relations d'équivalence.

1.1 Notations

On utilisera dans la suite les opérateurs ensemblistes usuels suivants :

- $\emptyset$ dénote l'ensemble vide ;
- Pour tous ensembles E_1 et E_2,
  - $E_1 \cup E_2$ représente *l'union* de E_1 et E_2 ;
  - $E_1 \cap E_2$ représente *l'intersection* de E_1 et E_2 ;
  - $E_1 - E_2$ représente *la différence ensembliste* de E_1 et E_2 ;
  - $E_1 \times E_2$ représente *le produit cartésien* de E_1 et E_2 ;
  - $E_1 \subseteq E_2$ représente la relation *d'inclusion* entre E_1 et E_2 ;
  - $E_1 \subset E_2$ représente la relation *d'inclusion stricte* entre E_1 et E_2 ;
- Pour tout ensemble E,
  - $|E|$ représente *le cardinal* de E ;
  - $min\,(E)$ (*resp.* $max\,(E)$) représente, lorsqu'il existe, le *plus petit* (*resp.* le *plus grand*) élément de E pour une relation d'ordre donnée.

Pour tout ensemble E, on désignera par E^* (*resp.* E^∞) l'ensemble des *séquences finies* (*resp.* des *séquences finies ou infinies*) d'éléments de E. Plus précisément, en notant

- $E^0 = \emptyset$,
- $E^1 = E$,
- et, $\forall i \geq 1$, $E^{i+1} = E^i \times E$,

on a :

$$E^* = \bigcup_{i=0}^{i<\infty} E^i$$

$$E^\infty = \bigcup_{i=0}^{\infty} E^i$$

Clairement, on a $E^* \subseteq E^\infty$.

Enfin, pour tout élément $\omega = (x_1, x_2, \ldots, x_k)$ de E^*, on notera :

- $\omega(i)$, le i^{eme} élément de ω :
  $\forall i,\ 1 \leq i \leq k\ .\ \omega(i) = x_i$
- $|\omega|$, la *longueur* de ω : $|\omega| = k$

1.2 Systèmes de transitions étiquetées

1.2.1 Définitions

On considère un ensemble $\mathcal{Q}$ dont les éléments sont appelés *états*, un ensemble $\mathcal{A}$ dont les éléments sont appelés *étiquettes* (ou *noms d'actions*), et une étiquette particulière, τ, non incluse dans $\mathcal{A}$.

Définition 1.2-1
Un système de transitions étiquetées S est un quadruplet (Q, A, T, q_0), où :

- Q désigne l'ensemble des états de S, $Q \subseteq \mathcal{Q}$,
- A représente l'ensemble des actions de S, $A \subseteq \mathcal{A} \cup \{\tau\}$,
- T est la relation de transition, $T \subseteq Q \times (A \cup \{\tau\}) \times Q$,
- q_0 est un élément particulier de Q, appelé *état initial* de S.

■

Remarque 1-1
Un système de transitions étiquetées peut donc être vu comme un graphe orienté, muni d'un état initial, et dont les arcs sont étiquetés par un label. Par conséquent, l'ensemble du vocabulaire défini pour les graphes reste valide, en particulier les notions usuelles de *chemin*, *chaîne*, *cycle* et *circuits*.
■

Intuitivement, le lien entre un système de transitions étiquetées S et un programme $\mathcal{P}$ est immédiat :

- Les états de S représentent les états de $\mathcal{P}$, définis par la valeur de ses variables et de son compteur ordinal ;
- La relation de transition de S définit les changement d'états du programme lors de l'exécution d'une instruction (à chaque instruction étant attaché un label).

Formellement, ce lien peut être établi à l'aide d'une sémantique opérationnelle du langage dans lequel $\mathcal{P}$ est décrit, qui indique alors comment associer à chaque programme un système de transitions étiquetées qui le modélise.

Pour un système de transitions étiquetées $S = (Q, A, T, q_0)$ et p un élément de Q, nous utiliserons les notations suivantes :

- Le prédicat $(p, a, q) \in T$ sera noté $p \xrightarrow{a}_T q$ (ou $p \xrightarrow{a} q$, s'il n'y a pas ambiguïté sur la relation de transition). L'état q est alors défini comme un *successeur* de p par l'action a. Un état sans successeur sera appelé un état *puits*.

- On désignera par $T[p]$ l'image de l'état p par la relation de transition T, et par $T_a[p]$ son image par la relation de transition *étiquetée* T_a :
$$T[p] = \{(a, q) \in A \times Q \mid p \xrightarrow{a}_T q\}$$
$$T_a[p] = \{q \in Q \mid p \xrightarrow{a}_T q\}$$
$T_a[p]$ représente alors l'ensemble des successeurs de p par l'action a.

- Ces notations pourront également être étendues à la relation de transition inverse T^{-1} :
$$T^{-1}[p] = \{(a, q) \in A \times Q \mid q \xrightarrow{a}_T p\}$$
$$T_a^{-1}[p] = \{q \in Q \mid q \xrightarrow{a}_T p\},$$
et $T_a[p]$ désigne alors l'ensemble des *prédécesseurs* de p par l'action a.

- On notera de manière similaire les successeurs et les prédécesseurs d'un *ensemble d'états* B de Q ($B \subseteq Q$) :
$$T[B] = \bigcup_{p \in B} T[p]$$
$$T_a[B] = \bigcup_{p \in B} T_a[p]$$
et,
$$T^{-1}[B] = \bigcup_{p \in B} T^{-1}[p]$$
$$T_a^{-1}[B] = \bigcup_{p \in B} T_a^{-1}[p]$$

- Enfin, nous considérerons également l'ensemble $\mathcal{A}\mathrm{ct}(p)$ des actions pour lesquelles p admet un successeur par T (i.e, les actions qui peuvent être effectuées à partir de p) :
$$\mathcal{A}\mathrm{ct}(p) = \{a \in A \mid \exists q \in Q \,.\, p \xrightarrow{a}_T q\}$$
et, pour $B \subseteq Q$,
$$\mathcal{A}\mathrm{ct}(B) = \bigcup_{p \in B} \mathcal{A}\mathrm{ct}(p)$$

Remarque 1-2
Les ensembles $T_a[B]$ et $T_a^{-1}[B]$ coïncident avec les images de B par les fonctions de *pre-* et *post-conditions* (ou *transformateurs de prédicats*) de 2^Q vers 2^Q :

$$pre_a(B) = T_a^{-1}[B] = \{q \in Q \mid \exists p \,.\, q \xrightarrow{a} p \ \wedge \ p \in B\}$$
$$post_a(B) = T_a[B] = \{q \in Q \mid \exists p \,.\, p \xrightarrow{a} q \ \wedge \ p \in B\}$$

■

Il est souvent intéressant de considérer les successeurs et les prédécesseurs d'un état (ou d'un ensemble d'états) non plus pour une action donnée, mais pour un *langage d'actions* λ défini sur l'ensemble $\mathcal{A}^*$ des séquences de $\mathcal{A}$ ($\lambda \subseteq \mathcal{A}^*$). Intuitivement, les notations suivantes permettent d'effectuer des *abstractions* sur l'ensemble des actions du système, en définissant *l'action abstraite* λ comme étant un ensemble de séquences d'actions concrètes de $\mathcal{A}$:

- Pour $\lambda \subseteq \mathcal{A}^*$, et pour $p, q \in Q$, on notera $p \xrightarrow{\lambda}_T q$ le prédicat défini par :
$$\lambda \ni a_1 \cdots a_n \ \wedge \ \exists q_1, \cdots, q_{n-1} \in Q \,.\, p \xrightarrow{a_1} q_1 \ \wedge \ q_1 \xrightarrow{a_2} q_2 \ \wedge \ \cdots \ \wedge \ q_{n-1} \xrightarrow{a_n} q.$$

- Les prédécesseurs et les successeurs d'un état p (ou d'un ensemble d'états B) par λ sont respectivement notés :

$$\begin{aligned} T_\lambda[p] &= \{q \in Q \mid p \xrightarrow{\lambda}_T q\} \\ T_\lambda[B] &= \bigcup_{p \in B} T_\lambda[p] \end{aligned}$$

et

$$\begin{aligned} T_\lambda^{-1}[p] &= \{q \in Q | q \xrightarrow{\lambda}_T p\} \\ T_\lambda^{-1}[B] &= \bigcup_{p \in B} T_\lambda^{-1}[p] \end{aligned}$$

- Enfin, si Λ dénote un ensemble de langages λ définis sur $\mathcal{A}^*$ ($\Lambda \subseteq 2^{\mathcal{A}^*}$), l'ensemble $\mathcal{A}\mathrm{ct}_\Lambda(p)$ des langages (ou actions abstraites) de Λ pour lesquelles p admet un successeur est défini par :

$$\mathcal{A}\mathrm{ct}_\Lambda(p) = \{\lambda \in \Lambda \mid \exists q \in Q \,.\, p \xrightarrow{\lambda}_T q\}$$

et, pour $B \subseteq Q$,

$$\mathcal{A}\mathrm{ct}_\Lambda(B) = \bigcup_{p \in B} \mathcal{A}\mathrm{ct}_\Lambda(p)$$

La définition suivante permet de caractériser les systèmes de transitions étiquetées en fonction de la structure de leur relation de transition :

Définition 1.2-2
Soit $S = (Q, A, T, q_0)$ un système de transitions étiquetées :

- S est dit *fini* si et seulement si il possède un nombre fini d'états,
- S est dit *à branchement fini* si et seulement si chacun de ses états a un nombre fini de successeurs :

  $\forall p \in Q \,.\, |T[p]| < \infty$,

  Pour un système S à branchement fini, on appelle *facteur de branchement* c de S, le rapport entre les cardinaux de sa relation de transition et de l'ensemble de ses états :

  $$c = \frac{|T|}{|Q|}$$

- S est dit *déterministe* si et seulement si chacun de ses états a au plus un successeur par une action donnée :

  $\forall p \in Q \,.\, \forall a \in \mathcal{A} \,.\, |T_a[p]| \leq 1.$

  Là encore, cette notation sera étendue aux langages d'actions de la façon suivante : si Λ est un ensemble de langages définis sur $\mathcal{A}$, S est dit *déterministe pour* Λ si et seulement si

  $\forall p \in Q \,.\, \forall \lambda \in \Lambda \,.\, |T_\lambda[p]| \leq 1.$

■

Dans la suite on ne considérera que des systèmes de transitions étiquetées finis, à branchement fini, déterministes ou non.

Enfin, nous définissons une *relation d'égalité* sur l'ensemble des systèmes de transitions étiquetées, qui permet d'identifier des systèmes qui sont identiques aux noms des états près. Cette relation peut donc être vue comme un isomorphisme entre graphes, et un système de transitions étiquetées comme une classe de graphes isomorphes.

Définition 1.2-3
Soient $S_i = (Q_i, A_i, T_i, q_{0i})_{i=1,2}$ deux systèmes de transitions étiquetées. S_1 et S_2 sont dits *égaux* (ou *identiques au noms des états près*), et on note $S_1 = S_2$ si et seulement si :

- $A_1 = A_2$
- Il existe une *bijection* Φ de Q_1 dans Q_2 telle que :
  - $q_{02} = \Phi(q_{01})$
  - $T_2 = \{(\Phi(p), a, \Phi(q)) \mid (p, a, q) \in T_1\}$

■

1.2.2 Séquences d'exécution

Après avoir présenté les systèmes de transitions étiquetées en tant que modèle formel pour représenter des programmes, il reste à préciser la notion de *séquence d'exécution*.

Définition 1.2-4
Soit $S = (Q, A, T, q_0)$ un système de transitions étiquetées, et soit Λ un ensemble de langages définis sur A^*.

- Pour tout état p_0 de Q, on note $Ex(p_0)$ l'ensemble des séquences d'exécution (finies) sur S issues de p_0 :
$$Ex(p_0) = \{(p_0, a_0, p_1, \ldots, a_{k-1}, p_k) \in Q \times (A \times Q)^* \mid \forall i,\, 0 \leq i < k-1 \,.\, p_i \xrightarrow{a_i}_T p_{i+1}\}$$
- De façon similaire, en considérant les langages λ_i de Λ comme des actions abstraites, on définit l'ensemble $Ex_\Lambda(p_0)$ de la manière suivante :
$$Ex_\Lambda(p_0) = \{(p_0, \lambda_0, p_1, \ldots, \lambda_{k-1}, p_k) \in Q \times (\Lambda \times Q)^* \mid \forall i,\, 0 \leq i < k-1 \,.\, p_i \xrightarrow{\lambda_i}_T p_{i+1}\}$$

■

On étend alors cette notation aux systèmes de transitions étiquetés eux-mêmes en définissant l'ensemble des séquences d'exécutions d'un système comme étant l'ensemble des séquences d'exécution issues de son état initial :

Définition 1.2-5
Soit $S = (Q, A, T, q_0)$ un système de transitions étiquetées, et soit Λ un ensemble de langages définis sur A^* :
$$\begin{aligned} Ex(S) &= Ex(q_0) \\ Ex_\Lambda(S) &= Ex_\Lambda(q_0) \end{aligned}$$

■

On termine alors cette section en donnant des notations et définitions complémentaires sur les séquences d'exécutions :

Définition 1.2-6
Soit $S = (Q, A, T, q)$ un système de transitions étiquetées, et soit $\sigma = (p_1, \lambda_1, p_2, \ldots, \lambda_{k-1}, p_k)$ une séquence d'exécution sur S, avec $\sigma \in Ex_\Lambda(p_1)$:

- Pour tout $1 \leq i \leq k$, on désigne par $\sigma(i)$ le i^{eme} *état* de σ : $\sigma(i) = p_i$;
- Le nombre d'états de σ (ou *longueur* de σ) est noté $|\sigma|$: $|\sigma| = k$;
- Le dernier élément de σ est noté $last(\sigma)$: $last(\sigma) = \sigma(|\sigma|) = p_k$

- σ est une séquence d'exécution *élémentaire* si et seulement si tout ses états sont distincts :

  $$\forall i,\ 0 \leq i \leq k,\ \forall j\ .\ i \neq j\ \Rightarrow\ \sigma(i) \neq \sigma(j)$$

  Par suite, si Q est fini alors toute séquence élémentaire de S est finie et l'ensemble des séquences élémentaires de S est également fini.

- σ est une séquence d'exécution *circuit* si et seulement si $\sigma(0) = \sigma(k) = p_0$.

Enfin, pour toutes séquences d'exécution σ et σ' définies sur S et satisfaisant :

$$\sigma = (p_0, \lambda_0, p_1, \ldots, \lambda_{k-1}, p_k)\ \wedge\ \sigma' = (p'_0, \lambda'_0, p'_1, \ldots, \lambda'_{k'-1}, p'_{k'})\ \wedge\ \exists \lambda\ .\ p_k \xrightarrow{\lambda}_T p'_0,$$

on note $(\sigma, \lambda, \sigma')$ la séquence obtenue par *concaténation* de σ et σ' :

$$(\sigma, \lambda, \sigma') = (p_0, \lambda_0, p_1, \ldots, \lambda_{k-1}, p_k, \lambda, p'_0, \lambda'_0, p'_1, \ldots, \lambda'_{k'-1}, p'_{k'})$$

■

1.3 Relations d'équivalence et partitions

On rappelle quelques définitions et notations générales concernant les relations d'équivalence et les partitions définies sur un ensemble Q.

1.3.1 Définitions

Définition 1.3-1
Pour tout ensemble Q, une *relation d'équivalence* R définie sur Q est une partie de Q^2 ($R \subseteq Q \times Q$) telle que :

- R est *réflexive* :

  $$\forall q \in Q\ .\ (q, q) \in R.$$

- R est *symétrique* :

  $$\forall (p, q) \in Q^2\ .\ (p, q) \in R\ -\ (q, p) \in R.$$

- R est *transitive* :

  $$\forall (p, q, r) \in Q^3\ .\ ((p, q) \in R\ \wedge\ (q, r) \in R)\ \Rightarrow\ (p, r) \in R$$

D'autre part, R est *relation de préordre* si et seulement si R est réflexive et transitive. ■

Remarque 1-3
Dans la suite, pour toute relation binaire R, on emploiera indifféremment les deux notations équivalentes

$$(p, q) \in R\ \text{ et }\ p\ R\ q.$$

■

Définition 1.3-2
On appelle *partition* d'un ensemble Q, un ensemble $\rho = \{X_i \mid i \in \mathcal{I}\}$ de parties de Q ($\forall i \in \mathcal{I}\ .\ X_i \subseteq Q$) tel que :

- Les éléments X_i de ρ sont *deux à deux disjoints* :

  $$\forall (i, j) \in \mathcal{I}\ .\ i \neq j\ \Rightarrow\ X_i \cap X_j = \emptyset$$

- ρ définit un *recouvrement* de Q :

$$\bigcup_{i \in \mathcal{I}} X_i = Q$$

■

A toute relation d'équivalence R définie sur Q, il est possible d'associer une partition ρ_R de Q telle que deux éléments de Q appartiennent au même élément de ρ_R si et seulement si ils sont équivalents par R. Réciproquement, à toute partition définie sur Q, il est possible d'associer une relation d'équivalence $\sim_\rho$ telle deux éléments de Q soient équivalents par $\sim_\rho$ si et seulement si ils appartiennent au même élément de ρ. Plus formellement :

Définition 1.3-3
Soit R une relation d'équivalence définie sur un ensemble Q. $\rho_R = \{X_i \mid i \in \mathcal{I}\}$ est la *partition associée* à R si et seulement si :

$$\forall (p,q) \in Q^2 \,.\, pRq \;-\; \exists i \in \mathcal{I} \,.\, \{p,q\} \subseteq X_i$$

Les éléments de ρ_R sont alors appelées les *classes d'équivalence* de R. De plus, pour tout état p de Q, on note $[p]_R$ la classe de ρ qui le contient. On a alors :

$$\rho_R = \{[p]_R \mid p \in Q\}$$

Enfin, pour une partition ρ donnée, on note $\sim_\rho$ la relation d'équivalence à laquelle elle est associée.
■

1.3.2 Treillis des relations et treillis des partitions

L'ensemble des relations d'équivalence sur Q^2, muni de l'opérateur d'inclusion $\subseteq$, forme un treillis complet (qui est un sous-treillis du treillis des relations binaires) et dont les extremums sont respectivement :

- Le *maximum* est la relation *universelle* Q^2, dans laquelle chaque élément est en relation avec tous les autres ;
- Le *minimum* est la relation *identité* $\{(p,p) \mid p \in Q\}$, dans laquelle chaque élément est en relation uniquement avec lui-même.

L'ensemble des opérateurs définis sur les relations d'équivalence (i.e., de 2^Q vers 2^Q) admet respectivement l'intersection $\cap$ et l'union $\cup$ comme borne inférieure et borne supérieure.

On dira qu'une relation R_1 est *plus forte* qu'une relation R_2 si et seulement si R_1 *est contenue* dans R_2 ($R_1 \subseteq R_2$). Clairement, si R_1 est plus forte que R_2, alors deux éléments de Q équivalents modulo R_1 seront équivalents modulo R_2 :

$$R_1 \subseteq R_2 \;-\; \forall (p,q) \in Q^2 \,.\, p\, R_1\, q \;\Rightarrow\; p\, R_2\, q$$

Enfin, pour toute fonction totale, croissante, $\mathcal{F}$ de 2^Q vers 2^Q on notera :

- $\mu\pi.\mathcal{F}(\pi)$ le plus petit point fixe de $\mathcal{F}$ par rapport à $\subseteq$
- $\nu\pi.\mathcal{F}(\pi)$ le plus grand point fixe de $\mathcal{F}$ par rapport à $\subseteq$

De façon similaire, il est possible de munir l'ensemble des partitions sur Q (i.e., une partie de 2^{2^Q}) d'une relation d'ordre $\sqsubseteq$ définie de la manière suivante :

Définition 1.3-4
Soient ρ et ρ' des partitions définies sur Q. On dit que ρ est un *raffinement* de ρ' (noté $\rho \sqsubseteq \rho'$) si et

seulement si :

$$\forall X \in \rho \,.\, \exists X' \in \rho' \,.\, X \subseteq X'$$

■

Remarque 1-4
Si ρ est un raffinement de ρ', alors la relation d'équivalence associée à ρ est plus forte que la relation d'équivalence associée à ρ' :

$$\rho \sqsubseteq \rho' \;-\; \sim_\rho \subseteq \sim'_\rho$$

■

L'ensemble des partitions sur 2^{2^Q}, muni de la relation d'ordre $\sqsubseteq$, forme également un treillis complet dont les extremums sont :

- Le *maximum* est la *partition universelle* $\rho = \{Q\}$.
- Le *minimum* est la *partition identité* $\rho = \{\{p\} \mid p \in Q\}$.

On note alors respectivement $\sqcap$ et $\sqcup$ les bornes inférieure et supérieure des opérateurs définis de 2^{2^Q} vers 2^{2^Q} :

Définition 1.3-5
Soient $\rho = \{X_i \mid i \in \mathcal{I}\}$ et $\rho' = \{X'_i \mid i \in \mathcal{I}'\}$ deux partitions définies sur Q :

- La partition $\rho \sqcap \rho'$ est définie comme l'intersection deux à deux des classes de ρ et de ρ' :
$$\rho \sqcap \rho' = \{X \cap X' \mid X \in \rho \;\wedge\; X' \in \rho'\}$$
- La partition $\rho \sqcup \rho'$ est alors définie comme la plus petite partition qui soit un majorant de ρ et ρ' (vis-à-vis de l'opérateur $\sqsubseteq$) :
$$\rho \sqcup \rho' = \prod_i \rho_i \,.\, \rho \sqsubseteq \rho_i \;\wedge\; \rho' \sqsubseteq \rho_i$$

■

Remarque 1-5
Il est facile de vérifier que :

- $\rho \sqcap \rho'$ est la plus grande partition (vis-à-vis de l'opérateur de raffinement $\sqsubseteq$) satisfaisant :
$$\forall (p,q) \in Q^2 \,.\, p \sim_{\rho \sqcap \rho'} q \;\Rightarrow\; p \sim_\rho q \;\wedge\; p \sim_{\rho'} q$$
- De façon duale, $\rho \sqcup \rho'$ est la plus petite partition qui vérifie :
$$\forall (p,q) \in Q^2 \,.\, p \sim_\rho q \;\wedge\; p \sim_{\rho'} q \;\Rightarrow\; p \sim_{\rho \sqcup \rho'} q$$

■

Enfin, pour toute fonction totale, croissante, $\mathcal{G}$ de 2^{2^Q} vers 2^{2^Q} on note :

- $\mu\pi \,.\, \mathcal{G}(\pi)$ le plus petit point fixe de $\mathcal{G}$ par rapport à $\subseteq$
- $\nu\pi \,.\, \mathcal{G}(\pi)$ le plus grand point fixe de $\mathcal{G}$ par rapport à $\subseteq$

Chapitre 2

Relations de simulation et de bisimulation

Nous présentons dans ce chapitre une famille de relations d'équivalence et de préordre entre systèmes de transitions étiquetées. Ces relations sont utilisées dans le cadre de la vérification des systèmes parallèles, pour lequel elles offrent un intérêt particulier. Afin de pouvoir motiver leur définition, nous rappelons en premier lieu quels sont leurs rôles dans le processus de vérification et, par suite, quelles sont les principales conditions qu'elles doivent satisfaire en pratique :

- Dans le cas de spécifications comportementales, une relation d'équivalence ou de préordre permet d'établir un ensemble de *critères de comparaison* pour décider si le système de transitions étiquetées modélisant le comportement d'un programme parallèle est "égal" ou non au système de transitions étiquetées représentant sa spécification : la notion d'"égalité" que l'on considère alors détermine la *sémantique* que l'on donne à ce programme. Par conséquent, dans cette approche, le choix d'une "bonne" relation d'équivalence dépend de la nature des propriétés que l'on souhaite vérifier : d'une part cette relation doit être suffisamment forte pour n'identifier que des systèmes satisfaisant les mêmes propriétés, mais, elle doit aussi conduire à une description intuitive du comportement attendu en permettant d'effectuer des abstractions sur les aspects du programme qui ne sont pas prises en compte par la vérification.

- Dans le cas de spécifications logiques, les relations d'équivalence sont utilisées de manière auxiliaire, dans un but purement pratique : afin de diminuer le coût de la vérification, au lieu d'évaluer les formules de la spécification sur le système de transitions étiquetées qui modélise le programme à vérifier, il est intéressant de pouvoir les évaluer sur le quotient de ce système modulo une relation d'équivalence convenablement choisie. Cependant, une telle approche n'est est possible que si la relation d'équivalence considérée préserve les formules de la spécification (i.e, chaque formule valide sur un système de transitions étiquetées l'est aussi sur son quotient modulo cette relation).

Enfin, rappelons que, dans la pratique, une relation d'équivalence ne sera réellement utile que s'il lui est associé un algorithme efficace de comparaison ou de minimisation permettant de traiter des exemples réels.

Pour répondre à ces divers besoins, un nombre important de relations ont été proposées dans la littérature, différant tant par leur sémantique que par le formalisme (ou *domaine*) sur lequel elles sont définies. Concernant les relations entre systèmes de transitions étiquetées, on peut trouver dans [Gla90] une présentation des différentes sémantiques existantes, ainsi qu'un certain nombre de critères

de classification. L'un de ces critères, qui permet de comparer entre elles la plupart des familles de relations proposées à ce jour, est basé sur la distinction que l'on fait usuellement entre le *temps linéaire* et le *temps arborescent* pour la description des systèmes parallèles : dans le premier cas, on s'intéresse uniquement à l'ensemble des *séquences d'exécution* du système (et par suite au *langage d'actions* défini par cet ensemble), alors que dans le second cas on prend également en compte la *structure de branchement* de ces séquences, ce qui revient à considérer des *arbres d'exécution*. Lorsque l'on compare les différentes sémantiques en fonction de leur capacité à distinguer deux systèmes du point de vue de la structure de branchement, on obtient un treillis complet [Gla90], ordonné par la relation d'inclusion, et qui est tel que :

- Son plus grand élément, celui qui distingue le moins de systèmes, est la sémantique correspondant à l'*équivalence de trace* [Hoa78], définie comme la relation d'égalité sur les ensembles de séquences d'exécution. Il correspond à l'égalité classique des langages pour la théorie des automates.

- Son plus petit élément, celui qui distingue le plus de systèmes, est la sémantique correspondant aux *équivalences de bisimulation* [Par81], qui peuvent être vues comme des relations d'égalité entre les structures de branchement des arbres d'exécution des systèmes.

Les autres relations se situe alors entre ces deux extremums. On peut citer par exemple l'*équivalence par modèles de refus* [BHR84], l'*équivalence par modèles d'acceptation* [BKO88] [GS86] et l'*équivalence de test* [NH84].

Parmi les différentes familles de relations qui ont été proposées dans le cadre de la vérification, les relations de bisimulations (ou *relations comportementales*) occupent une place prépondérante, et ce pour plusieurs raisons :

- A partir de la définition générale de bisimulation, il est possible de construire un certain nombre de relations d'équivalence ou de préordres, éventuellement dédiées à un type d'utilisation particulier, comme la vérification d'une certaine classe de propriétés, ou la préservation d'un sous-ensemble des formules d'une logique temporelle donnée.

- La sémantique de ces relations, qui repose sur la notion de *comportement global* du système est assez intuitive, ce qui facilite l'écriture de spécifications de types comportementales.

- Enfin, et c'est aussi une des raisons essentielles du succès de ces relations, il existe un algorithme [PT87] efficace de comparaison et de minimisation des systèmes de transitions étiquetées modulo une bisimulation particulière, la bisimulation forte. Cet algorithme a pu ensuite être étendue à d'autres relations de bisimulation [KS83, Sor87] mais aussi à des relations basées sur des sémantiques plus faibles comme l'équivalence de test [CH89], ou l'équivalence par modèle d'acceptation [Fer89].

Par suite, l'intérêt suscité par les relations de bisimulations a été à l'origine d'un nombre important d'outils de vérification, qui ont entre autres permis de mieux cerner les limitations actuelles posées par ce type d'approche, et de découvrir quelles sont les relations pour lesquelles les algorithmes se comportent le mieux en pratique.

Dans la suite du chapitre, nous rappelons tout d'abord la définition générale d'équivalence de bisimulation, puis nous présentons plus en détails les relations qui nous semblent prometteuses du point de vue de la vérification, et auxquelles nous nous sommes plus particulièrement intéressé dans ce travail. Nous terminons alors par une comparaison de ces différentes équivalences en les situant dans le treillis des partitions.

2.1 Bisimulation modulo un critère d'abstraction

Nous proposons tout d'abord une définition intuitive de la sémantique de bisimulation, basée sur la notion de *comportement observable* des programmes. Nous en donnons ensuite une définition plus formelle en termes de relation d'équivalence entre états des systèmes de transitions étiquetées.

2.1.1 Comportements observables

Dans une sémantique de bisimulation, on considère que l'*état* d'un programme à un instant donné est entièrement déterminé, d'une part, par l'*ensemble* des actions qui peuvent être effectuées à cet instant, et, d'autre part, par les états qui pourront être atteints après exécution de ces actions.

On dit alors que deux programmes se trouvent dans des états identiques si et seulement si leurs comportements possibles à partir de ces états sont les mêmes : à toute évolution de l'un d'eux par une séquence d'actions donnée correspond une évolution de l'autre par la même séquence d'actions, telle qu'à chaque instant les comportements offerts par les états atteints lors de l'exécution de ces séquences soient à leur tour identiques.

La *relation de bisimulation* permet de formaliser cette notion d'*égalité* de comportements en l'exprimant au niveau des modèles : deux systèmes de transitions étiquetées se bisimulent lorsqu'ils représentent des programmes qui offrent des comportements identiques. Plus précisément, la bisimulation est définie de manière récursive comme une relation d'équivalence sur l'ensemble des états des deux systèmes de transitions étiquetées :

> Deux états p et q se bisimulent si et seulement si pour tout successeur p' de p par une action a donnée, il existe un successeur q' de q par la même action a tel que p' et q' se bisimulent, et réciproquement.

De façon similaire, il existe une seconde relation, la *relation de simulation*, qui modélise la notion d'*inclusion* entre comportements : intuitivement un système simule un autre système si et seulement si celui-ci présente *au moins* le même comportement. La relation de simulation est définie à partir de la bisimulation en considérant non plus une relation d'équivalence mais la *relation de préordre* qui lui est associée.

Ces deux relations sont rarement utilisées telles quelles dans le cadre de la vérification, car elles ne permettent pas de rendre compte de la différence d'abstraction qui existe entre le comportement d'un programme et celui sa spécification comportementale. Il est donc nécessaire de définir des relations plus faibles, qui identifieront davantage de comportements.

Une solution possible consiste à adopter un point de vue plus relatif sur le concept d'action d'un système en établissant un *critère d'abstraction* [Bou85a]. On considère alors non plus les actions "concrètes" du système (i.e, les actions atomiques qu'il peut effectuer) mais ses actions abstraites, ou actions *observables*, constituées d'un ensemble de séquences d'actions concrètes. Plus précisément, un critère d'abstraction peut donc être vu comme la donnée d'une *partition partielle* de l'ensemble des séquences d'actions concrètes du système (deux séquences d'actions qui appartiennent à la même classe correspondent à la même action abstraite), ou, de façon équivalente, par la donnée d'un *langage d'actions observables* construit sur cet ensemble.

Nous adopterons donc une définition plus générale pour la bisimulation et la simulation en les considérant comme des familles de relations paramétrées par un langage d'actions observables. En instanciant ces relations par un langage donné, on détermine alors une relation d'équivalence (*resp.* de préordre) qui permet de comparer deux systèmes modulo certaines abstractions de leur comportement (la façon dont seront effectué ces abstractions étant fixée par le choix du langage d'actions

observables).

Nous donnons dans la section suivante une définition plus formelle de ces deux relations.

2.1.2 Relations de bisimulation et de simulation

Soient $S_i = (Q_i, A_i, T_i, q_{0i})_{(i=1,2)}$ deux systèmes de transitions étiquetées, soit Λ un ensemble de langages disjoints sur $\mathcal{A} \cup \{\tau\}$, et soit λ un élément de Λ ($\lambda \subseteq (\mathcal{A} \cup \{\tau\})^*$).

La relation de bisimulation est définie comme une famille de relations binaires sur $Q_1 \times Q_2$ paramétrée par Λ :

Définition 2.1-1
Soit l'opérateur $\mathcal{B}_\Lambda : 2^{Q_1 \times Q_2} \longrightarrow 2^{Q_1 \times Q_2}$ défini par :

$$\begin{aligned}\mathcal{B}_\Lambda(R) \quad = \quad \{(p_1,p_2) \mid \forall \lambda \in \Lambda \,.\, \forall q_1 \,.\, (p_1 \xrightarrow{\lambda}_{T_1} q_1 \;\Rightarrow\; \exists q_2 \,.\, (p_2 \xrightarrow{\lambda}_{T_2} q_2 \;\wedge\; (q_1,q_2) \in R)) \\ \forall q_2 \,.\, (p_2 \xrightarrow{\lambda}_{T_2} q_2 \;\Rightarrow\; \exists q_1 \,.\, (p_1 \xrightarrow{\lambda}_{T_1} q_1 \;\wedge\; (q_1,q_2) \in R))\}\end{aligned}$$

Une relation R sur $Q_1 \times Q_2$ est une bisimulation si et seulement si $R \subseteq \mathcal{B}_\Lambda(R)$. ■

L'*équivalence de bisimulation* $\sim_\Lambda$ pour le langage Λ est alors défini comme le plus grand point fixe de l'opérateur $\mathcal{B}_\Lambda$ [Par81] :

$$\sim^\Lambda = \nu R.\,((Q_1 \times Q_2) \;\cap\; \mathcal{B}_\Lambda(R))$$

Il est souvent commode de définir de façon plus inductive la relation $\sim_\Lambda$ comme l'intersection d'une suite décroissante de relations $\sim^k_{\Lambda(k \in \mathcal{N})}$ [Mil80] :

- $\sim^0_\Lambda = Q_1 \times Q_2$
- $\forall k \geq 0,\ \sim^{k+1}_\Lambda = \mathcal{B}_\Lambda(\sim^k_\Lambda)$

On a alors,

$$\sim_\Lambda = \bigcap_{i=0}^{\infty} (\sim^i_\Lambda)$$

De façon similaire, la relation de simulation est obtenue à partir de l'équivalence de bisimulation en considérant le préordre associé. Elle est donc également définie comme une famille de relations sur $Q_1 \times Q_2$ paramétrée par Λ :

Définition 2.1-2
Soit l'opérateur $\mathcal{I}_\Lambda : 2^{Q_1 \times Q_2} \longrightarrow 2^{Q_1 \times Q_2}$ défini par :

$$\mathcal{I}_\Lambda(R) = \{(p_1,p_2) \mid \forall \lambda \in \Lambda \,.\, \forall q_1 \,.\, (p_1 \xrightarrow{\lambda}_{T_1} q_1 \;\Rightarrow\; \exists q_2 \,.\, (p_2 \xrightarrow{\lambda}_{T_2} q_2 \;\wedge\; (q_1,q_2) \in R))\}$$

Une relation R sur $Q_1 \times Q_2$ est une simulation si et seulement si $R \subseteq \mathcal{I}_\Lambda(R)$. ■

On appelle alors *préordre de simulation* $\sqsubseteq_\Lambda$ pour le langage Λ le plus grand point fixe de l'opérateur $\mathcal{I}_\Lambda$:

$$\sqsubseteq_\Lambda = \nu R.\,((Q_1 \times Q_2) \;\cap\; \mathcal{I}_\Lambda(R))$$

Comme dans le cas de l'équivalence de bisimulation, la relation $\sqsubseteq_\Lambda$ peut être définie comme l'intersection d'une suite décroissante de relations $\sqsubseteq^k_{\Lambda(k \in \mathcal{N})}$:

- $\sqsubseteq^0_\Lambda = Q_1 \times Q_2$
- $\forall k \geq 0, \sqsubseteq^{k+1}_\Lambda = \mathcal{I}_\Lambda(\sqsubseteq^k_\Lambda)$

On a alors,

$$\sqsubseteq_\Lambda = \bigcap_{i=0}^{\infty} (\sqsubseteq_\Lambda^i)$$

Enfin, à partir du préordre de simulation, il est possible de définir une troisième famille de relations en considérant la relation d'équivalence engendrée par ce préordre : on appelle *équivalence de simulation* $\approx_\Lambda$ l'intersection de la relation $\sqsubseteq_\Lambda$ et de sa relation inverse :

$$x \approx_\Lambda y \ - \ x \sqsubseteq_\Lambda y \ \wedge \ y \sqsubseteq_\Lambda x$$

Nous montrons dans la section 2.6 que l'équivalence obtenue est bien différente de l'équivalence de bisimulation.

2.2 La bisimulation forte

La relation de bisimulation forte, notée $\sim$ et souvent désignée sous le terme d'*équivalence forte*, est l'équivalence de bisimulation obtenue lorsque l'on considère comme observable chacune des actions concrètes du système. Elle correspond donc à l'ensemble de langages Λ suivant :

$$\Lambda = \{\{a\} \mid a \in \mathcal{A}_\tau\}$$

Par suite, du fait qu'aucune abstraction n'est effectuée lors de la comparaison, seuls des systèmes offrant *exactement* les mêmes comportements pourront être identifiés par la bisimulation forte. Bien que cette relation ne soit donc pas adaptée à la vérification de spécifications comportementales, elle occupe néanmoins une place importante tant sur le plan théorique que sur le plan pratique :

- On peut la considérer comme la relation d'"égalité" pour les systèmes de transitions étiquetées. Par suite, un système donné est généralement défini modulo la bisimulation forte. Ceci est de plus justifié par le fait que si deux systèmes se bisimulent fortement, alors ils sont identifiés par n'importe quelle autre relation intéressante du point de vue de la vérification.
- elle préserve l'ensemble des propriétés que l'on peut exprimer à l'aide des logiques temporelles utilisées à l'heure actuelle. Ainsi, toute formule des logiques HML [HM85], LTAC [QS83] ou CTL^* [CES83] valide sur un système de transitions étiquetées S l'est également sur toute la classe des systèmes de transitions étiquetées équivalent à S modulo la bisimulation forte, et en particulier sur le système quotient de S pour cette relation.
- Enfin, la bisimulation forte est également importante du point de vue algorithmique : il existe en effet un algorithme efficace de calcul du quotient d'un système de transitions étiquetées pour cette relation, sur lequel repose les méthodes de vérification utilisées en pratique pour la plupart des autres relations de bisimulation. Ce point sera développé plus en détail dans le chapitre 3.

2.3 L'équivalence observationnelle

L'un des critères d'abstraction les plus célèbres utilisé pour la vérification de spécifications comportementales est le *critère d'observation* proposé par Milner pour l'algèbre de processus CCS, et sur lequel est basée l'*équivalence observationnelle* (ou *équivalence faible*) [Mil80].

Ce critère repose sur une partition des actions du système en deux classes, les actions *visibles*, qui sont celles que l'on souhaite prendre en compte lors de la vérification, et les actions *internes* (ou invisibles), sur lesquelles portent effectivement l'abstraction, et qui seront renommées à l'aide de l'étiquette particulière τ. Le point de vue adopté par Milner est alors de considérer qu'on ne peut

pas distinguer par la seule observation un système effectuant une ou plusieurs actions internes d'un système n'effectuant aucune action. Par suite, une action observable sera définie comme étant une action visible, éventuellement précédée et suivie par un nombre quelconque d'actions internes.

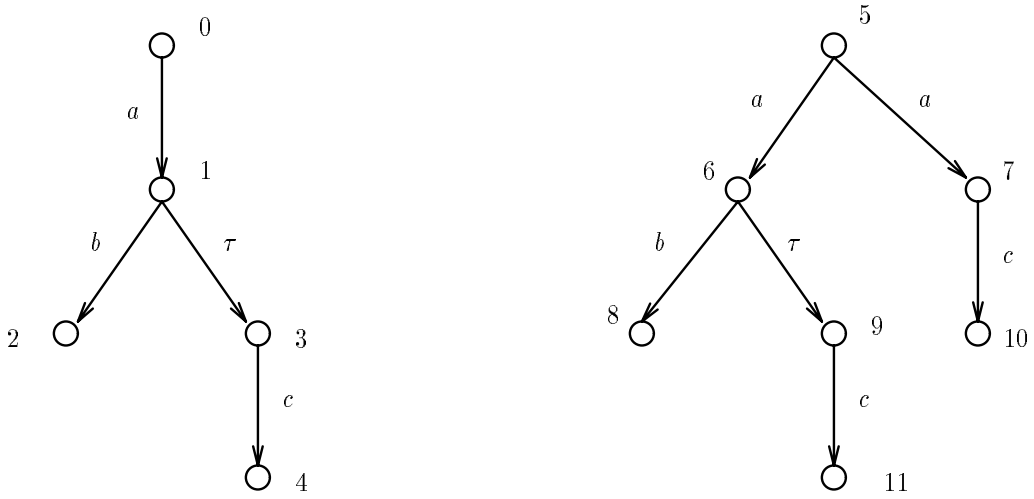
Plus formellement, l'équivalence observationnelle, notée $\approx_o$, est l'équivalence de bisimulation pour l'ensemble de langages Λ suivant :

$$\Lambda = \{\tau^*\} \cup \{\tau^* a \tau^* \mid a \in \mathcal{A}\}$$

Malgré cette définition attrayante, l'équivalence observationnelle souffre d'un certain nombre de désavantages du point de vue de la vérification :

- Sur le plan théorique, elle présente l'inconvénient de ne pas préserver la *structure de branchement* des systèmes [GW89], comme illustré par l'exemple 2-1 (ce point étant également repris dans la section 2.4). Contrairement à la bisimulation forte, l'équivalence observationnelle ne pourra donc pas être utilisée pour la vérification de spécifications exprimant des propriétés liées à la structure de branchement du système (comme certaines formules des logiques arborescentes).

  Exemple 2-1
  La figure suivante représente deux systèmes de transitions étiquetées S_1 et S_2 observationnellement équivalents qui diffèrent par leur structure de branchement :



  A partir de leur états initiaux (*resp.* 0 et 5), les deux systèmes peuvent exécuter l'action observable ($\tau^* a \tau^*$). Après exécution de cette action, il est possible d'atteindre :

  - soit des états à partir desquels l'action b peut être exécutée (*resp.* 1 pour S_1, 6 pour S_2),
  - soit des états à partir desquels l'exécution de b est impossible (*resp.* 3 pour S_1, 9 et 7 pour S_2).

  La différence entre ces deux systèmes du point de vue de la structure de branchement réside alors dans "le moment" à partir duquel le système "choisit" de façon non déterministe entre ces deux types de successeurs par l'action ($\tau^* a \tau^*$) : pour S_1, ce choix est toujours effectué après exécution de l'action a alors que pour S_2, il est également possible qu'il ait lieu avant son exécution. ■

- Sur le plan pratique, l'utilisation de l'équivalence observationnelle pour la vérification de spécifications comportementales est également limité en raison du coût en temps souvent prohibitif de la procédure de décision associée à cette relation. En effet, l'algorithme usuel (décrit dans

le chapitre 3) consiste à effectuer un certain nombre de transformations sur les systèmes de transitions étiquetées à comparer, puis à décider si les systèmes obtenus après cette première phase sont équivalents modulo la relation de bisimulation forte. Toutefois, les résultats obtenus à partir des outils "classiques" existants montrent que ces transformations, basées sur des calculs de fermeture transitive des relations de transition, s'avèrent extrêmement coûteuses dès que l'on compare des systèmes de taille moyenne [GV90, Qin91]. En conséquence, il semble que si cet algorithme est bien adapté à la vérification de systèmes de petite taille (quelques milliers d'états), il ne convient pas pour traiter des exemples plus réalistes.

Un certain nombre de travaux ont donc été menés ces dernières années, soit pour tenter d'améliorer la procédure de décision de l'équivalence observationnelle, soit pour proposer des alternatives à cette relation. Deux types d'approches nous semblent particulièrement prometteuses :

- La première approche consiste à définir une relation de bisimulation plus forte que l'équivalence observationnelle, la *bisimulation de branchement* [GW89], qui préserve la structure de branchement des systèmes. Associée à une procédure de décision efficace, et implémentée dans plusieurs outils, cette relation tient désormais une place importante du point de vue la vérification.

- Dans la seconde approche, on s'intéresse à une large classe de propriétés que l'on est amené à vérifier dans la pratique, qui sont les *propriétés de sûreté* des systèmes. Une relation d'équivalence permettant de caractériser exactement cette classe dans une sémantique de simulation a donc été définie [Rod88], pour laquelle il existe une procédure de décision plus efficace que celle de l'équivalence observationnelle.

Ces deux approches sont présentées dans la suite de façon plus détaillée à travers les relations d'équivalences qui leurs sont associées.

2.4 Les relations préservant la structure de branchement

2.4.1 Définition générale

La *bisimulation de branchement* (ou *branching bisimulation*) a été proposée par van Glabbeek et Weijland à partir du constat suivant :

> L'équivalence observationnelle ne préserve pas la structure de branchement des systèmes car elle ne respecte pas la notion intuitive d'*égalité des comportements observables* sur laquelle repose la relation de bisimulation.

En effet, il a été vu dans la section 2.1.1 que deux systèmes présentent des comportements observables identiques si et seulement si à chaque évolution de l'un des systèmes par une séquence d'actions observables correspond une évolution de l'autre système par la même séquence d'actions telle qu'*à chaque instant* les possibilités d'évolution des deux systèmes par une action donnée soient les mêmes. Or, en reprenant l'exemple 2-1 il apparaît que cette condition n'est pas respectée dans le cas de l'équivalence observationnelle :

A partir de son état initial, le système S_2 peut effectuer la séquence d'actions observables :

$5 \xrightarrow{a} 7 \xrightarrow{c} 10.$

A cette évolution correspond une seule évolution de S_1 par la même séquence d'actions observables :

$0 \xrightarrow{a} 1 \xrightarrow{\tau} 3 \xrightarrow{c} 4.$

Clairement, les possibilités d'évolution des systèmes S_1 et S_2 ne sont pas à chaque instant identiques lors de l'exécution de ces deux séquences : S_1 peut interrompre l'exécution de la première action

observable en exécutant une action b (à partir de l'état 1), ce que ne peut faire S_2.

L'idée de van Glabbeek et Weijland a donc été de proposer une nouvelle relation, basée sur ce même critère d'abstraction, mais qui respectant la notion d'égalité des comportements des systèmes. Intuitivement, cette relation est obtenue en renforçant la définition de l'équivalence observationnelle par une contrainte supplémentaire sur les états atteints *pendant* l'exécution d'une action observable :

Deux états p et q appartenant à des systèmes S_1 et S_2 sont équivalents si et seulement si :

- A chaque successeur p' de p par une action observable $\tau^* a \tau^*$ correspond un successeur q' de q par la même action observable tel que d'une part p' et q' soient équivalents.
- A chaque "état intermédiaire" p'' de S_1 atteint pendant l'exécution de l'action $\tau^* a \tau^*$, corresponde un état q'' de S_2 également atteint lors de l'exécution de cette action qui lui soit équivalent.

A partir de cette définition intuitive, il est en fait possible de construire un certain nombre de relations qui vont différer uniquement dans la façon dont sont "choisis" les états intermédiaires qui doivent rester équivalents lors de l'exécution d'une action observable. Plus formellement, nous considérons une relation générale $\mathcal{R}_C$ paramétrée par une fonction booléenne C exprimant une condition sur les états atteints lors de l'exécution d'une action observable. Au choix d'une fonction C correspondra alors une relation donnée.

Définition 2.4-1
Soit C une fonction booléenne. $\mathcal{R}_C$ est la plus grande relation $\mathcal{R}$ *symétrique* sur $Q_1 \times Q_2$ satisfaisant :

$p \mathcal{R} q \ - \ \forall a \in \mathcal{A} \cup \{\tau\} \, . \, \forall p' \, . \, p \xrightarrow{a} p' \ \Rightarrow$

(i) $a = \tau$ et $p' \mathcal{R} q$,

ou

(ii) $\exists \, q_1, q', q_2 \, . \, q \xrightarrow{\tau^*} q_1 \xrightarrow{a} q' \xrightarrow{\tau^*} q_2$ et $C(\mathcal{R}, p, q_1, q', q_2)$.

■

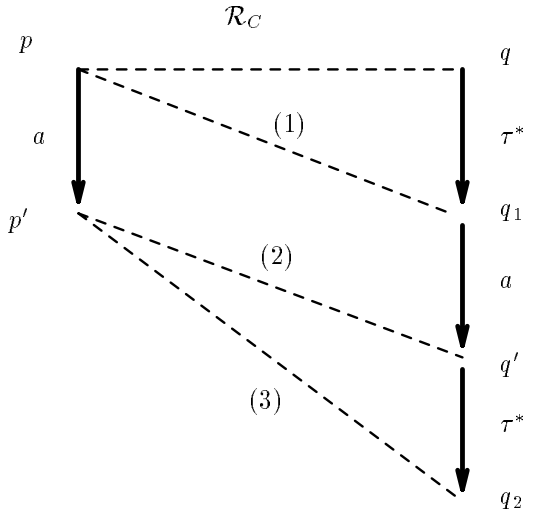
Nous considérerons ici des fonctions $C(\mathcal{R}, p, q_1, q', q_2)$ obtenues à partir de conjonctions des prédicats binaires suivants :

(1) $p \mathcal{R} q_1$

(2) $p' \mathcal{R} q'$

(3) $p' \mathcal{R} q_2$

Par suite, le point (ii) de la définition de $\mathcal{R}_C$ peut être schématisé par la figure suivante :



Plus précisément, quatre fonctions seront envisagées pour C, donnant lieu à quatre relations d'équivalences :

- La relation la plus faible est obtenue lorsque l'on ne fait aucune hypothèse sur les états intermédiaires, ce qui correspond à la fonction

  $C_{obs} = (p' \mathcal{R} q_2)$.

  Comme on pouvait s'y attendre, cette relation n'est autre que l'équivalence observationnelle.

- La relation la plus forte, qui est la bisimulation de branchement de van Glabbeeck et Weijland, est obtenue en exigeant que les états intermédiaires q_1, q' et q_2 soient respectivement équivalents à q_1 et à q', ce qui correspond à la fonction :

  $C_{br} = (p \mathcal{R} q_1) \wedge (p' \mathcal{R} q') \wedge (p' \mathcal{R} q_2)$.

Les deux autres relations occupent des positions intermédiaires entre l'équivalence observationnelle et la bisimulation de branchement :

- La *delay bisimulation* [NMV90] est obtenue en considérant la fonction $C_d = (p' \mathcal{R} q')$.
- La *η-bisimulation* [BvG87] est obtenue en considérant la fonction $C_\eta = (p \mathcal{R} q_1) \wedge (p' \mathcal{R} q_2)$.

Il peut paraître intéressant de renforcer davantage la définition de la bisimulation de branchement en exigeant que *tous* les états intermédiaires qui sont atteints entre les occurrences de deux actions internes lors de l'exécution des séquences $q \xrightarrow{\tau^*} q_1$ et $q' \xrightarrow{\tau^*} q_2$ restent équivalents à p et p' respectivement. Toutefois, le lemme suivant ([NMV90]) montre que la relation d'équivalence ainsi obtenue coïncide avec la bisimulation de branchement telle que nous l'avons défini ici :

Lemme 2.4-1
Soit $S = (Q, A, T, q_0)$ un système de transitions étiquetées et soit $p_0, p_1, \ldots, p_n$ des états de Q ($n > 0$) tels que $p \xrightarrow{\tau}_{T_1} p_1 \xrightarrow{\tau}_{T_1} p_2 \cdots \xrightarrow{\tau}_{T_1} p_n \xrightarrow{\tau}_{T_1} p'$. En notant $\mathcal{R}_{br}$, la relation $\mathcal{R}_C$ (*cf.* définition 2.4-1) obtenue pour $C = C_{br}$, on a :

$p \mathcal{R}_{br} p_n \; - \; \forall i, 0 \leq i \leq n \; . \; p \mathcal{R}_{br} p_i$

■

Remarque 2-1
De façon similaire, il est facile de voir que la bisimulation de branchement et la delay bisimulation coïncident également avec les relations d'équivalence obtenues lorsque l'on considère respectivement

les fonctions booléennes :

$$C'_{br} = (p\ \mathcal{R}\ q_1)\ \wedge\ (p'\ \mathcal{R}\ q')$$
$$C'_{d} = (p'\ \mathcal{R}\ q')\ \wedge\ (p'\ \mathcal{R}\ q_2)$$

■

Dans la suite, nous nous intéressons plus particulièrement à la bisimulation de branchement et la delay bisimulation.

2.4.2 La bisimulation de branchement

En adoptant un formalisme similaire à celui utilisé pour les relations de simulation et de bisimulation, la bisimulation de branchement peut être définie de la façon suivante :

Définition 2.4-2

Soit $\mathcal{B}^{br} : 2^{Q_1 \times Q_2} \longrightarrow 2^{Q_1 \times Q_2}$ défini par :

$$\begin{aligned}\mathcal{B}^{br}(R) = \{(p_1,p_2) \mid \forall a \in \mathcal{A} \cup \{\tau\}\ .\ & \\ \forall q_1\ .\ (p_1 \xrightarrow{a}_{T_1} q_1\ \Rightarrow\ & (a = \tau\ \wedge\ (q_1,p_2) \in R)\ \vee \\ & (\exists q_2 q'_2\ .\ (p_2 \xrightarrow{\tau^*}_{T_2} q'_2\ \wedge\ q'_2 \xrightarrow{a}_{T_2} q_2\ \wedge\ (p_1,q'_2) \in R\ \wedge\ (q_1,q_2) \in R))) \\ \forall q_2\ .\ (p_2 \xrightarrow{a}_{T_2} q_2\ \Rightarrow\ & (a = \tau\ \wedge\ (p_1,q_2) \in R)\ \vee \\ & (\exists q_1 q'_1\ .\ (p_1 \xrightarrow{\tau^*}_{T_1} q'_1\ \wedge\ q'_1 \xrightarrow{a}_{T_1} q_1\ \wedge\ (q'_1,p_2) \in R\ \wedge\ (q_1,q_2) \in R)))\}\end{aligned}$$

Une relation R sur $Q_1 \times Q_2$. R est une bisimulation de branchement si et seulement si $R \subseteq \mathcal{B}^{br}(R)$.

■

Dans la suite, nous appellerons *bisimulation de branchement*, notée $\sim_{br}$ le plus grand point fixe de l'opérateur $\mathcal{B}^{br}$:

$$\sim_{br} = \nu R.\ ((Q_1 \times Q_2)\ \cap\ \mathcal{B}^{br}(R))$$

Comme les relations de bisimulation et de simulation, la relation $\sim_{br}$ peut également être construite comme intersection d'une suite décroissante de relations ${\sim_{br}}^{k}{}_{(k \in \mathcal{N})}$:

- ${\sim_{br}}^{0} = Q_1 \times Q_2$
- $\forall k \geq 0, {\sim_{br}}^{k+1} = \mathcal{B}^{br}({\sim_{br}}^{k})$

On a alors :

$$\sim_{br} = \bigcap_{i=0}^{\infty} ({\sim_{br}}^{i})$$

Sur le plan de la vérification, la bisimulation de branchement permet de remédier à la plupart des inconvénients liés à l'utilisation de l'équivalence observationnelle :

- Préservant la structure de branchement des systèmes, elle peut être utilisée pour la vérification de spécifications exprimées à l'aide des logiques arborescentes. Cette relation est par exemple *adéquate* pour un sous-ensemble de la logique CTL [NV90] : toute formule de ce sous-ensemble, valide sur un système de transitions étiquetées donné, l'est également sur son quotient modulo la bisimulation de branchement.

- De plus, il existe un algorithme efficace pour comparer (*resp.* minimiser) des systèmes de transitions étiquetés modulo la bisimulation de branchement (*cf.* chapitre 3). Cet algorithme, qui ne nécessite pas le calcul préalable de fermetures transitives sur les relations de transitions permet de traiter des systèmes de taille moyenne (i.e, quelques centaines de milliers d'états).

- Enfin, il n'existe pas à notre connaissance d'exemple "réel" de système qui ait été vérifié à l'aide de l'équivalence observationnelle et qui n'aurait pu l'être en utilisant la bisimulation de branchement.

2.4.3 La delay bisimulation

Dans la section 2.4.1, la delay bisimulation a été obtenue à partir de la relation $\mathcal{R}_C$ en considérant une fonction booléenne C particulière. Plus précisément, cette relation (notée $\approx_d$) est définie de la manière suivante :

Définition 2.4-3

$$
\begin{aligned}
p \approx_d q \quad - \quad & ((\forall a \in A \cup \{\tau\} \,.\, \forall p' \,.\, p \xrightarrow{a}_{T_1} p' \;\Rightarrow \\
& \quad (a = \tau \;\wedge\; p' \approx_d q) \;\vee\; (\exists q_1, q' \,.\, q \xrightarrow{\tau^*}_{T2} q_1 \xrightarrow{a}_{T2} q' \;\wedge\; p' \approx_d q')))) \\
& \wedge \\
& ((\forall a \in A \cup \{\tau\} \,.\, \forall q' \,.\, q \xrightarrow{a}_{T_2} q' \;\Rightarrow \\
& \quad (a = \tau \;\wedge\; p \approx_d q') \;\vee\; (\exists p_1, p' \,.\, p \xrightarrow{\tau^*}_{T1} p_1 \xrightarrow{a}_{T1} p' \;\wedge\; p' \approx_d q'))))
\end{aligned}
$$

■

La proposition 2.4-1 permet de donner une autre définition de cette relation en terme d'équivalence de bisimulation.

Proposition 2.4-1

$$
\begin{aligned}
p \approx_d q \quad - \quad & (\forall a \in A \,.\, \forall p' \,.\, (p \xrightarrow{a}_{T_1} p' \;\Rightarrow\; \exists q' \,.\, (q \xrightarrow{\tau^* a}_{T2} q' \;\wedge\; p' \approx_d q'))) \\
& \quad \wedge (\forall p' \,.\, (p \xrightarrow{\tau}_{T_1} p' \;\Rightarrow\; \exists q' \,.\, (q \xrightarrow{\tau^*}_{T2} q' \;\wedge\; p' \approx_d q'))) \\
& \wedge \\
& (\forall a \in A \,.\, \forall q' \,.\, (q \xrightarrow{a}_{T_2} q' \;\Rightarrow\; \exists p' \,.\, (p \xrightarrow{\tau^* a}_{T1} p' \;\wedge\; p' \approx_d q'))) \\
& \quad \wedge (\forall q' \,.\, (q \xrightarrow{\tau}_{T_2} q' \;\Rightarrow\; \exists p' \,.\, (p \xrightarrow{\tau^*}_{T1} p' \;\wedge\; p' \approx_d q')))
\end{aligned}
$$

■

Preuve : Elle est immédiate du fait que l'implication :

$$
\begin{aligned}
& (\forall a \in A \cup \{\tau\} \,.\, \forall p' \,.\, p \xrightarrow{a}_{T_1} p' \;\Rightarrow \\
& \quad (a = \tau \;\wedge\; p' \approx_d q) \;\vee\; (\exists q_1, q' \,.\, q \xrightarrow{\tau^*}_{T2} q_1 \xrightarrow{a}_{T2} q' \;\wedge\; p' \approx_d q'))
\end{aligned}
$$

est logiquement équivalente à :

$$
\begin{aligned}
& (\forall a \in A \,.\, \forall p' \,.\, (p \xrightarrow{a}_{T_1} p' \;\Rightarrow\; \exists q' \,.\, (q \xrightarrow{\tau^* a}_{T2} q' \;\wedge\; p' \approx_d q'))) \\
& \quad \wedge \quad (\forall p' \,.\, (p \xrightarrow{\tau}_{T_1} p' \;\Rightarrow\; \exists q' \,.\, (q \xrightarrow{\tau^*}_{T2} q' \;\wedge\; p' \approx_d q')))
\end{aligned}
$$

□.

A partir de la proposition 2.4-1, il est facile de voir que la relation $\approx_d$ est l'équivalence de bisimulation obtenue en considérant l'ensemble de langage :

$$\Lambda = \{\tau^*\} \;\cup\; \{\tau^* a \mid a \in \mathcal{A}\}$$

Du point de vue de la vérification, l'intérêt de cette relation par rapport à la bisimulation de branchement n'est pas évident au premier abord :

- Elle ne préserve pas toutes les propriétés exprimables à l'aide des logiques temporelles arborescentes (par exemple seul un sous-ensemble de CTL est préservé), et d'autre part elle ne semble pas non plus mieux adaptée à la vérification de spécifications comportementales.
- La procédure de décision "classique" qui lui est associée, et qui est construite sur le même principe que celle de l'équivalence observationnelle, n'est pas plus efficace dans le cas général que celle de la bisimulation de branchement. Néanmoins, nous verrons au chapitre 5 que cette relation peut jouer un rôle lorsque l'on se place dans le cadre de la comparaison "à la volée".

2.5 Les relations préservant les propriétés de sûreté

2.5.1 Propriétés de sûreté

Parmi les différentes classes de propriétés considérées en pratique pour spécifier le comportement attendu d'un système parallèle, on distingue généralement entre *propriétés de sûreté* et *propriétés de vivacité*. Cette distinction, originalement proposée par Lamport [Lam77], est motivée intuitivement de la manière suivante :

- Une propriété de sûreté (ou *safety*) exprime le fait que toute action effectuée par le système est une action correcte, ou, de façon équivalente, que le système ne présente jamais un comportement non prévu par sa spécification.
- Une propriété de vivacité (ou *liveness*) exprime le fait toute action correcte sera *inévitablement* effectuée par le système.

Dans la pratique, il apparaît qu'un certain nombre de propriétés utiles peuvent être exprimées sous la forme de propriétés de sûreté, et que leur vérification est généralement plus facile à mettre en œuvre que celle des propriétés de vivacité.

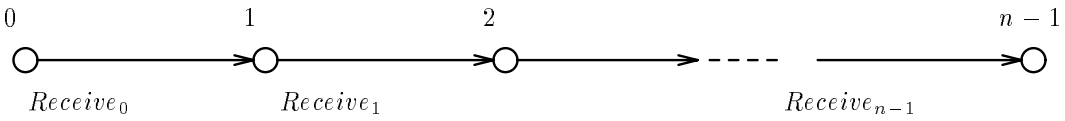
Les spécifications suivantes, empruntées au domaine des protocoles de communication, en présentent quelques exemples :

Exemple 2-2
L'une des propriétés souvent requise pour les protocoles est la *préservation de l'ordre* des messages transmis. Nous prenons ici l'exemple d'un protocole de diffusion atomique, le protocole *rel/REL* [SE90], qui permet le transfert de messages entre une station émettrice et un groupe de stations réceptrices, tout en tolérant les pannes potentielles de chacune des stations engagées dans la communication. Cet exemple est présenté plus en détails dans l'annexe B.

La spécification de ce protocole comprend entre-autres la propriété de sûreté P suivante :

> Toute séquence de messages adressée par un émetteur E ne pourra être reçue par chacun des récepteurs que dans l'ordre où elle a été émise.

Plus précisément, si l'on numérote chaque message émis par E à l'aide des entiers successifs 0, 1, ..., $n-1$, et si l'on considère comme seules actions visibles les réceptions de ces messages (notées $Receive_i$) par un récepteur R donné, on obtient la spécification comportementale représentée par le graphe $S_{(E,R)}$ de la figure suivante lorsque n messages sont émis de E vers R :



La spécification complète de la propriété P est alors représentée par l'ensemble des systèmes de transitions étiquetés $S_{(E,R)}$ obtenus pour chacun des couples émetteur-récepteur du réseau. ■

Exemple 2-3

Cet exemple, issu de [Rod88], décrit une partie du service d'un protocole de transport de la couche 4 du schéma de normalisation proposé par l'ISO. Ce protocole permet la communication entre deux entités, l'*initiateur* et l'*accepteur*, en faisant appel à des fonctions de la couche inférieure (couche *réseau*). Informellement, le fonctionnement du protocole peut se décomposer en trois phases :

- La première phase consiste à établir la connexion entre les deux entités. L'initiateur commence par émettre une *demande de connexion* ($ConReq$). Le réseau transmet alors une *indication de connexion* ($ConInd$) à l'accepteur, qui peut émettre à son tour une *réponse de connexion* ($ConRep$). Dans ce cas, une *confirmation de connexion* ($ConConf$) est retournée par le réseau à l'initiateur, et la connexion est établie.

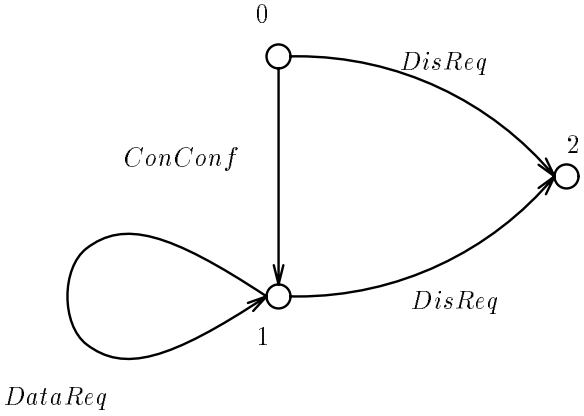
- Une fois la connexion établie, l'initiateur peut alors envoyer des messages à l'accepteur. Pour ce faire, il émet un certain nombre de *demandes de données* ($DataReq$), transmises à l'accepteur par le réseau sous forme d'*indications de données* ($DataInd$).

- A tout moment l'initiateur peut décider d'interrompre la communication. Il émet dans ce cas une *demande de déconnexion* ($DisReq$), transmise à l'accepteur par le réseau sous forme d'une *indication de déconnexion* ($DiscInd$). La connexion est alors terminée lorsque l'initiateur reçoit une *confirmation de déconnexion* ($DisConf$) de la part du réseau.

Un certain nombre de propriétés de sûreté décrivant le service de ce protocole sont présentées dans [Rod88] à l'aide de la logique temporelle LTAC [QS83]. Nous considérons ici l'une d'entre-elles, notée P, et nous l'exprimons de façon comportementale par un système de transitions étiquetées :

> Toute demande de données est précédée d'une confirmation de connexion, et il n'y a pas de demande de déconnexion entre les occurrences de ces deux actions.

En considérant comme observables les seules actions $DataReq$, $ConConf$ et $DisReq$, la propriété P est représentée par le système suivant :

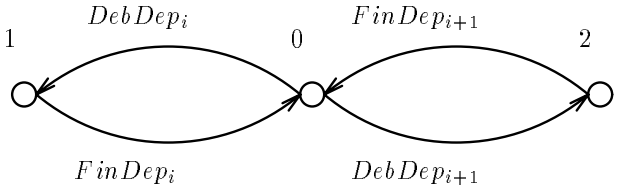


■

Exemple 2-4
Enfin, nous considérons dans cet exemple un protocole destiné à assurer la sécurité des dépassements dans une file de véhicules automobiles [EåFJ90]. De façon plus précise, la file de véhicules est modélisée par un ensemble de processus communicants, chacun d'eux étant repéré par un numéro d'ordre dans la file. L'une des propriétés critique que devra respecter toute implémentation de ce protocole est alors la propriété de sûreté P suivante :

> Une fois que le véhicule i a commencé une opération de dépassement, ni le véhicule $i+1$, ni le véhicule $i-1$ ne peuvent à leur tour entamer un dépassement avant que i n'ait terminé le sien.

Une autre formulation possible pour P est d'imposer que, pour tout couple de véhicules adjacents $(i, i+1)$, les opérations de dépassements soient toujours effectuées en exclusion mutuelle. En représentant par $DebDep_i$ (*resp.* $FinDep_i$) les actions de début (*resp.* de fin) de dépassement du véhicule i, on obtient alors la spécification comportementale S_i suivante :



La spécification complète de P est alors représentée par l'ensemble des graphes obtenus lorsque l'on considère chaque véhicule i de la file. ■

L'examen de ces exemples montre qu'il existe un certain nombre de cas que l'on rencontre souvent lorsque l'on spécifie le comportement attendu d'un système parallèle et qui correspondent à des énoncés de propriétés de sûreté. D'une manière générale, on a affaire à ce type de propriété dans les situations suivantes (cette liste n'étant pas exhaustive) :

- pour exprimer des conditions sur l'ordre dans lequel certaines actions peuvent être exécutées,
- pour spécifier que deux actions ne sont jamais exécutables à partir d'un même état,
- pour vérifier qu'une (ou plusieurs) actions données seront toujours exécutées entre les occurrences de deux autres actions (ce qui permet par exemple de vérifier que deux processus sont bien en exclusion mutuelle).
- et, dualement, pour vérifier qu'aucune action intervient entre les occurrences de deux autres actions.

Par contre, *l'absence de famine* ne peut pas être exprimée comme une propriété de sûreté.

Dans les spécifications précédentes, la sémantique que l'on donne aux graphes représentant les propriétés de sûreté est différente de celle que l'on donne habituellement aux systèmes de transitions étiquetées. Considérons par exemple le système de transitions étiquetées $S_{(E,R)}$ de l'exemple 2-2. L'existence de la transition $1 \stackrel{Receive_2}{\longrightarrow} 2$ est généralement interprétée comme la conjonction de deux prédicats [Rod88] :

> P_1. Si, à partir de l'état 1, on effectue l'action $Receive_2$, on atteint l'état 2. De plus, la seule action exécutable à partir de l'état 1 est l'action $Receive_2$.
>
> P_2. L'état 1 n'est pas un état puits : l'état 2 sera inévitablement atteint à partir de cet état.

Dans notre cas, seul le prédicat P_1 doit être pris en compte dans la sémantique donnée à $S_{(E,R)}$, la transition $1 \stackrel{Receive_2}{\longrightarrow} 2$ étant alors intuitivement interprétée de la façon suivante : si le récepteur vient

de recevoir le message 1, et *s'il reçoit un autre message*, alors ce message sera le message 2. De tels systèmes de transitions étiquetées, dont les transitions sont interprétées à l'aide du seul prédicat P_1, sont appelés *graphes de sûreté* [Rod88].

Afin de pouvoir vérifier si une propriété de sûreté est valide ou non sur un système de transitions étiquetées donné, il reste à définir une relation permettant de comparer ce système avec le graphe de sûreté qui décrit la propriété. Dans ce but, nous présentons dans la section suivante trois relations entre systèmes de transitions étiquetées.

2.5.2 Préordre de sûreté, équivalence de sûreté, et τ^*a-bisimulation

Formellement, le préordre de sûreté, noté $\sqsubseteq_s$, est le préordre de simulation $\sqsubseteq_\Lambda$ obtenu pour l'ensemble de langages Λ_{τ^*a} défini par :

$$\Lambda_{\tau^*a} = \{\tau^*a \mid a \in \mathcal{A}\}$$

Remarque 2-2
Cette relation coïncide en fait avec les différents préordres de simulation obtenus pour les ensembles de langages suivants :

$$\Lambda_d = \{\tau^*\} \cup \Lambda_{\tau^*a}$$
$$\Lambda_o = \{\tau^*\} \cup \{\tau^*a\tau^* \mid a \in \mathcal{A}_\tau\}$$

Dans le cas de Λ_o, cette propriété sera justifiée par la suite dans la proposition 2.6-6. Une preuve similaire pourrait être effectuée pour la famille de langages Λ_d. ■

L'équivalence de sûreté, notée $\approx_s$, est alors définie comme la relation d'équivalence associée à ce préordre, soit l'équivalence de simulation obtenue pour l'ensemble de langages Λ_{τ^*a} :

$$\approx_s = \sqsubseteq_s \cap \sqsubseteq_s^{-1} .$$

Le préordre de sûreté permet la comparaison entre un système de transitions étiquetée et un graphe de sûreté. En effet, si l'on considère deux systèmes de transitions étiquetées S_1 et S_2, tels que S_2 ne comporte que des actions visibles (i.e, non renommées en τ), $S_1 \sqsubseteq_s S_2$ si et seulement si S_1 satisfait la propriété de sûreté décrite par S_2 (alors interprété comme un graphe de sûreté).

L'équivalence de sûreté caractérise l'ensemble des propriétés de sûreté qui peuvent être exprimées dans une sémantique arborescente [BFG+91]. Il existe en effet une logique temporelle arborescente, la logique BSL (Branching-time Safety Logic) permettant de décrire exactement cette classe de propriétés, et qui est expressive et adéquate pour la relation $\approx_s$:

- Toute formule BSL représente une union de classes d'équivalences pour la relation $\approx_s$.
- A toute classe d'équivalence pour la relation $\approx_s$ peut être associée une formule BSL qui la caractérise.

Bien que les relations de préordre et d'équivalence de sûreté soient suffisantes sur le plan théorique pour vérifier les propriétés de sûreté des systèmes, nous proposons également une troisième relation, la τ^*a-*bisimulation*, dont le rôle est essentiellement pratique. La τ^*a-bisimulation est définie comme étant l'équivalence de bisimulation obtenue pour l'ensemble de langage Λ_{τ^*a}. Du point de vue de la vérification, l'intérêt de cette relation est double :

- Etant plus forte que l'équivalence de sûreté, cette relation préserve les propriétés de sûreté : si une formule BSL est valide sur un système de transitions étiquetées donné, elle l'est également sur son quotient pour la τ^*a-bisimulation.
- De plus, s'agissant d'une équivalence de bisimulation, nous verrons dans les chapitres suivants que la procédure de décision qui lui est associée est plus efficace que celle de l'équivalence

de sûreté, que ce soit en utilisant les algorithmes classiques, ou en appliquant les méthodes de comparaison ou minimisation "à la volée". En outre, il se trouve que pour un nombre important d'exemples rencontrés dans la pratique ces deux équivalences coïncident (*cf.* section 2.6).

2.5.3 Discussion

Nous avons présenté une relation d'équivalence $\approx_s$ qui permet de vérifier les propriétés de sûreté des systèmes. Nous complétons ici cette présentation en discutant brièvement des limitations de cette relation, mais aussi plus généralement des restrictions existantes sur l'utilisation des relations d'équivalences pour la vérification de propriétés de vivacité.

L'une des propriétés souvent requise lorsque l'on vérifie un système parallèle est l'*absence de blocages* (ou *deadlock freedom*). Intuitivement, vérifier qu'un système ne comporte pas de blocage consiste à s'assurer que dans chaque état accessible du système il est possible d'effectuer au moins une des actions observables prévue par la spécification. Par suite, il ne s'agit pas d'une propriété de sûreté, puisqu'un système ne pouvant effectuer aucune action présente nécessairement un comportement sûr vis-à-vis de n'importe quelle spécification.

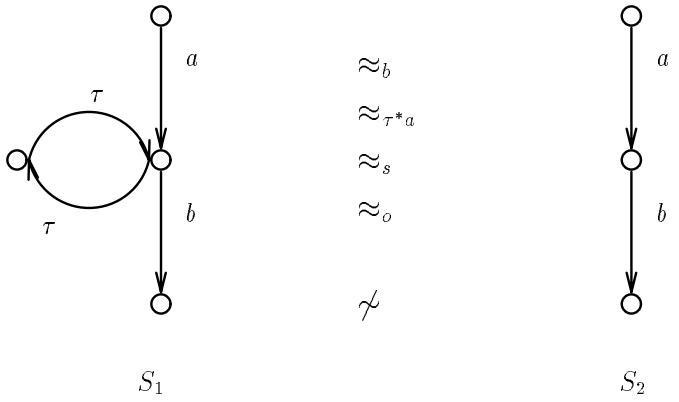
En conséquence, cette propriété n'est pas préservée par l'équivalence de sûreté (un système de transitions étiquetées comportant des blocages pouvant être équivalent à un autre système qui en est dépourvu) et elle ne pourra donc pas être vérifié à l'aide de la relation $\approx_s$. Toutefois, ce type de vérification, qui peut être menée à l'aide d'algorithmes classiques de parcours de graphes, ne pose pas de problèmes cruciaux en pratique.

Remarque 2-3
A contrario, l'absence de blocage peut être exprimée sous la forme d'une propriété de sûreté dans le cas des *logiques linéaires* du fait que l'on ne considère que les comportement infinis des systèmes [AS87]. ■

Un second type de propriétés que l'on peut être amené à rencontrer lorsque l'on vérifie un système parallèle est celui qui correspond aux propriétés de vivacité, exprimées sous la forme d'*inévitabilité*, qui permettent de spécifier qu'une action pourra toujours être exécutée, ou qu'un état pourra toujours être atteint. De telles propriétés sont par exemple requises pour s'assurer qu'un message est reçu, ou qu'un *timeout* expire au bout d'un temps fini. Néanmoins, l'utilisation de relations d'équivalence pour vérifier ce type de propriétés ne peut généralement se faire que si on prend en compte un certain nombre de conditions préliminaires :

- En premier lieu, l'exemple suivant montre que, parmi l'ensemble des relations que nous avons défini dans ce chapitre, seule la bisimulation forte préserve les propriétés d'inévitabilité :

  Exemple 2-5



On note P la propriété :

> l'action a est inévitablement suivie de l'action b

Intuitivement, S_1 ne vérifie pas la propriété P et S_2 la vérifie, bien que ces deux systèmes soit équivalents vis-à-vis de chacune des bisimulations ou simulations faibles définies dans ce chapitre. ■

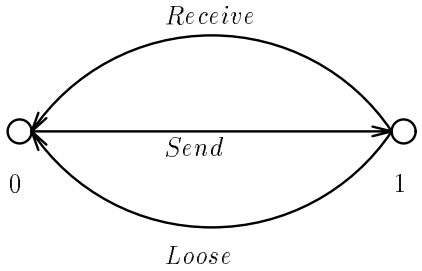
D'une façon générale, il est facile de voir que, dès lors qu'une relation d'équivalence identifie l'ensemble des états appartenant à une même composante fortement connexe de la relation de transition étiquetée par l'action τ, elle ne permet pas la vérification des propriétés d'inévitabilité. Une solution possible consiste alors à utiliser des relations d'équivalence plus fortes que celles que nous avons présentées, et qui identifient uniquement les états ayant les mêmes possibilités de *divergence*. Un exemple de telle relation construit à partir de la bisimulation de branchement, la *divergence sensible branching bisimulation* peut être trouvé dans [NV90].

- D'autre part, la vérification de propriétés d'inévitabilité nécessite que le formalisme utilisé pour décrire le système que l'on vérifie soit suffisamment puissant pour pouvoir exprimer des assertions sur son comportement infini. Plus précisément, il doit permettre d'inclure, par exemple, dans la description des composants du système des informations de la forme :

  > L'action a, dont l'exécution par le processus P est possible infiniment souvent, sera inévitablement exécutée par ce processus.

  Les langages de description basées sur les algèbres de processus classiques, dans lesquels le choix entre deux événements concurrents est modélisé une sémantique d'entrelacement, ne permettent pas d'exprimer ce type d'assertions. Par suite, comme illustré sur l'exemple suivant, il est généralement impossible de vérifier une propriété d'inévitabilité sur un système décrit à l'aide de ces langages.

Exemple 2-6
On considère un protocole de "bit alterné" très simplifié composé d'un émetteur, d'un récepteur et d'un médium de communication non fiable (i.e, susceptible de perdre une certaine partie des messages qui lui sont transmis). Modélisé à l'aide d'un langage de type algèbre de processus, le comportement de ce médium pourrait être représenté par le système de transitions étiquetées suivant, où les actions *Receive*, *Send* et *Loose* dénotent respectivement une réception d'un message venant de l'émetteur, une émission vers le récepteur ou une perte du message :



Il existe un comportement possible de ce système qui consiste à perdre systématiquement tout message qui lui est adressé. Un tel comportement, qui ne correspond pas à la description informelle du médium, invalide la propriété exprimant que tout message est inévitablement acheminé. ■

Dans le cas de l'exemple précédent, les solutions possibles consistent alors soit à spécifier exactement quels sont les messages qui sont perdus et quels sont ceux qui sont transmis correctement (et par suite la vérification ne devient valide que dans ce cas particulier), soit à utiliser un autre formalisme de description. Un exemple de tel formalisme, basé sur les automates de mots infinis, est proposé dans [?]. Toutefois, n'étant pas exprimable en termes de systèmes de transitions étiquetées, il requiert des méthodes de vérification plus puissantes que celles basées strictement sur les relations de bisimulation.

2.6 Comparaison des différentes relations

Le but de cette section est de comparer entre elles vis-à-vis de l'inclusion les différentes relations d'équivalence et de préordres qui ont été définies dans ce chapitre. Même si cette démarche est déjà rendue nécessaire sur un plan théorique, elle présente également beaucoup d'intérêts du point de vue pratique. En effet, lorsque l'on est amené à vérifier qu'un système de transitions étiquetées satisfait une spécification donnée (qu'elle soit de type logique ou comportementale), il peut s'avérer particulièrement utile de savoir que telle relation est préservée par telle autre, ou que deux relations coïncident dans certains cas particuliers. Ainsi, il devient possible de déterminer une *stratégie* pour la vérification, en fonction des procédures de décision offertes par les outils que l'on a à sa disposition.

Par suite, cette section sera divisée en deux parties : nous examinons dans un premier temps les positions relatives dans le treillis des partitions des relations présentées dans ce chapitre lorsque l'on considère des systèmes de transitions étiquetées quelconques, puis nous montrons comment certaines de ces relations sont identifiées lorsque l'on se restreint à des systèmes moins généraux. Nous énonçons tout d'abord un certain nombre de propositions préliminaires pour la comparaison de ces différentes relations.

2.6.1 Propositions préliminaires

La proposition 2.6-1 compare les équivalences de simulation et de bisimulation obtenues pour un même critère d'abstraction.

Proposition 2.6-1
Soit Λ un ensemble de langages sur $\mathcal{A}$. L'équivalence de bisimulation obtenue pour Λ est plus forte que l'équivalence de simulation correspondante :

$$\sim_\Lambda \subseteq \approx_\Lambda$$

■

Preuve : elle est immédiate. Par définition des relations $\sim_\Lambda$ et $\sqsubseteq_\Lambda$, on a :

$$\sim_\Lambda \subseteq \sqsubseteq_\Lambda$$

Puisque $\sim_\Lambda$ est symétrique, on a également :

$$\sim_\Lambda \subseteq (\sqsubseteq_\Lambda)^{-1}$$

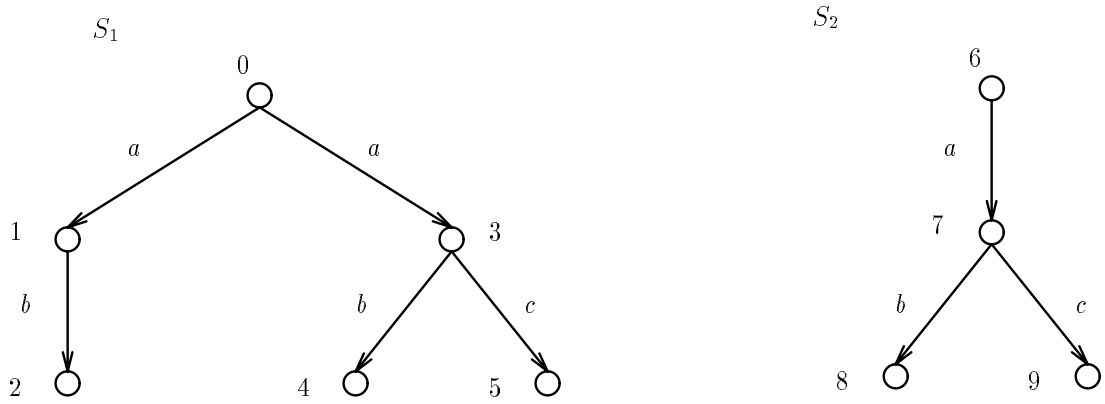
Par suite,

$$\sim_\Lambda \subseteq (\sqsubseteq_\Lambda \cap (\sqsubseteq_\Lambda)^{-1})$$

□.

L'exemple suivant permet de voir que cette inclusion est stricte (les deux relations sont bien différentes) :

Exemple 2-7



Le critère d'abstraction considéré est le critère "fort", pour lequel toute action est observable. Il correspond donc à l'ensemble de langages :

$$\Lambda = \{\{a\} \mid a \in \mathcal{A} \cup \{\tau\}\}$$

. On a $S_1 \sqsubseteq_\Lambda S_2$ et $S_2 \sqsubseteq_\Lambda S_1$, mais $S_1 \not\sim_\Lambda S_2$. ■

Dans l'exemple précédent, la transition $0 \xrightarrow{a} 1$, qui met en cause l'existence d'une bisimulation entre S_1 et S_2 est en fait une *transition redondante* [Rod88]. Nous montrons que les relations $\sim_\Lambda$ et $\sqsubseteq_\Lambda$ coïncident lorsque l'on considère des systèmes de transitions étiquetées dépourvus de transitions redondantes.

Définition 2.6-1
Soit $S = (Q, A, T, q_0)$ un système de transitions étiquetées, soit $p \in Q$, soit Λ un ensemble de langages sur $\mathcal{A}$ et soit $\lambda \in \Lambda$.

La transition $p \xrightarrow{\lambda}_T p'$ est redondante pour Λ si et seulement si :

$$\exists p" \ . \ (p \xrightarrow{\lambda}_T p" \ \wedge \ p' \neq p" \ \wedge \ p' \sqsubseteq_\Lambda p")$$

■

Proposition 2.6-2
Les relations $\sim_\Lambda$ et $\approx_\Lambda$ coïncident sur l'ensemble des systèmes de transitions étiquetées dépourvus de transitions redondantes pour Λ. ■

Preuve : Soient S_1 et S_2 deux systèmes de transitions étiquetées dépourvue de transitions redondantes pour l'ensemble de langages Λ. On montre :

$$S_1 \approx_\Lambda S_2 \quad - \quad S_1 \sim_\Lambda S_2$$

Puisque $\sim_\Lambda \subseteq \approx_\Lambda$ (proposition 2.6-1), il suffit de vérifier que l'on a :

$$S_1 \sqsubseteq_\Lambda S_2 \quad \Rightarrow \quad S_1 \sim_\Lambda S_2 .$$

Soient $p_1 \in Q_1$ et $p_2 \in Q_2$. On montre par induction sur k que :

$$\forall k \, . \, (p_1 \sqsubseteq_\Lambda p_2 \ \wedge \ p_2 \sqsubseteq_\Lambda p_1) \quad \Rightarrow \quad p_1 \sim^k_\Lambda p_2$$

- pour $k = 0$, la proposition est évidente.
- Soit $k > 0$. On suppose :

  $$\forall i, \ i \leq k \ (p_1 \sqsubseteq_\Lambda p_2 \ \wedge \ p_2 \sqsubseteq_\Lambda p_1) \quad \Rightarrow \quad p_1 \sim^i_\Lambda p_2$$

  On montre alors que $p_1 \sim^{k+1}_\Lambda p_2$. Soit (λ, p'_1) tel que :

  $$\lambda \in \Lambda \ \wedge \ p_1 \xrightarrow{\lambda}_{T_1} p'_1$$

  On a,

  $$p_1 \sqsubseteq_\Lambda p_2 \ \wedge \ p_1 \xrightarrow{\lambda}_{T_1} p'_1 \quad \Rightarrow \quad \exists p'_2 \, . \, p_2 \xrightarrow{\lambda}_{T_2} p'_2 \ \wedge \ p'_1 \sqsubseteq_\Lambda p'_2$$

  $$p_2 \sqsubseteq_\Lambda p_1 \ \wedge \ p_2 \xrightarrow{\lambda}_{T_2} p'_2 \quad \Rightarrow \quad \exists p''_1 \, . \, p_1 \xrightarrow{\lambda}_{T_1} p''_1 \ \wedge \ p'_2 \sqsubseteq_\Lambda p''_1$$

  Or, par transitivité de la relation $\sqsubseteq_\Lambda$:

  $$p'_1 \sqsubseteq_\Lambda p'_2 \ \wedge \ p'_2 \sqsubseteq_\Lambda p''_1 \ \Rightarrow \ p'_1 \sqsubseteq_\Lambda p''_1$$

  On a donc :

  $$p_1 \xrightarrow{\lambda}_{T_1} p'_1 \ \wedge \ p_1 \xrightarrow{\lambda}_{T_1} p''_1 \ \wedge \ p'_1 \sqsubseteq_\Lambda p''_1$$

  Par suite, S_1 ne comportant pas de transitions redondantes pour l'ensemble de langages Λ, les états p'_1 et p''_1 sont confondus, d'où $p'_2 \sqsubseteq p'_1$. Par hypothèse d'induction,

  $$(p'_1 \sqsubseteq_\Lambda p'_2 \ \wedge \ p'_2 \sqsubseteq_\Lambda p'_1) \quad \Rightarrow \quad p'_1 \sim^k_\Lambda p'_2$$

  On prouve de façon similaire :

  $$\forall (\lambda, p'_2) \, . \, p_2 \xrightarrow{\lambda}_{T_2} p'_2 \quad \Rightarrow \quad \exists p'_1 \, . \, (p_1 \xrightarrow{\lambda}_{T_1} p'_1 \ \wedge \ p'_1 \sim^k_\Lambda p'_2)$$

  Par suite, on a bien $p_1 \sim^{k+1}_\Lambda p_2$.

□.

La proposition 2.6-3 permet de comparer les équivalences de bisimulation (*resp.* les préordres de simulation) obtenues pour des ensembles de langages Λ_1 et Λ_2 entre lesquelles il existe une relation d'"inclusion". Plus précisément, nous montrons que lorsque les langages de Λ_1 et Λ_2 sont donnés sous forme d'*expressions régulières*, et lorsque tout λ_1 de Λ_1 peut être obtenue par concaténation d'un nombre fini de langages λ_2 de Λ_2, alors :

- l'équivalence de bisimulation $\sim_{\Lambda_1}$ est plus forte que l'équivalence de bisimulation $\sim_{\Lambda_2}$,
- le préordre de simulation $\sqsubseteq_{\Lambda_1}$ est plus fort que le préordre de simulation $\sqsubseteq_{\Lambda_2}$.

Définition 2.6-2
Soient Λ_1 et Λ_2 deux ensembles de langages réguliers définis sur $\mathcal{A} \cup \{\tau\}$. On définit une relation d'"inclusion" (notée $\ll$) entre Λ_1 et Λ_2 de la façon suivante :

$\Lambda_1 \ll \Lambda_2$ –

$$\forall \lambda_1 \, . \, (\lambda_1 \in \Lambda_1 \ \Rightarrow \ (\exists \, \lambda_2^1, \lambda_2^2, \ldots, \lambda_2^n \, . \, ((\forall i \, . \, \lambda_2^i \in \Lambda_2) \ \wedge \ (\lambda_1 \subseteq (\lambda_2^1)^* (\lambda_2^2)^* \ldots (\lambda_2^n)^*))))$$

■

Exemple 2-8

- L'ensemble de langages défini pour la τ^*a-bisimulation est inclus dans celui défini pour la bisimulation forte :

$$\{\tau^*a \mid a \in \mathcal{A}\} \ll \{\{a\} \mid a \in \mathcal{A} \cup \{\tau\}\}$$

- Les ensembles de langages définis pour la τ^*a-bisimulation et pour l'équivalence observationnelle sont incomparables vis-à-vis de la relation $\ll$:

$$\{\tau^*a \mid a \in \mathcal{A}\} \not\ll \{\{\tau^*\} \cup \{\tau^*a\tau^* \mid a \in \mathcal{A}\}\}$$
$$\{\{\tau^*\} \cup \{\tau^*a\tau^* \mid a \in \mathcal{A}\}\} \not\ll \{\tau^*a \mid a \in \mathcal{A}\}$$

■

Proposition 2.6-3
Soient Λ_1 et Λ_2 deux ensembles de langages réguliers définis sur $\mathcal{A} \cup \{\tau\}$.

$$\Lambda_1 \ll \Lambda_2 \Rightarrow (\sqsubseteq_{\Lambda_2} \subseteq \sqsubseteq_{\Lambda_1}) \wedge (\sim_{\Lambda_2} \subseteq \sim_{\Lambda_1})$$

■

Preuve : Nous effectuons la preuve uniquement pour les préordres de simulation (le cas des équivalences de bisimulation pouvant être traité de manière identique par symétrie). Soit R une relation sur $Q_1 \times Q_2$. On montre que si R est un préordre de simulation pour Λ_2 alors c'est également un préordre de simulation pour Λ_1 :

$$R \subseteq \mathcal{I}_{\Lambda_2}(R) \Rightarrow R \subseteq \mathcal{I}_{\Lambda_1}(R)$$

Soit $R \subseteq \mathcal{I}_{\Lambda_2}(R)$, soient $(p,q) \in R$, et soit p' tel que $p \xrightarrow{\lambda_1}_{T_1} p'$ pour $\lambda_1 \in \Lambda_1$. Comme $\Lambda_1 \ll \Lambda_2$, on a :

$$p \xrightarrow{\lambda_1}_{T_1} p' \Rightarrow (\exists \lambda_2^1, \lambda_2^2, \ldots, \lambda_2^n \in \Lambda_2) \,.\, p \xrightarrow{(\lambda_2^1)^*}_{T_1} p_1 \xrightarrow{(\lambda_2^2)^*}_{T_1} p_2 \cdots p_{n-1} \xrightarrow{(\lambda_2^n)^*}_{T_1} p'$$

Or, par hypothèse :

$$(p,q) \in R \Rightarrow (p,q) \in \mathcal{I}_{\Lambda_2}(R)$$

d'où :

$$p \xrightarrow{(\lambda_1)^*}_{T_1} p' \Rightarrow \exists q_1, \ldots, q_n \,.\, q_i \xrightarrow{(\lambda_2^{i+1})^*}_{T_2} q_{i+1} \wedge (p_i, q_i) \in R$$
$$\Rightarrow (p_i, q_i) \in \mathcal{I}_{\Lambda_2}(R), \text{ puisque } R \text{ est un préordre de simulation pour } \Lambda_2$$

Par suite,

$$p \xrightarrow{\lambda_1}_{T_1} p' \Rightarrow \exists q' \,.\, q \xrightarrow{\lambda_1}_{T_2} q' \wedge (p', q') \in R$$

ce qui justifie que R est bien un préordre de simulation pour l'ensemble de langages Λ_1. □.

Enfin, nous donnons un corollaire à la proposition 2.6-3.

Corollaire 2.6-1
Soient Λ_1 et Λ_2 deux ensembles de langages réguliers sur $\mathcal{A} \cup \{\tau\}$.

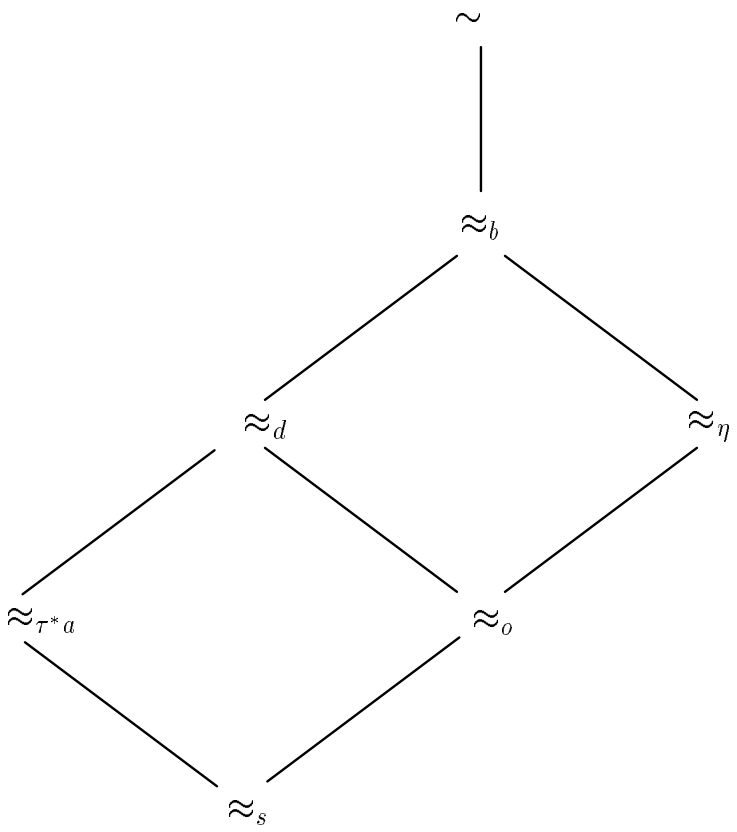
$$\Lambda_1 \subseteq \Lambda_2 \Rightarrow (\sqsubseteq_{\Lambda_2} \subseteq \sqsubseteq_{\Lambda_1}) \wedge (\sim_{\Lambda_2} \subseteq \sim_{\Lambda_1})$$

■

Preuve : Elle est immédiate puisque $\subseteq \Rightarrow \ll$. □.

2.6.2 Cas général

Nous sommes désormais en mesure de comparer les relations d'équivalence définies dans ce chapitre lorsque l'on considère des systèmes de transitions étiquetées quelconques. Plus précisément, nous montrons que ces différentes relations sont ordonnées vis-à-vis de l'inclusion selon le treillis suivant :



Cette figure a le mérite de présenter de manière synthétique un ensemble de résultats qui vont être justifiés dans la suite en construisant les différentes portions du treillis et en comparant entre eux les extremums ainsi obtenus.

Nous construisons tout d'abord la portion de treillis contenant l'équivalence observationnelle ($\approx_o$) et les équivalences de branchement présentées dans la section 2.4 ($\approx_b$, $\approx_d$ et $\approx_\eta$). Ces relations pouvant être définies à partir d'une même relation plus générale $\mathcal{R}_C$ paramétrée par une fonction booléenne (définition 2.4-1), nous utiliserons la proposition suivante qui permet de comparer deux relations $\mathcal{R}_{C_1}$ et $\mathcal{R}_{C_2}$ en fonction de C_1 et C_2 :

Proposition 2.6-4
Soient C_1 et C_2 deux fonctions booléennes.

$$(C_1 \Rightarrow C_2) \Rightarrow (\mathcal{R}_{C_1} \subseteq \mathcal{R}_{C_2})$$

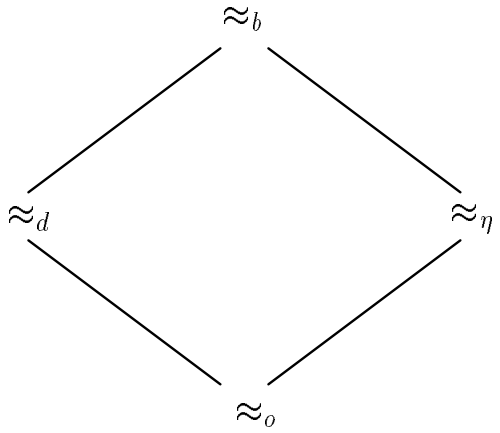
■

Preuve : immédiate □.

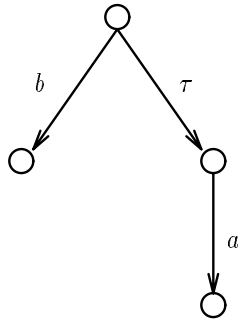
D'autre part, on a les implications suivantes entre les fonctions C_{br}, C_d, C_η et C_{obs} :

$C_{br} \Rightarrow C_d$
$C_{br} \Rightarrow C_\eta$
$C_d \Rightarrow C_{obs}$
$C_\eta \Rightarrow C_{obs}$

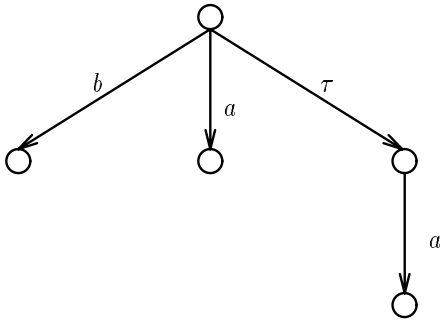
Par suite, et d'après la proposition 2.6-4, on obtient bien la portion de treillis suivante :



Il est facile de voir que les inclusions $\approx_b \subseteq \approx_d$ et $\approx_b \subseteq \approx_\eta$ sont strictes. La figure suivante fournit un contre-exemple pour la première d'entre elles, un contre-exemple pour la seconde étant fourni par la figure de l'exemple 2-1.



S_3



S_4

Les systèmes de transitions étiquetées S_3 et S_4 sont équivalents pour la relation $\approx_d$ et ne le sont pas pour $\approx_b$. De même, les systèmes de transitions étiquetées S_1 et S_2 (de l' exemple 2-1) sont équivalents pour la relation $\approx_\eta$ et ne le sont pas pour $\approx_b$.

Nous comparons maintenant les relations préservant les propriétés de sûreté présentées dans la section 2.5. D'après la proposition 2.6-1, on a immédiatement $\approx_{\tau^* a} \subseteq \approx_s$, ce qui correspond à la portion de treillis suivante :

Pour terminer la comparaison (et obtenir le treillis de la figure 2.6.2), il reste à préciser les positions relatives de ces deux portions de treillis déjà construites, et y inclure la relation de bisimulation forte $\sim$. Nous comparons tout d'abord la relation $\approx_{\tau^* a}$ avec les relations $\approx_d$, $\approx_\eta$ et $\approx_o$.

Proposition 2.6-5

(*i*) $\approx_d \subseteq \approx_{\tau^* a}$

(*ii*) $(\approx_{\tau^* a} \not\subseteq \approx_o) \wedge (\approx_o \not\subseteq \approx_{\tau^* a})$

(*iii*) $(\approx_{\tau^* a} \not\subseteq \approx_\eta) \wedge (\approx_\eta \not\subseteq \approx_{\tau^* a})$

■

Preuve :

(*i*) C'est une conséquence directe du corollaire 2.6-1. On a, en effet, l'inclusion suivante entre les ensembles de langages définissant la delay bisimulation et la $\tau^* a$-bisimulation :

$$(\{\tau^*\} \cup \{\tau^* a \mid a \in \mathcal{A}\}) \subseteq \{\tau^* a \mid a \in \mathcal{A}\}$$

(*ii*) Les systèmes de transitions étiquetées S_1 et S_2 de l'exemple 2-1 sont équivalents pour les relations $\approx_o$ et $\approx_\eta$, mais ne le sont pas pour la relation $\approx_{\tau^* a}$.

(*iii*) Les systèmes de transitions étiquetées de la figure suivante sont équivalents pour la relation $\approx_{\tau^* a}$ mais ne le sont ni pour la relation $\approx_o$, ni pour la relation $\approx_\eta$.

□.

Il reste alors à comparer les relations $\approx_s$ et $\approx_o$. Pour ce faire, nous montrons tout d'abord que les relations de préordre $\sqsubseteq_s$ et $\sqsubseteq_o$ qui leurs sont associées coïncident.

Définition 2.6-3
Soit $\sqsubseteq_o$ la relation définie sur $Q_1 \times Q_2$ par :

$p \sqsubseteq_o q$ $-$

$\forall a \in \mathcal{A} \cup \{\tau\} \,.\, \forall p' \,.\, p \xrightarrow{a}_{T_1} p' \;\Rightarrow\; (a = \tau \wedge p' \sqsubseteq_o q) \vee (\exists q' \,.\, q \xrightarrow{\tau^* a \tau^*}_{T_2} q' \wedge p' \sqsubseteq_o q')$ ■

Il est facile de voir que la relation $\sqsubseteq_o$ est le préordre de simulation obtenu pour l'ensemble de langages

$$\Lambda = \{\tau^*\} \cup \{\tau^* a \tau^* \mid a \in \mathcal{A}\}$$

Proposition 2.6-6
Soient S_1 et S_2 deux systèmes de transitions étiquetées.

$$S_1 \sqsubseteq_s S_2 \quad - \quad S_1 \sqsubseteq_o S_2$$

■

La preuve de cette proposition est basée sur le lemme suivant :

Lemme 2.6-1
Soit $S = (Q, A, T, q_0)$ un système de transitions étiquetées.

$$\forall p \in Q,\ p \xrightarrow{\tau}_T p' \ \Rightarrow \ p' \sqsubseteq_s p \ \wedge \ p' \sqsubseteq_o p$$

■

On trouve une preuve de ce lemme pour la relation $\sqsubseteq_s$ dans [Rod88], et il est aisé d'en déduire une preuve similaire pour la relation $\sqsubseteq_o$.

Preuve (de la proposition 2.6-6) : Soit $(p, q) \in Q_1 \times Q_2$.

sens '$\Rightarrow$' : On montre que la relation $\sqsubseteq_s$ satisfait bien les conditions de la définition de $\sqsubseteq_o$ (définition 2.6-3). On suppose $p \sqsubseteq_s q$. Soit $a \in \mathcal{A} \cup \{\tau\}$, et soit p' tel que $p \xrightarrow{a}_{T_1} p'$.

- Si $a = \tau$, alors, d'après le lemme 2.6-1, on a $p' \sqsubseteq_s p$, d'où $p' \sqsubseteq_s q$.
- Si $a \neq \tau$, alors, par définition de $\sqsubseteq_s$: $\exists q' \,.\, q \xrightarrow{\tau^* a}_{T_2} q' \ \wedge \ p' \sqsubseteq_s q'$

Par suite, on a $\sqsubseteq_s \subseteq \sqsubseteq_o$.

sens '$\Leftarrow$' : De façon similaire, on montre que la relation $\sqsubseteq_o$ est bien un préordre de simulation pour l'ensemble de langages $\Lambda = \{\tau^* a \mid a \in \mathcal{A}\}$. On suppose $p \sqsubseteq_o q$. Soit $a \in \mathcal{A}$, et soit p' tel que $p \xrightarrow{\tau^* a}_{T_1} p'$. Plus précisément,

$$\exists p_1, \ldots, p_n \,.\, p \xrightarrow{\tau}_{T_1} p_1 \xrightarrow{\tau}_{T_1} \cdots \xrightarrow{\tau}_{T_1} p_n \ \wedge \ p_n \xrightarrow{a}_{T_1} p'$$

Or, d'après le définition 2.6-3,

$$p \sqsubseteq_o q \ \wedge \ p \xrightarrow{\tau}_{T_1} p_1 \ \Rightarrow \ p_1 \sqsubseteq_o q \ \vee \ (\exists q_1 \,.\, q \xrightarrow{\tau}_{T_2} q_1 \ \wedge \ p_1 \sqsubseteq_o q_1)$$

De proche en proche,

$$\exists q_n \,.\, q \xrightarrow{\tau^*}_{T_2} q_n \ \wedge \ p_n \sqsubseteq_o q_n$$

Par suite, par définition de $\sqsubseteq_o$,

$$p_n \xrightarrow{a}_{T_1} p' \ \Rightarrow \ \exists q", q' \,.\, q \xrightarrow{\tau^* a}_{T_2} q" \xrightarrow{\tau^*}_{T_2} q' \ \wedge \ p' \sqsubseteq_o q'$$

Or, d'après le lemme 2.6-1, on a $q' \sqsubseteq_o q"$, d'où $p' \sqsubseteq_o q"$. Par suite, on a bien :

$$\forall a \in \mathcal{A} \,.\, \forall p' \,.\, p \xrightarrow{\tau^* a}_{T_1} p' \ \Rightarrow \ \exists q" \,.\, q \xrightarrow{\tau^* a}_{T_2} q" \ \wedge \ p' \sqsubseteq_o q".$$

Proposition 2.6-7

$$\approx_o \subseteq \approx_s$$

■

Preuve : On note $\approx_o^{sim}$ l'équivalence de simulation obtenue à partir du préordre de simulation $\sqsubseteq_o$:

$$\approx_o^{sim} = \sqsubseteq_o \ \wedge (\sqsubseteq_o)^{-1}$$

On a alors, d'après la proposition 2.6-6, $\sqsubseteq_o \equiv \sqsubseteq_s \Rightarrow \approx_o^{sim} \equiv \approx_s$. Or, d'après la proposition 2.6-1, $\approx_o \subseteq \approx_o^{sim}$ d'où $\approx_o \subseteq \approx_s$. □.

Enfin, en effectuant une preuve analogue à celle de la proposition 2.6-3, il est facile de voir que l'on a $\sim \subseteq \approx_b$.

Des propositions 2.6-5 et 2.6-7 on déduit alors le treillis présenté au début de cette section, qui ordonne vis-à-vis de la relation d'inclusion les différentes relations d'équivalence présentées dans ce chapitre lorsque l'on considère des systèmes de transitions étiquetées quelconques.

2.6.3 Cas particuliers

On considère maintenant des classes plus particulières de systèmes de transitions étiquetées, et on examine comment est modifié le treillis présenté dans le section précédente. Plus précisément, nous nous intéressons à trois cas de figure que l'on rencontre souvent en pratique, et qui sont respectivement :

- La comparaison d'un système de transitions étiquetées *tau-free* (i.e, ne comportant pas de τ-transitions) avec un système de transitions étiquetées quelconque ;
- La comparaison ou la minimisation de systèmes de transitions étiquetées dont le quotient par la bisimulation de branchement est *τ-free* ;
- La comparaison ou la minimisation de systèmes de transitions étiquetées dont le quotient par la bisimulation de branchement est *τ-free* et déterministe.

Nous rappelons tout d'abord dans la proposition suivante les résultats obtenus lorsque l'on considère des classes de systèmes de transitions étiquetées encore plus particulières — et sans doute assez peu intéressante en pratique — que sont les systèmes de transitions étiquetées τ-free et déterministes.

Définition 2.6-4
Un système de transitions étiquetées $S = (Q, A, T, q_0)$ est τ-free si et seulement si

$$\forall p \in Q,\ T_\tau[p] = \emptyset$$

■

Proposition 2.6-8

(i) Pour deux systèmes de transitions étiquetées τ-free, les relations $\sim$, $\approx_b$, $\approx_d$, $\approx_\eta$, $\approx_o$ et $\approx_{\tau^* a}$ coïncident.

(ii) Pour deux systèmes de transition étiquetées τ-free et déterministes, les relations précédentes coïncident également avec la relation $\approx_s$.

■

Preuve :

(i) Il est immédiat que les équivalences de bisimulation $\sim$, $\approx_o$ et $\approx_{\tau^* a}$ coïncident lorsque l'on considère des systèmes τ-free. De même, il est facile de voir à partir de la définition de $\mathcal{R}_C$ qu'il en est de même des relations $\approx_o$, $\approx_b$, $\approx_d$ et $\approx_\eta$.

(ii) Si les systèmes de transitions sont τ-free et déterministes, ils ne comportent pas de transition redondantes pour le critère d'observation associé à la relation $\approx_s$. Par suite, d'après la proposition 2.6-2, les relations $\approx_s$ et $\approx_{\tau^* a}$ coïncident.

□.

Dans la pratique, il est assez fréquent que le système de transitions étiquetées qui décrit une spécification comportementale ne comporte pas de τ-transitions. En effet, lorsque les actions que l'on souhaite observer sont convenablement choisies, il semble assez peu naturel de spécifier le comportement attendu d'un programme en y incluant des actions internes. La vérification de ce type de spécifications consiste alors à comparer, pour une relation d'équivalence donné, un système de transitions étiquetées τ-free (modélisant la spécification) avec un système de transitions étiquetées quelconque (modélisant le programme). Nous montrons que dans ce cas de figure certaines des équivalences que nous avons présentées coïncident.

Proposition 2.6-9
Soient S_1 et S_2 deux systèmes de transitions étiquetées tels que S_2 (ou S_1) soit τ-free.

$$S_1 \approx_o S_2 \quad - \quad S_1 \approx_b S_2$$

■

Preuve : Nous montrons uniquement le sens '$\Rightarrow$', le sens '$\Leftarrow$' étant une conséquence de la proposition 2.6-4. Si l'on suppose S_2 τ-free, la définition de la relation $\approx_b$ est la disjonction des trois propriétés suivantes :

(i) $\forall p' \,.\, p \xrightarrow{\tau}_{T_1} p' \;\Rightarrow\; p' \approx_b q$

(ii) $\forall a \in \mathcal{A} \,.\, \forall p' \,.\, p \xrightarrow{a}_{T_1} p' \;\Rightarrow\; \exists q' \,.\, q \xrightarrow{a}_{T_2} q' \;\wedge\; p' \approx_b q'$

(iii) $\forall a \in \mathcal{A} \,.\, \forall q' \,.\, q \xrightarrow{a}_{T_2} q' \;\Rightarrow\; \exists p_1, p' \,.\, p \xrightarrow{\tau^*}_{T_1} p_1 \xrightarrow{a}_{T_1} p' \;\wedge\; p_1 \approx_b q \;\wedge\; p' \approx_b q'$

On montre alors, par induction sur k, que :

$$\forall k \,.\, p \approx_o q \;\Rightarrow\; p \approx_b^k q$$

Pour $k = 0$, la proposition est évidente. Soit $k > 0$, on suppose

$$\forall i, i \leq k \,.\, p \approx_o q \;\Rightarrow\; p \approx_b^i q$$

On montre alors que $p \approx_b^{k+1} q$. Comme $T_\tau[q] = \emptyset$, les cas (i) et (ii) sont immédiatement vérifiés. Le cas (iii) se justifie de la façon suivante :

$$p \approx_o q \;\wedge\; q \xrightarrow{a}_{T_2} q' \;\Rightarrow\; \exists p_1, p', p_2 \,.\, p \xrightarrow{\tau^*}_{T_1} p_1 \xrightarrow{a}_{T_1} p' \xrightarrow{a}_{T_1} p_2 \;\wedge\; p_2 \approx_o q'$$

Or,

$$\begin{aligned} p \approx_o q \;\wedge\; p \xrightarrow{\tau^*}_{T_1} p_1 \quad &\Rightarrow \quad p_1 \approx_o q \text{ car } T_\tau[q] = \emptyset \\ &\Rightarrow \quad p_1 \approx_b^k q \end{aligned}$$

D'autre part,

$$p_1 \approx_o q \;\wedge\; p_1 \xrightarrow{a}_{T_1} p' \;\Rightarrow\; \exists q" \,.\, q \xrightarrow{a}_{T_2} q" \;\wedge\; p' \approx_o q" \text{ car } T_\tau[q] = \emptyset$$

et,

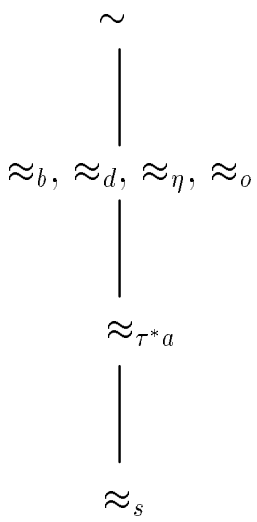
$$p' \approx_o q" \;\wedge\; p' \xrightarrow{\tau^*}_{T_1} p_2 \;\Rightarrow\; p_2 \approx_o q"$$

Par suite, on a

$$p' \approx_o q" \;\wedge\; p_2 \approx_o q" \;\wedge\; p_2 \approx_o q' \;\Rightarrow\; p' \approx_o q' \;\Rightarrow\; p' \approx_b^k q'$$

En conséquence, (iii) est vérifié, et on a bien $p \approx_b^{k+1} q$. □.

D'après la proposition 2.6-9, lorsque l'on compare un système de transitions étiquetées τ-free avec un système de transitions étiquetées quelconque, les relations d'équivalence présentées dans ce chapitre sont ordonnées selon le treillis suivant :



La proposition 2.6-10, permet de comparer les relations d'équivalence présentées dans ce chapitre lorsque l'on se restreint aux systèmes de transitions étiquetées dont le quotient par la bisimulation de branchement est τ-free (*resp.* τ-free et déterministe).

Proposition 2.6-10
Soient S_1 et S_2 deux systèmes de transitions étiquetées.

(i) Si les quotients de S_1 et S_2 pour l'équivalence de branchement sont τ-free, alors :

$$S_1 \approx_{\tau^* a} S_2 \quad - \quad S_1 \approx_b S_2$$

(ii) Si les quotients de S_1 et S_2 pour l'équivalence de branchement sont τ-free et déterministes, alors

$$S_1 \approx_s S_2 \quad - \quad S_1 \approx_b S_2$$

■

Preuve : Il suffit dans chaque cas de montrer les sens directs.

(i) Par l'absurde : on suppose $S_1 \approx_{\tau^* a} S_2 \ \wedge \ S_1 \not\approx_b S_2$ Comme $\approx_b \subseteq \approx_{\tau^* a}$, on a

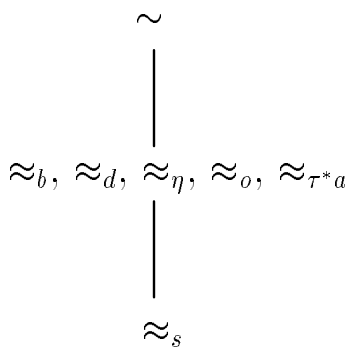
$$S_1/_{\approx_b} \approx_{\tau^* a} S_2/_{\approx_b} \ \wedge \ S_1/_{\approx_b} \not\approx_b S_2/_{\approx_b}$$

Or, $S_1/_{\approx_b}$ et $S_2/_{\approx_b}$ étant τ-free, les relations $\approx_{\tau^* a}$ et $\approx_b$ coïncident pour ces deux systèmes (proposition 2.6-8). Par suite, on a nécessairement $S_1 \approx_b S_2$.

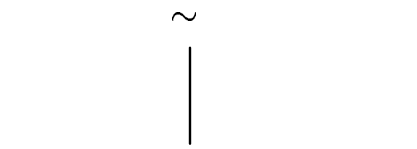
(ii) La preuve est similaire à celle du cas précédent.

□.

Par suite, lorsque l'on considère des systèmes de transitions étiquetées dont le quotient par la bisimulation de branchement est τ-free (*resp.* τ-free et déterministe), les relations d'équivalence sont ordonnées selon le treillis Π_1 (*resp.* Π_2) suivant :



Π_1



Π_2

En conclusion, l'expérience montre que les systèmes de transitions étiquetées que l'on est amené à comparer dans le cadre de la vérification de spécifications comportementales ont souvent un quotient par la bisimulation de branchement qui est τ-*free*. En théorie, la minimisation ou la comparaison de tels systèmes par rapport à n'importe quelle relation d'équivalence comprise entre la bisimulation de branchement et la τ^*a-bisimulation pourrait donc se faire en utilisant l'une ou l'autre des procédures de décision relatives à ces deux relations. Néanmoins, cette solution reste limitée dans le sens où nous avons pas actuellement de critère pour décider efficacement si un système de transition étiqueté appartient à cette classe sans l'avoir généré entièrement au préalable.

La deuxième classe de systèmes de transitions étiquetées, ceux dont le quotient par la bisimulation de branchement (ou par l'équivalence observationnelle, *cf.* proposition 2.6-9) est τ-free et déterministe est mentionné dans [Mil89] sous le nom de *"determinate or predictable systems"*. L'idée intuitive associée est la suivante :

> Un système est "prévisible" s'il présente toujours le même comportement observable chaque fois qu'il est plongé dans le même environnement.

Cette classe particulière de systèmes est également étudié dans [Qin91], qui propose une procédure de décision plus efficace que celles employées habituellement pour l'équivalence observationnelle et la bisimulation de branchement est décrite. Dans le même esprit, nous proposons au chapitre 5 un algorithme de comparaison "à la volée" adapté à la comparaison et à la minimisation des systèmes "prévisibles", et permettant de décider si un système de transitions étiquetées est "prévisible" ou non sans le générer entièrement au préalable.

Toutefois, il semble peu probable que cette classe de systèmes soit suffisante pour répondre à tous les besoins. En effet, même si les systèmes que l'on traite (tels les protocoles de communication) sont souvent "prévisibles" lorsque l'on observe leur comportement global, il n'en est pas de même lors de la vérification où l'on ne s'intéresse qu'à une *abstraction* de ce comportement.

Exemple 2-9
Le système de transitions étiquetées obtenu lors de la vérification du protocole *rel/REL* [BM91] (*cf.* exemple 2-2 et annexe B) n'est pas un système "prévisible" pour la τ^*a-bisimulation lorsque l'on considère comme visibles uniquement les actions de réception effectuées par l'une des stations R. Le quotient de ce système pour cette relation d'équivalence n'est pas déterministe, du fait que R puisse arbitrairement choisir entre deux types de comportements :

- recevoir le message i et continuer à fonctionner jusqu'a réception du message $i+1$,
- recevoir le message i et cesser de fonctionner avant réception du prochain message.

■

Partie II

Les méthodes classiques de vérification

Chapitre 3

Les algorithmes classiques

Nous présentons dans ce chapitre les procédures de décision couramment utilisées dans les outils de vérification classiques basés sur les relations d'équivalence comportementales. Nous rappelons tout d'abord le principe général sur lequel reposent les algorithmes de comparaison et réduction des systèmes de transitions étiquetées modulo une équivalence de bisimulation, puis nous décrivons ensuite de façon plus détaillée les algorithmes mis en œuvre en pratique dans le cas des relations présentées au chapitre précédent.

3.1 Calcul d'une partition compatible avec une relation de transition

La comparaison et la réduction des systèmes de transition étiquetées modulo une équivalence de bisimulation sont toutes deux basées sur un même algorithme, qui consiste à calculer les classes d'équivalence pour cette relation sur les ensembles d'états des systèmes (i.e, la partition qui lui est associée). Nous décrivons dans cette section le principe de cet algorithme, qui repose sur la notion de *compatibilité* d'une partition avec une relation de transition.

Définition 3.1-1
Soit $S = (Q, A, T, q_0)$ un système de transitions étiquetées, soit $\rho = \{Bi \mid i \in I\}$ une partition sur Q, et soit Λ un ensemble de langages disjoints sur A. La partition ρ est *compatible* avec T pour l'ensemble de langages Λ si et seulement si :

$$\forall \lambda \in \Lambda \,.\, \forall i, j \in I \,.\, \forall p, q \in B_i \,.\, (T_\lambda[p] \cap B_j \neq \emptyset \;-\; T_\lambda[q] \cap B_j \neq \emptyset)$$

■

Le lien entre relation de bisimulation et partition compatible avec une relation de transition est établi dans la proposition 3.1-1 :

Proposition 3.1-1
Soit $S = (Q, A, T, q_0)$ un système de transitions étiquetées et soit R une relation binaire définie sur $Q \times Q$.

R est une bisimulation pour l'ensemble de langages Λ si et seulement si la partition associée à R est compatible avec T pour Λ. ■

La preuve de cette proposition s'obtient immédiatement en comparant les définitions de "compatibil-

ité" et "relation de bisimulation".

La proposition 3.1-1 permet de ramener le problème du calcul des classes d'équivalence pour une bisimulation à la recherche de la plus grande partition (vis-à-vis de l'opérateur de raffinement $\sqsubseteq$, *cf.* chapitre 1) qui soit compatible avec une relation de transition :

> Etant donnée une partition initiale ρ_{init} sur un ensemble Q, une relation T sur $Q \times Q$ étiquetée par les éléments d'un ensemble A, et Λ un ensemble de langages sur A, trouver le plus grand raffinement ρ' de ρ_{init} qui soit compatible avec T pour Λ.

L'énoncé précédent (ou l'une des ses variantes) est connu sous le nom de *Relational Coarsest Partition problem*, ou encore *RCP problem*. Le treillis des partitions (ordonné par l'opérateur $\sqsubseteq$) étant un treillis complet, le *RCP problem* admet bien une solution unique ρ, qui coïncide avec la partition associée à la plus grande bisimulation sur $Q \times Q$ contenant ρ_{init}. Par suite, si ρ_{init} est la partition universelle (i.e, $\rho_{init} = \{Q\}$), ρ est bien l'équivalence de bisimulation sur $Q \times Q$.

Le lien entre *RCP problem* et relation de bisimulation a été établi par Kanellakis et Smolka [KS83], qui ont proposé une solution dans le cas de la bisimulation forte dont les coûts en temps et mémoire sont respectivement $O(m.n)$ et $O(m+n)$, où m et n représentent respectivement les nombres d'états et de transitions ($m = |T|$ et $n = |Q|$). Paige et Tarjan ont amélioré cet algorithme et obtenu une solution dont le coût en temps est $O(m.log(n))$ [PT87]. C'est cette solution qui a été implémentée dans le cadre du logiciel ALDÉBARAN [Fer88].

Néanmoins, ces deux algorithmes restent identiques dans leur principe : ils consistent à calculer des raffinements successifs (ρ_i) de la partition initiale ρ_{init} jusqu'à *stabilité*, de sorte que chaque nouvelle partition ρ_{i+1} obtenue soit compatible avec la relation de transition T par rapport à la partition précédente ρ_i. La partition finale ρ ainsi obtenue est alors solution du *RPC problem*.

Afin de présenter plus formellement ce principe, nous donnons tout d'abord une autre caractérisation de la compatibilité, puis nous précisons les notions de "stabilité" et de partition "compatible par rapport à une autre".

Proposition 3.1-2
Soit $S = (Q, A, T, q_0)$ un système de transitions étiquetées, soit $\rho = \{Bi \mid i \in I\}$ une partition sur Q, et soit Λ un ensemble de langages sur A. La partition ρ est *compatible* avec T pour l'ensemble de langages Λ si et seulement si :

$$\forall \lambda \in \Lambda \,.\, \forall i, j \in I \,.\, (B_i \subseteq T_\lambda^{-1}(B_j) \;\vee\; B_i \cap T_\lambda^{-1}(B_j) = \emptyset)$$

■

Preuve : elle est donnée dans [Fer88]. □.

Définition 3.1-2

Soit $S = (Q, A, T, q_0)$ un système de transitions étiquetées, soient ρ et ρ' des partitions sur Q, et soit Λ un ensemble de langages sur A. On définit les prédicats π_Λ^X et π_Λ par :

$$\begin{aligned} \pi_\Lambda^X(\rho) &= \forall \lambda \in \Lambda \,.\, \forall B \in \rho \,.\, (B \subseteq pre_\lambda(X) \;\vee\; B \cap pre_\lambda(X) = \emptyset) \\ \pi_\Lambda(\rho, \rho') &= \bigwedge_{X \in \rho'} \pi_\Lambda^X(\rho) \end{aligned}$$

■

Une partition ρ sera dite *stable* par rapport à la classe X pour l'ensemble de langages Λ si et seulement si $\pi_\Lambda^X(\rho)$ est valide. Par suite, elle sera stable par rapport à une partition ρ' (ou compatible par rapport à ρ') pour Λ si et seulement si $\pi_\Lambda(\rho, \rho')$ est valide. Enfin, il est facile de voir qu'une partition est

compatible avec une relation de transition si et seulement si elle est stable par rapport à elle même (i.e, si et seulement si $\pi_\Lambda(\rho,\rho)$ est valide).

Nous définissons alors les opérateurs $Split_\Lambda$, Ref_Λ et $Refine_\Lambda$ qui permettent, à partir d'une partition donnée ρ, de construire un raffinement de ρ qui soit compatible par rapport à ρ l'ensemble de langages Λ. Cette propriété sera justifiée par la proposition 3.1-3.

Définition 3.1-3
Soit $S=(Q,A,T,q_0)$ un système de transitions étiquetées, soient ρ et ρ' des partitions sur Q, soient X et Y des sous-ensembles de Q et soit Λ un ensemble de langages sur A.

$$\begin{aligned} Split_\Lambda(Y,X) &= \prod_{\lambda\in\Lambda} \{Y\cap T_\lambda^{-1}[X], Y\setminus T_\lambda^{-1}[X]\} \\ Ref_\Lambda(\rho,X) &= \bigcup_{Y\in\rho} Split_\Lambda(Y,X) \\ Refine_\Lambda(\rho,\rho') &= \prod_{X\in\rho'} Ref_\Lambda(\rho,X) \end{aligned}$$

■

Proposition 3.1-3
Soient ρ et ρ' des partitions sur un ensemble Q. L'opérateur Ref_Λ vérifie les propriétés suivantes :

(i) $Ref_\Lambda(\rho,X)\sqsubseteq\rho$

(ii) $\pi_\Lambda^X(\rho)\ -\ \rho = Ref_\Lambda(\rho,X)$

(iii) $\pi_\Lambda^X(Ref_\Lambda(\rho,X))$

De même, l'opérateur $Refine_\Lambda$ vérifie :

(iv) $Refine_\Lambda(\rho,\rho')\sqsubseteq\rho$

(v) $\pi_\Lambda(\rho,\rho')\ -\ \rho = Refine_\Lambda(\rho,\rho')$

(vi) $\pi_\Lambda(Refine_\Lambda(\rho,\rho'),\rho')$

(vii) $\pi_\Lambda^X(\rho)\ \wedge\ X\in\rho'\ \Rightarrow\ Refine_\Lambda(\rho,\rho'-\{X\}) = Refine_\Lambda(\rho,\rho')$

■

Preuve : La preuve d'une proposition similaire peut être trouvée dans [Fer90]. □.

Nous définissons alors, à l'aide de l'opérateur $Refine_\Lambda$, un nouvel opérateur Φ_Λ sur le treillis des partitions, tel que :

$$\Phi_\Lambda(\rho) = Refine_\Lambda(\rho,\rho)$$

L'opérateur Φ_Λ étant monotone et défini sur un treillis complet, il admet pour toute partition initiale ρ_{init} un plus grand point fixe ρ_{Φ_Λ} :

$$\rho_{\Phi_\Lambda} = \nu\rho.\,(\rho_{init}\ \sqcap\ \Phi_\Lambda(\rho))$$

De plus, compte-tenu de la proposition 3.1-3, étant donné une partition initiale ρ_{init}, la partition ρ_{Φ_Λ} est solution du *RPC problem* : Elle correspond en effet au plus grand raffinement de ρ_{init} qui soit compatible avec la relation de transition pour l'ensemble de langages Λ.

La proposition 3.1-1 permet alors de justifier le lien entre l'équivalence de bisimulation et la partition la moins fine compatible avec une relation de transition :

Proposition 3.1-4
Soit $S=(Q,A,T,q_0)$ un système de transitions étiquetées, et soit Λ un ensemble de langages sur

A. Soient $\sim_\Lambda$ et ρ_Λ les plus grands points-fixes des opérateurs $\mathcal{B}_\Lambda$ (*cf.* définition 2.1-1, chapitre 2) et Φ_Λ :

$$\sim^\Lambda \;=\; \nu R.\,((Q \times Q) \cap \mathcal{B}_\Lambda(R))$$
$$\rho_\Lambda \;=\; \nu \rho.\,(\{Q\} \sqcap \Phi_\Lambda(\rho))$$

ρ_Λ est la partition associée à la relation $\sim_\Lambda$. ■

Enfin, dans le cas des systèmes de transitions étiquetées à branchement fini, l'opérateur Φ_Λ est $\sqcap$-continu [Fer90], et son plus grand point-fixe ρ_{Φ_Λ} peut être obtenu en calculant la limite d'une suite décroissante $(\rho_i)_{i\in\mathcal{N}}$ définie par :

- $\rho_0 \;=\; \rho_{init}$
- $\forall i \geq 0,\ \rho_{i+1} \;=\; \Phi_\Lambda(\rho_i)$

De ce résultat, on déduit immédiatement le schéma général d'un algorithme de calcul des classes d'équivalence de la relation $\sim_\Lambda$:

Algorithme 3-1

début
$\rho := \{Q\}$;
$\rho' := \Phi_\Lambda(\rho)$;
tantque $(\rho' \neq \rho)$ **faire**
$\rho := \rho'$;
$\rho' := \Phi_\Lambda(\rho)$;
ftantque
(* ρ contient les classes d'équivalence pour $\sim_\Lambda$ *)
fin.

■

Remarque 3-1
Le fait de choisir comme partition initiale ρ_{init} la partition universelle $\{Q\}$ assure que la relation d'équivalence obtenue est bien la *plus grande* bisimulation définie sur Q. Plus généralement, pour une partition ρ_{init} donnée, l'algorithme 3-1 calcule la plus grande bisimulation contenue dans la relation d'équivalence définie par ρ_{init} : deux états sont alors équivalents si et seulement si ils se bisimulent et s'ils appartiennent à une même classe de ρ_{init}. ■

Avant de décrire de façon plus précise les algorithmes utilisés en pratique dans le cas des relations présentées au chapitre précédent, nous terminons cette section en montrant comment le calcul de la partition associée à une équivalence de bisimulation $\sim_\Lambda$ permet de résoudre le problème de la comparaison et de la minimisation des systèmes de transitions étiquetées.

3.1.1 Comparaison des systèmes de transitions étiquetées

Nous définissons tout d'abord l'*union* de deux systèmes de transitions étiquetées.

Définition 3.1-4
Soient $S_i = (Q_i, A_i, T_i, q_{0i})_{(i=1,2)}$ deux systèmes de transitions étiquetées tels que $Q_1 \cap Q_2 = \emptyset$ (cette condition pouvant toujours être satisfaite moyennant un éventuel renommage). Le système de transitions étiquetées *union* de S_1 et S_2, noté $S_1 \cup S_2$, est défini par :

$$S_1 \cup S_2 = (Q_1 \cup Q_2 \cup \{q_0\}, A_1 \cup A_2, T_1 \cup T_2, q_0)$$

■

La proposition 3.1-5 fournit une *procédure de décision* pour l'équivalence de bisimulation $\sim_\Lambda$: Pour comparer deux systèmes de transitions étiquetées S_1 et S_2 modulo $\sim_\Lambda$, il suffit de vérifier si leur états initiaux appartiennent à la même classe de la plus grande partition compatible avec la relation de transition de $S_1 \cup S_2$.

Proposition 3.1-5
Soient S_1 et S_2 deux systèmes de transitions étiquetées d'états initiaux respectifs q_{01} et q_{02}, et soit $S = (Q, A, T, q_0)$ le système de transition étiqueté union de S_1 et S_2. Soit ρ_Λ la plus grande partition sur Q compatible avec T pour l'ensemble de langages Λ sur $\mathcal{A}$. On a alors :

$$S_1 \sim_\Lambda S_2 \;-\; \exists B \in \rho_\Lambda \,.\, \{q_{01}, q_{02}\} \subseteq B$$

■

Preuve : On pose $S_i = (Q_i, A_i, T_i, q_{0i})_{(i=1,2)}$. D'après la proposition 3.1-4, ρ_Λ est la partition associée à l'équivalence de bisimulation R_Λ définie sur $Q \times Q$, avec $Q = Q_1 \cup Q_2$. Or, Q_1 et Q_2 étant disjoints, les relations T_1 et T_2 le sont également, et par suite :

$$\forall i \in \{1,2\} \,.\, \forall p \in Q_i \,.\, \forall(\lambda, p') \,.\, (p \xrightarrow{\lambda}_T p' \;-\; p \xrightarrow{\lambda}_{T_i} p')$$

En conséquence, il est facile de voir que la relation R_Λ coïncide avec l'équivalence de bisimulation $\sim_\Lambda$ définie sur $Q_1 \times Q_2$. □.

L'algorithme de comparaison s'obtient alors directement à partir de l'algorithme de calcul des classes d'équivalence pour la relation $\sim_\Lambda$ sur un système de transition étiquetée :

Algorithme 3-2

début
  $\rho := \{Q_1 \cup Q_2\}$;
  $\rho' := \Phi_\Lambda(\rho)$;
  tantque $(\rho' \neq \rho)$ et $(\exists B \in \rho' \,.\, \{q_{01}, q_{02}\} \subseteq B)$ **faire**
    $\rho := \rho'$;
    $\rho' := \Phi_\Lambda(\rho)$;
  ftantque
  si $\exists B \in \rho' \,.\, \{q_{01}, q_{02}\} \subseteq B$ **alors**
    afficher $(S_1 \sim_\Lambda S_2)$
  sinon
    afficher $(S_1 \not\sim_\Lambda S_2)$
  fsi.
fin.

■

3.1.2 Minimisation des systèmes de transitions étiquetées

Avant de présenter un algorithme général de minimisation, nous donnons tout d'abord une définition plus précise du *quotient* d'un système de transitions étiquetées modulo une équivalence de bisimulation.

Informellement, le quotient $S/\sim_\Lambda$ d'un système de transitions étiquetées $S = (Q, A, T, q_0)$ modulo une équivalence de bisimulation $\sim_\Lambda$ est défini en fonction des deux contraintes suivantes :

- D'une part, $S/\sim_\Lambda$ doit être le plus petit système de transitions étiquetées, en nombre d'états, qui soit équivalent à S modulo $\sim_\Lambda$.
- D'autre part, $S/\sim_\Lambda$ doit définir une *forme normale* pour S : Pour tout système S' équivalent

à S modulo $\sim_\Lambda$, $S'/\sim_\Lambda$ et $S/\sim_\Lambda$ doivent être *identiques* au nom des états près.

Pour satisfaire la première contrainte, l'ensemble des états de $S/\sim_\Lambda$ sera défini à partir de l'ensemble des classes d'équivalence sur Q pour pour la relation $\sim_\Lambda$, ou, de façon équivalente, à partir de la plus grande partition compatible avec T pour Λ.

Pour obtenir une forme normale (et satisfaire ainsi la seconde contrainte), la relation de transition de $S/\sim_\Lambda$ est construite de la façon suivante : on fixe tout d'abord pour chacun des langages λ_i de Λ un *représentant canonique* $[\lambda_i]$, élément de λ_i, tels qu'à deux langages différents correspondent des représentants différents. On construit alors pour chaque couple (p,q) de Q tel que $p \xrightarrow{\lambda}_T q$ une transition de $S/\sim_\Lambda$ entre les classes d'équivalences de p et de q, étiquetée par $[\lambda]$. Du fait que la partition associée à $\sim_\Lambda$ est compatible avec T pour Λ, une telle relation de transition est toujours définie.

Définition 3.1-5
Soit Λ un ensemble de langages *disjoints* sur $\mathcal{A}$. Pour tout langage λ_i de Λ, on note $[\lambda_i]$ un *représentant canonique* de λ_i. ■

Remarque 3-2

1. Le représentant canonique $[\lambda_i]$ peut en théorie être fixé de façon arbitraire parmi les éléments du langage λ_i. Toutefois, afin d'obtenir un système quotient avec un nombre minimal d'états (définition 3.1-6), il est préférable de choisir un $[\lambda_i]$ de longueur minimale. Nous supposons dans la suite que pour tout les ensembles Λ considérés, il est possible de définir pour chaque λ_i de Λ un représentant canonique $[\lambda_i]$ de longueur 1, c'est à dire élément de $\mathcal{A}_\tau$. Nous verrons que cette hypothèse est justifiée pour toute les équivalences de bisimulation définies au chapitre précédent.

2. Dans le cas particulier de la bisimulation forte $\sim$, on a $\Lambda = \{\{a\} \mid a \in \mathcal{A}_\tau\}$, d'où :

   $$\forall\ \{a\} \in \Lambda,\ [\{a\}] = a$$

■

Définition 3.1-6
Soit $S = (Q, A, T, q_0)$ un système de transitions étiquetées et soit $\sim_\Lambda$ l'équivalence de bisimulation pour l'ensemble de langages Λ, définie sur $Q \times Q$. Le système de transitions étiquetées *quotient* de S, noté $S/\sim_\Lambda$, est défini par :

$$S/\sim_\Lambda = (Q/\sim_\Lambda, A/\sim_\Lambda, T/\sim_\Lambda, [q_0]_{\sim_\Lambda})$$

avec

- $Q/\sim_\Lambda = \{[p]_{\sim_\Lambda} \mid p \in Q\}$,
- $A/\sim_\Lambda = \{[\lambda] \mid \lambda \in \Lambda \ \wedge\ \lambda \subseteq A^*\}$.
- $T/\sim_\Lambda$ est définie par : $\forall p \in Q\ .\ ((p,\lambda,p') \in T\ \Rightarrow\ ([p]_{\sim_\Lambda}, \lambda, [p']_{\sim_\Lambda}) \in T/\sim_\Lambda)$.

■

La proposition suivante justifie que le quotient d'un système de transitions étiquetées S vérifie bien la notion informelle de "plus petit système de transitions étiquetées équivalent à S".

Proposition 3.1-6
Soit S un système de transitions étiquetées et soit $S/\sim_\Lambda$ son quotient par la relation $\sim_\Lambda$. On a alors :

(*i*) $S/\sim_\Lambda\ \sim_\Lambda\ S$

(*ii*) $S/\sim_\Lambda$ est le plus petit système de transitions étiquetées (en nombre d'états) équivalent à S modulo $\sim_\Lambda$.

■

Preuve :

(i) Pour montrer $S/\sim_\Lambda \ \sim_\Lambda \ S$, il suffit de montrer qu'il existe une relation de bisimulation pour l'ensemble de langages Λ entre ces deux systèmes de transitions. Du fait que l'ensemble des classes d'équivalence pour $\sim_\Lambda$ définit une partition compatible sur Q avec T pour Λ (proposition 3.1-4), on a :

$$\forall p \in Q \ . \ (p \xrightarrow{\lambda}_T p' \ - \ [p]_{\sim_\Lambda} \xrightarrow{[\lambda]}_{T/\sim_\Lambda} [p']_{\sim_\Lambda})$$

Par suite, et puisque $[\lambda] \in \lambda$ la relation $\{(p, [p]_{\sim_\Lambda}) \mid p \in Q\}$ est une relation de bisimulation pour Λ entre S et $S/\sim_\Lambda$.

(ii) C'est une conséquence directe du fait que $\sim_\Lambda$ est la plus grande relation de bisimulation sur S, c'est à dire celle qui contient le plus petit nombre de classes.

□.

La proposition suivante justifie que $S/\sim_\Lambda$ est bien une forme normale pour S.

Proposition 3.1-7
Soient $S_i = (Q_i, A_i, T_i, q_{0i})_{(i=1,2)}$ deux systèmes de transitions étiquetées, et soit $\sim_\Lambda$ l'équivalence de bisimulation pour l'ensemble de langages Λ. En notant $=$ la relation d'égalité entre systèmes de transitions (définition 1.2-3, chapitre 1), on a :

$$S_1 \sim_\Lambda S_2 \ - \ S_1/\sim_\Lambda = S_2/\sim_\Lambda$$

■

Preuve :

$\Leftarrow$: La preuve est immédiate par transitivité de $\sim_\Lambda$:

$$S_1/\sim_\Lambda = S_2/\sim_\Lambda \ \Rightarrow \ S_1/\sim_\Lambda \sim_\Lambda S_2/\sim_\Lambda \ \text{ puisque } = \subseteq \sim_\Lambda.$$

D'autre part, d'après la proposition 3.1-6 (i), $S_1/\sim_\Lambda \sim_\Lambda S_1$ et $S_2/\sim_\Lambda \sim_\Lambda S_2$, d'où $S_1 \sim_\Lambda S_2$.

$\Rightarrow$: Soient R_1 et R_2 les équivalences de bisimulation pour l'ensemble de langages Λ définies respectivement sur Q_1^2 et Q_2^2. Les ensembles d'états des systèmes de transitions étiquetées $S_1/\sim_\Lambda$ et $S_2/\sim_\Lambda$ sont alors définies par :

$$Q_1/\sim_\Lambda \ = \ \{[p]_{R_1} \mid p \in Q_1\}$$
$$Q_2/\sim_\Lambda \ = \ \{[p]_{R_2} \mid p \in Q_2\}$$

On vérifie alors que l'on a bien *égalité* entre $S_1/\sim_\Lambda$ et $S_2/\sim_\Lambda$ au sens de la définition 1.2-3 (chapitre 1) :

- Soit $\varphi \ : \ Q_1/\sim_\Lambda \longrightarrow Q_2/\sim_\Lambda$, la fonction définie par :

  $$\varphi([p]_{R_1}) = \{[q]_{R_2} \mid q \in Q_2 \ \wedge \ p \sim_\Lambda q\}$$

  Il est alors facile de voir que φ définit une bijection entre $Q_1/\sim_\Lambda$ et $Q_2/\sim_\Lambda$.
- $S_1 \sim S_2 \ \Rightarrow \ A_1 = A_2$.
- $[p]_{R_1} \xrightarrow{[\lambda]}_{T_1/\sim_\Lambda} [p]_{R_2} \ - \ \varphi([p]_{R_2}) \xrightarrow{[\lambda]}_{T_1/\sim_\Lambda} \varphi([p]_{R_2})$
- $\varphi([q_{01}]_{R_1}) = [q_{02}]_{R_2}$.

□.

Nous donnons alors le schéma général de l'algorithme de calcul du quotient d'un système de transitions étiquetées modulo une équivalence de bisimulation $\sim_\Lambda$. Il consiste en deux phases successives :

1. La construction de son ensemble d'états $Q/\sim_\Lambda$, qui est obtenu en calculant les classes d'équivalence sur Q pour la relation $\sim_\Lambda$ à l'aide de l'algorithme 3-1. On note ρ la partition ainsi obtenue.

2. La construction de sa relation de transition $T/\sim_\Lambda$, obtenue en appliquant l'algorithme suivant :

   Algorithme 3-3

   début
   pour tout $B \in \rho$
   pour tout $p \in B$
   pour tout $(\lambda, q) \in T_\Lambda[p]$
   insérer $(B, [\lambda], [q]_\rho)$ **dans** $T/\sim_\Lambda$
   fpour
   fpour
   fpour
   fin.

■

A partir de ces algorithmes généraux pour la comparaison et de réduction des systèmes de transitions étiquetées modulo une équivalence de bisimulation, nous présentons dans les sections suivantes de façon plus détaillées les solutions généralement mises en œuvre dans les outils "classiques" de vérification. Nous considérons plus particulièrement les relations présentées au chapitre précédent : la bisimulation forte, les bisimulations faibles, la bisimulation de branchement et l'équivalence de sûreté.

3.2 La bisimulation forte

Nous avons vu que dans le cas de la bisimulation forte, l'ensemble des classes d'équivalence sur les états d'un système de transitions étiquetées est obtenu en calculant une partition ρ_Φ, solution du *RCP problem* pour l'ensemble de langages

$$\Lambda_f = \{\{a\} \mid a \in \mathcal{A}_\tau\}.$$

Pour construire une telle partition, un algorithme efficace a été proposé par Paige et Tarjan [PT87], dont le coût en temps est de l'ordre de O(m.log(n)), m et n étant respectivement les nombres d'états et de transitions du système. Nous adoptons ici une présentation de cet algorithme similaire à celle retenue dans [Fer90] : On donne tout d'abord un premier algorithme dont le coût en temps est de l'ordre de $(O(m.n))$, puis on montre comment la réduction logarithmique proposée par Paige et Tarjan s'en déduit.

3.2.1 Un premier algorithme

Soit $S = (Q, A, T, q_0)$ un système de transitions étiquetées. D'après les résultats de la section 3.1, la plus grande partition sur Q, compatible avec la relation de transition T pour un ensemble de langages Λ coïncide avec la partition ρ_Φ, plus grand point-fixe de l'opérateur Φ_Λ, obtenue en calculant la limite de la suite $(\rho_i)_{i\in\mathcal{N}}$ définie par

- $\rho_0 = \rho$
- $\forall i \geq 0,\ \rho_{i+1} = \Phi_\Lambda(\rho_i)$

avec :

$$\begin{aligned} \Phi_\Lambda(\rho) &= Refine_\Lambda(\rho,\rho) \\ Refine_\Lambda(\rho_1,\rho_2) &= \prod_{X\in\rho_2} Ref_\Lambda(\rho_1,X) \end{aligned}$$

La proposition suivante établit une première propriété de l'opérateur $Refine_\Lambda$ sur laquelle est basé l'algorithme de raffinement.

Proposition 3.2-1
Soit ρ une partition définie sur un ensemble Q, et soit $\rho' = Refine_\Lambda(\rho,\rho)$. On a :

$$Refine_\Lambda(\rho',\rho') = Refine_\Lambda(\rho',\rho'-\rho)$$

■

Preuve : On a :

$$\forall X\in\rho_2 \,.\, \pi^X_\Lambda(\rho_1) \Rightarrow Refine_\Lambda(\rho_1,\rho_2) = Refine_\Lambda(\rho_1,\rho_2 - X)$$

Or, d'après la proposition 3.1-3,

$$\forall X\in\rho \,.\, \pi^X_\Lambda(Ref_\Lambda(\rho,X))$$

ce qui entraîne

$$\forall X\in\rho \,.\, \pi^X_\Lambda(Refine_\Lambda(\rho,\rho))$$

Par suite, on a bien $Refine_\Lambda(\rho',\rho') = Refine_\Lambda(\rho',\rho'-\rho)$. □.

D'après la proposition 3.2-1, pour obtenir la partition ρ_{i+1} lors du calcul de ρ_Φ, il n'est pas nécessaire de raffiner ρ_i par rapport à l'ensemble de ses classes : il suffit en pratique de considérer l'ensemble de *partitionneurs* W_i constitué des classes de ρ_i qui ont été obtenues au pas i (i.e, qui n'appartiennent pas à ρ_{i-1}). Nous définissons donc un opérateur Φ'_Λ, qui, à partir d'une partition et d'un ensemble de partitionneurs donné (ρ,W), retourne une nouvelle partition et un nouvel ensemble de partitionneurs.

Définition 3.2-1
Soit ρ une partition sur un ensemble Q, et W un ensemble de partitionneurs ($W\subseteq\rho$).

$$\Phi'_\Lambda(\rho,W) = (\rho',W')$$

avec

$$\begin{aligned} \rho' &= Refine_\Lambda(\rho,W) \\ W' &= Refine_\Lambda(\rho,W) - \rho \end{aligned}$$

■

L'opérateur Φ'_Λ est monotone sur le treillis des partitions et il admet comme plus grand point-fixe le couple $(\rho_\Phi,\emptyset)$ [Fer88]. En conséquence, dans le cas où le système de transitions étiquetées S est à branchement fini, la partition ρ_Φ coïncide avec la limite de la suite $(\rho_i,W_i)_{i\in\mathcal{N}}$ définie par :

- $(\rho_0,W_0) = (\rho,\rho)$
- $\forall i\geq 0,\ (\rho_{i+1},W_{i+1}) = \Phi'_\Lambda(\rho_i,W_i)$

On déduit de cette propriété un premier algorithme dans lequel le calcul des partitions ρ_i et des ensembles W_i est fait de façon simultanée :

début
  $(\rho,W) := (\{Q\},\{Q\})$;
  repeter
    $(\rho,W) := \Phi'(\rho,W)$
  jusqua $W=\emptyset$
fin.

Afin de pouvoir diminuer le nombre de raffinement nécessaires et mettre en œuvre efficacement l'opération $(Refine_\Lambda(\rho, W)$, nous donnons dans la proposition suivante deux propritétés de l'opérateur Ref_Λ, justifiées dans [Fer88] :

Proposition 3.2-2
Soit ρ une partition défini sur un ensemble Q.

(i) $\forall X_i, X_j \in \rho \; . \; Ref_\Lambda(Ref_\Lambda(\rho, X_i), X_j) \; = \; Ref_\Lambda(Ref_\Lambda(\rho, X_j), X_i)$

(ii) Soit $X \in \rho$ tel que $X = \bigcup_i X_i$. On a alors :

$$\prod_i Ref_\Lambda(\rho, X_i) \; \sqcap \; Ref_\Lambda(\rho, X) \; = \; \prod_i Ref_\Lambda(\rho, X_i)$$

■

De la proposition 3.2-2, on déduit les informations suivantes :

- Les raffinements par rapport aux éléments de W peuvent être effectués dans n'importe quel ordre lors du calcul de $(Refine_\Lambda(\rho_i, W_i)$ (d'après (i)).
- Lorsqu'un élément X de W_i (qui appartient donc à ρ_i) est décomposé en un ensemble de classes X_i lors d'un raffinememt, il n'est pas utile de partitionner les autres éléments de ρ_i par rapport à X puisqu'ils seront partitionnés par rapport à chacun des X_i lors du calcul de ρ_{i+1} (d'après (ii)). Par suite, X peut être remplacé par l'ensemble des X_i dans W.
- Dès qu'un élément B de W_i a été selectionné pour partitionner ρ_i il peut être enlevé de l'ensemble W : en effet, que B soit ou non décomposé lors du raffinement, il est inutile de partitionner les éléments de ρ_{i+1} par rapport à B (d'après la proposition 3.2-1 et (ii)).

Plus formellement, nous définissons un nouvel opérateur Φ" qui calcule le couple (ρ', W') défini ci-après :

Définition 3.2-2
Soit ρ une partition sur un ensemble Q, W un ensemble de partitionneurs et B un élément de W.

$$\Phi_\Lambda"(\rho, W, B) = (\rho', W')$$

avec

$$\begin{aligned} \rho' &= Ref_\Lambda(\rho, B) \\ W' &= (Ref_\Lambda(\rho, B) - \rho) \; \cup \; (Ref_\Lambda(\rho, B) \cap W) - \{B\} \end{aligned}$$

■

La partition ρ_Φ est alors obtenue en calculant la limite $(\rho_\Phi, \emptyset)$ de la suite $(\rho_i, W_i)_{i \in \mathcal{N}}$ définie par :

- $(\rho_0, W_0) \; = \; (\rho, \rho)$
- $\forall i \geq 0, \; (\rho_{i+1}, W_{i+1}) \; = \; \Phi"_\Lambda(\rho_i, W_i, B)$ pour B appartenant à W_i.

Ce calcul se traduit alors par l'algorithme suivant :

début
  $(\rho, W) := (\{Q\}, \{Q\})$;
  repeter
    choisir B **dans** W ;
    $(\rho, W) := \Phi"_\Lambda(\rho, W, B)$
  jusqua $W = \emptyset$
fin.

Enfin, il reste à expliciter l'opération $Ref_\Lambda(\rho, B)$. On note $Eff_\Lambda(\rho, B)$ l'ensemble des classes X de ρ pour lesquelles le partitionnement par rapport à B est *effectif*, c'est à dire pour lesquelles

$Split_\Lambda(X,B) \neq X$:

$$Eff_\Lambda(\rho,B) \;=\; \{X \in \rho \mid \exists \lambda \in \Lambda \,.\, X \cap T_\lambda^{-1}(B) = \emptyset \;\wedge\; X \not\subseteq T_\lambda^{-1}(B)\}$$

Algorithme 3-4

début
  $\rho := \{Q\}$; $W := \{Q\}$;
  repeter (1)
    choisir W **dans** B ;
    $W := W$ - $\{B\}$;
    pour tout $X \in Eff_\Lambda(\rho,B)$
      $N := Split_\Lambda(X,B)$;
      $\rho := \rho - \{X\} \cup N$;
      si $X \in W$ **alors**
        remplacer X **par** N **dans** W ;
      sinon
        ajouter N **dans** W ;
      fsi
    fpour
  jusqua $W = \emptyset$
fin.

■

Proposition 3.2-3
Soit $(\rho_i, W_i)_{i\in\mathcal{N}}$ la suite des partitions et des ensembles de partitionneurs obtenues lors de l'exécution de l'algorithme 3-4. On note $nb\_W(q)$ le nombre de fois qu'une classe contenant l'état q est utilisée comme partitionneur :

$$\forall q \in Q \,.\, nb\_W(q) \;=\; |\{B \mid q \in B \;\wedge\; \exists i \,.\, \rho_{i+1} = Ref_\Lambda(\rho_i, B)\}|.$$

L'algorithme 3-4 peut être implémenté avec un coût de l'ordre de

$$O(nb\_W(q).\, \Sigma_{q\in Q}\, |T_\Lambda^{-1}[q]|)$$

en temps, et $O(|T|)$ en mémoire. ■

Preuve : Elle repose sur les arguments suivants (une preuve complète peut être trouvée dans [Fer88]) :

- Le corps de l'itération principale **(1)** peut être réalisé en un coût en temps de $O(\,\Sigma_{q\in B}\, |T_\Lambda^{-1}[q]|)$.
- Le nombre total d'exécutions de l'itération **(1)** est majoré par le nombre maximum de partitionneurs utilisés, soit $|W|$, et par suite, le coût en temps de l'algorithme est de l'ordre de

$$O(\,\Sigma_{B\in W}\;\; \Sigma_{q\in B}\, |T_\Lambda^{-1}[q]|) \;=\; O(nb\_W(q).\, \Sigma_{q\in Q}\, |T_\Lambda^{-1}[q]|).$$

□.

Corollaire 3.2-1
Pour l'ensemble de langages $\Lambda_f = \{\{a\} \mid a \in \mathcal{A}_\tau\}$ l'algorithme 3-4 peut être implémenté avec un coût de $O(m.n)$ en temps et $O(m)$ en mémoire, où $m = |T|$ et $n = |Q|$. ■

Preuve : Il est facile de voir que,

- $\Sigma_{q\in Q}\, |T_{\Lambda_f}^{-1}[q]| = m$,
- et d'autre part la suite des blocs (B_i) contenant q et inserés dans W est strictement décroissante et contient donc au plus n éléments. Par suite, $nb\_W(q) = n$.

□.

3.2.2 La solution de Paige et Tarjan

Afin d'améliorer l'efficacité de l'algorithme 3-4, l'idée proposée par Paige et Tarjan consiste à diminuer le nombre de partitionneurs utilisés en majorant la fonction $nb\_W(q)$ par $log(n)$. Le principe de leur algorithme repose sur la proposition suivante :

Proposition 3.2-4
Soit ρ une partition définie sur un ensemble Q, X et B des classes de ρ telles que X soit *stable* par rapport à B et $B = B_1 \cup B_2$. On a alors :

$$Split_\Lambda(X, B_1) \sqcap Split_\Lambda(X, B_2) = \prod_{\lambda \in \Lambda} \{X_1^\lambda, X_2^\lambda, X_3^\lambda\}$$

avec,

$$\begin{aligned} X_1^\lambda &= (X \cap T_\lambda^{-1}[B_1]) - T_\lambda^{-1}[B_2] \\ X_2^\lambda &= (X \cap T_\lambda^{-1}[B_2]) - T_\lambda^{-1}[B_1] \\ X_3^\lambda &= (X \cap T_\lambda^{-1}[B_1]) \cap T_\lambda^{-1}[B_2] \end{aligned}$$

chacun des X_i^λ pouvant être éventuellement vide. ■

Preuve : Elle est immédiate en remarquant que si X est stable par rapport à B on a :

$$\forall \lambda \in \Lambda \,.\, (X \subseteq T_\lambda^{-1}[B]) \vee (X \cap T_\lambda^{-1}[B] = \emptyset)$$

Par suite X est effectivement partitionnée par rapport à (B_1, B_2) (i.e, au moins deux des X_i^λ sont non vides) uniquement lorsque $X \subseteq T^{-1}[B]$. □.

Avant de montrer comment cette proposition peut s'appliquer lors du calcul de la partition ρ_Φ, nous donnons en premier lieu un algorithme qui permet de calculer les classes X_i^λ avec un coût en temps de l'ordre de la taille *de la plus petite* des deux classes B_1 et B_2. C'est cette propriété qui justifiera par la suite la réduction logarithmique obtenue pour l'algorithme 3-4.

On définit la fonction nb_succ_B, qui, pour tout état p, retourne le nombre de successeurs de p dans la classe B pour l'action étiquetée par λ :

$$nb\_succ_B(p, \lambda) = |T_\lambda[p] \cap B|$$

En supposant que B_1 est la plus petite des deux sous-classes de B ($|B_1| \leq |B_2|$), le calcul des classes X_i^λ pour λ fixé, peut se faire à partir de $nb\_succ_B(p, \lambda)$ en utilisant l'algorithme suivant :

1. On détermine $nb\_succ_{B_1}(p, \lambda)$, ce qui peut se faire en $O(|B_1|)$ en utilisant la relation de transition inverse. On pourra alors déduire $nb\_succ_{B_2}(p, \lambda)$ à l'aide de $nb\_succ_B(p, \lambda)$.

2. On calcule les classe $\{X_1^\lambda, X_2^\lambda, X_3^\lambda\}$ en appliquant pour chaque $p \in B_1$ les règles R_1, R_2 et R_3 suivantes :

$$\frac{nb\_succ_{B_1}(p, \lambda) = nb\_succ_B(p, \lambda)}{X_1 = X_1 \cup \{p\}} \quad [R1]$$

$$\frac{nb\_succ_{B_1}(p, \lambda) = 0}{X_2 = X_2 \cup \{p\}} \quad [R2]$$

$$\frac{0 < nb\_succ_{B_1}(p, \lambda) < nb\_succ_B(p, \lambda)}{X_3 = X_3 \cup \{p\}} \quad [R1]$$

L'ensemble des classes X_i^λ est alors obtenu en itérant cette opération pour chaque élément λ de Λ.

La proposition 3.2-4 permet de diminuer le coût des raffinements lors du calcul de la limite de la suite $(\rho_k, W_k)_{k \in \mathcal{N}}$ définie dans la section précédente à l'aide de l'opérateur Φ'_Λ : En effet, par définition de

Φ'_Λ, chaque fois qu'un élément B de W_k est partitionné en (B_1, B_2) alors B_1 et B_2 sont insérées dans W_{k+1}. Par suite, toutes les classes X de ρ_{k+1} seront partitionnées au pas suivant en $\{X_1^\lambda, X_2^\lambda, X_3^\lambda\}$ comme indiqué dans la proposition 3.2-4. Cette opération peut alors se faire en $O(min(|B_1|, |B_2|)$ au lieu du $O(|B_1| + |B_2|)$ requis par l'algorithme précédent.

Pour pouvoir mettre en œuvre cet algorithme, il est donc nécessaire de mémoriser de quelle façon a été obtenu chaque nouveau partitionneur inclus dans W. La solution proposée par Paige et Tarjan consiste à définir deux sortes de partitionneurs dans W, les *blocs simples* et les *blocs arbres*.

- Un bloc arbre modélise l'ensemble de partitionneurs $\{B_i\}$ obtenus lors du raffinement d'une classe B telle que la partition courante ρ soit stable par rapport à B. Le bloc arbre n associé à B sera donc représenté par un arbre *binaire* dont les feuilles sont les classes B_i qui appartiennent à ρ. Par suite, chaque fois que l'un des B_i sera à son tour partitionné, n sera mis à jour en remplaçant la feuille B_i par le bloc arbre représentant sa décomposition. Raffiner ρ par rapport à n consiste alors à partitionner chaque classe de ρ par rapport à l'union des feuilles du plus petit des deux fils de n conformément à la proposition 3.2-4.

- Un bloc simple X est un élément de W qui n'a pas été obtenu par raffinement d'une classe pour laquelle la partition courante ρ est notoirement stable. Il sera donc représenté par une classe de ρ, comme dans l'algorithme 3-4 où tout les éléments de W étaient considérés comme blocs simples. Le raffinement de ρ par rapport à X se fait alors de façon similaire au raffinements effectués dans cet algorithme (à l'aide de l'opérateur $Ref_\Lambda(\rho, X)$). Nous verrons dans la suite que la présence de blocs simples dans W lorsque l'on applique l'algorithme de Paige et Tarjan se produit uniquement lorsque la partition initiale n'est pas la partition universelle.

On décrit tout d'abord plus formellement l'opération de raffinement d'une partition par rapport à un partitionneur bloc simple ou bloc arbre puis on donne la nouvelle définition obtenue pour l'opérateur Φ_Λ".

Définition 3.2-3
Soit ρ une partition défini sur un ensemble Q. Pour tout partitionneur x (bloc simple ou bloc arbre), on note $Classe\,(x)$ l'ensemble des classes associées à x obtenues de la façon suivante :

- Si x est un bloc simple X, $Classe\,(x) = X$.

- Si x est un blox arbre n, dont les fils gauches et droit sont respectivement n_1 et n_2, alors, $Classe\,(x) = \{B_1, B_2\}$ avec :
  $$B_1 = \cup\{B' \mid B' \text{ est une feuille de } n_1\}$$
  $$B_2 = \cup\{B' \mid B' \text{ est une feuille de } n_2\}$$

Le raffinement de ρ par rapport à x est alors définie par :

$$Ref_\Lambda(\rho, x) = \prod_{B \in Classe\,(x)} Ref_\Lambda(\rho, B)$$

■

Définition 3.2-4
Soit ρ une partition sur un ensemble Q, et W un ensemble de partitionneurs constitué de blocs simples et de blocs arbres. Soit x un élément de W.

$$\Phi_\Lambda"(\rho, W, x) = (\rho', W')$$

avec,

- $\rho' = Ref_\Lambda(\rho, x)$ (*cf.* définition 3.2-3),

- Soit $X \in \rho$. W' est défini par :

1) Si $X \in W$, $X \neq x$, et X n'est pas partitionnée alors X est laissée inchangée dans W.

2) Si $X \in W$ et X est partitionnée en $\{X_i\}$, alors X est remplacée par $\{X_i\}$ dans W.

3) Si $X \notin W$ (ni en tant que bloc simple, ni en tant que feuille d'un bloc arbre), et X est partitionnée en $\{X_i\}$, alors le bloc arbre représentant cette décomposition est inséré dans W.

4) Les blocs arbres n de W sont transformés en blocs arbres n', obtenus à partir de n en remplaçant toutes les feuilles de n correspondant à des classes partitionnées par le bloc arbre représentant ces décompositions.

De plus, si x est un bloc arbre n alors

5) Les fils gauches et droits non feuilles de n sont insérés dans W'.

■

L'algorithme complet est décrit dans [Fer88]. Il consiste principalement à adapter l'algorithme 3-4 au nouvel opérateur Φ_Λ" en maintenant le coût en temps du corps de l'itération principale en

$O(\ \Sigma_{q \in B}\ |T^{-1}[q]|)$.

D'autre part, en reprenant les notations introduites dans la proposition 3.2-3, il est facile de voir que la suite (B_i) des partitionneurs insérés dans W et contenant un état q donné vérifie

$$\forall i\ .\ |B_{i+1}| \leq \frac{|B_i|}{2}$$

Par suite, le nombre d'élément de cette suite est majoré par $log(n)$, d'où $nb\_W(q) \leq log(n)$, ce qui justifie un coût en en temps de $O(m.log(n))$ pour l'algorithme complet.

3.3 Les bisimulations faibles

Les algorithmes décrits dans la section précédente peuvent en théorie s'appliquer à n'importe quelle relation de bisimulation. Néammoins, les complexités en temps indiquées ne sont valides que dans le cas de la bisimulation forte et peuvent devenir exponentielles dans le cas d'autres relations de bisimulation.

Exemple 3-1
Nous considérons la τ^*a-bisimulation (*cf.* chapitre 2, section 2.5.2), qui est l'équivalence de bisimulation obtenue pour le langage $\Lambda = \{\tau^*a \mid a \in \mathcal{A}\}$. D'après la proposition 3.2-3, le coût en temps de l'algorithme 3-4 est de l'ordre de :

$$O(nb\_W(q).\ \Sigma_{q \in Q}\ |T_\Lambda^{-1}[q]|) = O(nb\_W(q).\ \Sigma_{q \in Q}\ \ \Sigma_{a \in \mathcal{A}}\ |T_{\tau^*a}^{-1}[q]|)$$

■

En conséquence, et pour des raisons d'efficacité, une autre solution a été adoptée dans les outils de vérification classiques pour comparer ou minimiser des systèmes de transitions étiquetées par rapport à une bisimulation faible $\sim_\Lambda$. Cette solution est généralement constituée de deux phases :

- La première phase, souvent désignée par le terme de *saturation*, consiste à construire pour chaque système de transitions étiquetées S une *forme normale modulo la bisimulation forte*. Cette "forme normale" est obtenue en remplaçant chaque transition de S, étiquetée par un élément d'un langage λ_i de Λ, par une transition étiquetée par son représentant canonique $[\lambda_i]$ (*cf.* définition 3.1-5).

Du fait qu'il ne s'agisse pas à proprement parler de "formes normales" (puisque deux systèmes équivalents peuvent avoir des formes normales différentes), nous adopterons dans la suite le terme de *forme pré-normale* introduit dans [Fer89], et nous noterons $\mathrm{PNF}_\Lambda(S)$.

- La seconde phase consiste alors à appliquer aux formes pré-normales les algorithmes relatifs à la bisimulation forte décrits dans la section précédente :
  - Calculer le quotient d'un système de transitions étiquetés S pour la relation $\sim_\Lambda$ revient à calculer le quotient de $\mathrm{PNF}_\Lambda(S)$ pour la bisimulation forte.
  - Comparer deux systèmes de transitions étiquetées S_1 et S_2 modulo $\sim_\Lambda$ revient à décider si $\mathrm{PNF}_\Lambda(S_1)$ et $\mathrm{PNF}_\Lambda(S_2)$ se bisimulent fortement.

Nous justifions en premier lieu le principe de cette solution puis nous montrons comment elle peut être mise en œuvre en pratique dans le cas de l'équivalence observationnelle, de la delay bisimulation et de la $\tau^* a$-bisimulation.

3.3.1 Principe général

Définition 3.3-1
Soit $S = (Q, A, T, q_0)$ un système de transitions étiquetées, et soit Λ un ensemble de langages disjoints sur $\mathcal{A}$. La forme pré-normale de S pour l'équivalence de bisimulation $\sim_\Lambda$ est le système de transitions étiquetées

$$\mathrm{PNF}_\Lambda(S) = (Q, A', T', q_0)$$

avec :

- $A' = \{[\lambda] \mid \lambda \in \Lambda\}$.
- $T' = \{(p, [\lambda], q) \mid (p, \lambda, q) \in T\}$.

■

On a la proposition immédiate suivante :

Proposition 3.3-1
Pour tout système de transitions étiquetées,

$$\mathrm{PNF}_\Lambda(S) \sim_\Lambda S$$

■

Il reste à vérifier que cette définition de forme pré-normale satisfait bien les conditions énoncées, et qu'en particulier deux systèmes de transitions étiquetées sont équivalents modulo l'équivalence de bisimulation $\sim_\Lambda$ si et seulement si leurs formes pré-normales se bisimulent fortement.

Lemme 3.3-1
Soient $S = (Q, A, T, q_0)$ un système de transitions étiquetées et ρ une partition définie sur Q. ρ est compatible avec la relation de transition de S pour un ensemble de langages disjoints Λ si et seulement si ρ est compatible avec la relation de transition de $\mathrm{PNF}_\Lambda(S)$ pour l'ensemble de langages $\Lambda_f = \{\{[\lambda]\} \mid \lambda \in \Lambda\}$. ■

Preuve : On pose $\mathrm{PNF}_\Lambda(S) = (Q, A', T', q_0)$. Par définition de la compatibilité, ρ est compatible avec T pour Λ si et seulement si

$$\forall \lambda \in \Lambda \,.\, \forall i, j \in J \,.\, \forall p, q \in B_i \,.\, (T_\lambda[p] \cap B_j \neq \emptyset \;-\; T_\lambda[q] \cap B_j \neq \emptyset)$$

Or, par définition de $\mathrm{PNF}_\Lambda(S)$,

$$\forall p \in Q \,.\, \forall B \in \rho \,.\, \forall \lambda \in \Lambda \,.\, (T_\lambda[p] \cap B \neq \emptyset \;-\; T_{[\lambda]}[p] \cap B \neq \emptyset).$$

Par suite, ρ est compatible avec T pour Λ si et seulement si ρ est compatible avec T' pour Λ_f. □.

Proposition 3.3-2
Soient S, S_1 et S_2 deux systèmes de transitions étiquetés et soit Λ un ensemble de langages disjoints sur $\mathcal{A}$, et soit $\Lambda_f = \{\{[\lambda]\} \mid \lambda \in \Lambda\}$. On a :

(*i*) $S/\sim_\Lambda = \mathrm{PNF}_\Lambda(S)/\sim_{\Lambda_f}$

(*ii*) $S_1 \sim_\Lambda S_2 \;-\; \mathrm{PNF}_\Lambda(S_1) \sim_{\Lambda_f} \mathrm{PNF}_\Lambda(S_2)$

■

Preuve :

(*i*) On pose $S = (Q, A, T, q_0)$ et $\mathrm{PNF}_\Lambda(S) = (Q, A', T', q_0)$.

- D'après la définition de système quotient, $Q/\sim_\Lambda$ est l'ensemble des éléments de la plus grande partition compatible avec T pour Λ. Or, par le lemme 3.3-1, cette partition coïncide avec la plus grande partition compatible avec T' pour Λ_f. Par suite, les ensembles d'états de $S/\sim_\Lambda$ et $\mathrm{PNF}_\Lambda(S)/\sim_{\Lambda_f}$ sont identiques.
- On montre alors que les relations de transitions $T/\sim_\Lambda$ et $T'/\sim_{\Lambda_f}$ sont également identiques :

$$\forall (p,q) \in Q/\sim_\Lambda \;.\; p \xrightarrow{[\lambda]}_{T/\sim_\Lambda} q \quad - \quad p \xrightarrow{\lambda}_T q \;\; p \xrightarrow{[\lambda]}_{T'/\sim_{\Lambda_f}} q$$

(*ii*) On justifie tout d'abord le sens direct. Par la proposition 3.1-5, Si S_1 et S_2 sont équivalents modulo $\sim_\Lambda$ alors il existe une partition ρ sur $Q_1 \cup Q_2$, compatible avec $T_1 \cup T_2$ pour Λ et telle qu'il existe une classe de ρ contenant (q_{01}, q_{02}). Or, d'après le lemme 3.3-1, ρ est également compatible avec $T_1' \cup T_2'$ pour Λ_f. Par suite, il existe une relation de bisimulation entre $\mathrm{PNF}_\Lambda(S_1)$ et $\mathrm{PNF}_\Lambda(S_2)$ pour l'ensemble de langages Λ_f.

La réciproque se justifie par des arguments similaires.

□.

Nous terminons cette section en décrivant de manière plus précise comment le calcul des formes pré-normales est généralemnt mis en œuvre dans le cas des bisimulations faibles présentées au chapitre précédent. Nous donnons pour chacune des ces relations les complexités des algorithmes obtenus, et nous discutons brièvement leur comportement en pratique.

3.3.2 L'équivalence observationnelle

L'équivalence observationnelle est l'équivalence de bisimulation obtenue pour l'ensemble de langages

$$\Lambda_o = \{\tau^*\} \cup \{\tau^* a \tau^* \mid a \in \mathcal{A}\}.$$

Les représentants canoniques des éléments de Λ sont choisis de la façon suivante :

- $[\{\tau^*\}] = \tau$
- $\forall a \in \mathcal{A} \;.\; [\{\tau^* a \tau^*\}] = a$.

Par conséquent, la forme pré-normale d'un système de transitions étiquetées $S = (Q, A, T, q_0)$ est le système $\mathrm{PNF}_o(S) = (Q, A, T', q_0)$, avec T' défini par :

$$T' = \{(p, a, q) \mid a \in A \;\wedge\; p \xrightarrow{\tau^* a \tau^*}_T q\} \cup \{(p, \tau, q) \mid p \xrightarrow{\tau^*}_T q\}.$$

La relation de transition T' peut être calculée en appliquant l'algorithme suivant [Fer88]:

1. On construit tout d'abord la relation T", obtenue à partir de T par *fermeture transitive* de la relation de transition étiquetée par τ :

$$\begin{aligned} T" \quad = \quad & \{(p,\tau,q) \mid p \xrightarrow{\tau^*}_T q\} \cup \\ & \{(p,a,q) \mid a \neq \tau \ \wedge \ p \xrightarrow{a}_T q\}. \end{aligned}$$

2. La relation de transition T' est alors obtenue à partir de T" de la façon suivante :

$$\begin{aligned} T' \quad = \quad & \{(p,\tau,q) \mid p \xrightarrow{\tau}_{T"} q\} \cup \\ & \{(p,a,q) \mid \exists p' \,.\, p \xrightarrow{\tau}_{T"} p' \ \wedge \ p' \xrightarrow{a}_{T"} q\} \cup \\ & \{(p,a,q) \mid \exists q' \,.\, p \xrightarrow{a}_{T"} q' \ \wedge \ q' \xrightarrow{\tau}_{T"} q\}. \end{aligned}$$

Le nombre de transitions de T' pouvant atteindre n^2 dans le cas le plus défavorable, le coût en mémoire de cet algorithme est de $O(m+n^2)$.

On note n_τ le nombre d'états de Q comportant des successeurs par l'action τ :

$$n_\tau = |\{q \in Q \,.\, T_\tau[q] \neq \emptyset\}|.$$

Le coût en temps de l'algorithme de calcul de T' est alors le suivant :

- En utilisant l'algorithme classique de calcul de fermeture transitive [AHU74], T" peut être construite en $O(n_\tau^3)$. Cette complexité peut toutefois être ramenée à $O(n_\tau^{2.376})$ à l'aide d'algorithmes plus sophistiqués [CW87].
- Le calcul de T' à partir de T" peut être réalisé en $O(n.m)$.

Remarque 3-3
Afin de diminuer la valeur de n_τ on peut appliquer un certain nombre de transformations préliminaires à la relation T, qui préservent l'équivalence observationnelle. Des exemples de telles transformations (comme l'élimination des circuits de τ) sont proposés dans [Fer88]. Par ailleurs, d'autres types de transformations permettent également de réduire le nombre de *transitions* du système quotient obtenu après minimisation modulo la bisimulation forte de $\mathrm{PNF}_o(S)$. ■

Toutefois, sur la plupart des systèmes de transitions étiquetées que l'on rencontre en pratique, la valeur de n_τ reste en général très élevée (i.e, de l'ordre de n), et par suite, un nombre important de transitions sont ajoutées à T lors du calcul de T'. En conséquence, l'efficacité de cet algorithme reste limitée, et son application semble reservée au systèmes de transitions étiquetées de petite taille (i.e, de l'ordre de la dizaine de milliers d'états).

3.3.3 La τ^*a-bisimulation

la τ^*a-bismulation est l'équivalence de bisimulation obtenue pour l'ensemble de langages

$$\Lambda_{\tau^*a} = \{\tau^*a \mid a \in \mathcal{A}\}.$$

Par suite, les représentants canoniques des éléments de Λ_{τ^*a} sont définis par

$$\forall a \in \mathcal{A} \,.\, [\{\tau^*a\}] = a,$$

et la forme pré-normale pour la τ^*a-bismulation du système de transitions étiquetées $S = (Q, A, T, q_0)$ est le système $\mathrm{PNF}_{\tau^*a}(S) = (Q, A, T', q_0)$, avec

$$T' = \{(p,a,q) \mid p \xrightarrow{\tau^*a}_T q\}$$

A partir de la définition de T', il est facile de voir que l'ensemble des états de $\mathrm{PNF}_{\tau^*a}(S)$, qui est par définition le sous-ensemble de Q *accessible* depuis q_0 par la relation de transition T', coïncide avec

l'ensemble Q' des états de Q ayant un prédécésseur par une action *visible* :

$$Q' = \{q \in Q \mid \exists a \in \mathcal{A} \,.\, T_a^{-1}[q] \neq \emptyset\}.$$

On en déduit alors immédiatement un algorithme de calcul de la relation T' :

On détermine le sous-ensemble Q' de Q défini ci-dessus.

On calcule la relation T' sur les états de Q' :

$$T' = \{(p, a, q) \mid p \in Q' \ \wedge \ p \xrightarrow{\tau^* a}_T q\}$$

On note n_a le cardinal de Q'. Le nombre de transitions de T' atteint n_a^2 dans le cas le plus défavorable, et par conséquent le coût en mémoire de cet algorithme est de $O(n_a^2)$. D'autre part, l'ensemble Q' peut être déterminé en $O(m)$ en temps (par un parcours de la relation de transition inverse T^{-1}), et le calcul de la relation T' sur Q peut être réalisé en $O(m.n_a)$

Pour les systèmes de transitions étiquetées que l'on rencontre dans le cadre de la vérification, il apparaît que le nombre de transitions visibles m_a est généralement très faible devant le nombre total de transitions. De fait, n_a est la plupart du temps largement inférieur à n (puisque n_a est majoré par m_a). Par conséquent, les complexités en temps et en mémoire de l'algorithme de calcul de formes pré-normales sont bien moindres dans le cas de la τ^*a-bisimulation que dans le cas de l'équivalence observationnelle. Dans la pratique ce gain se répercute sur la taille des exemples que l'on peut traiter en utilisant la τ^*a-bisimulation, qui peuvent alors dépasser la centaine de milliers d'états.

3.3.4 La delay bisimulation

La delay bisimulation est l'équivalence de bisimulation obtenue pour l'ensemble de langages

$$\Lambda_d = \{\tau^*\} \ \cup \ \{\tau^* a \mid a \in \mathcal{A}\}$$

De même que pour l'équivalence observationnelle et la τ^*a-bisimulation, les représentants canoniques des éléments de Λ_d sont :

- $[\{\tau^*\}] = \tau$
- $\forall a \in \mathcal{A} \,.\, [\{\tau^* a\}] = a$,

Par suite, la forme normale pour la delay bismulation du système de transitions étiquetées $S = (Q, A, T, q_0)$ est le système $\mathrm{PNF}_d(S) = (Q, A, T', q_0)$, avec

$$T' = \{(p, a, q) \mid p \xrightarrow{\tau^* a}_T q\} \ \cup \ \{(p, \tau, q) \mid p \xrightarrow{\tau^*}_T q\}.$$

En s'inspirant des résultats obtenus dans le cas des relations précédentes, on déduit un algorithme de calcul de la relation de transition T' :

1. On construit le sous-ensemble Q' de Q défini par :
$$Q' = \{q \in Q \mid \exists a \in \mathcal{A} \,.\, T_a^{-1}[q] \neq \emptyset\}.$$
2. On *ajoute* alors à T les transitions visibles de T' correspondants au τ^*a-transitions de T :
$$T" = T \ \cup \ \{(p, a, q) \mid p \xrightarrow{\tau^* a}_T q\}.$$
3. On termine de construire T' par fermeture transitive de la relation T_τ :
$$T' = T" \ \cup \ \{(p, \tau, q) \mid p \xrightarrow{\tau^*}_T q\}.$$

En reprenant les complexités obtenus pour les algorithmes précédents, le coût en mémoire de cet algorithme est de de $O(m + n^2)$, et le coût en temps des différentes phases sont respectivement en $O(m)$, $O(m.n_a)$ et $O(n_\tau^3)$.

Remarque 3-4
De même que dans le cas de l'équivalence observationnelle, il est possible de définir un certain nombre de transformations supplémentaires pour la relation T, qui préservent la delay bisimulation et qui permettent de réduire le nombre de transitions de $\mathrm{PNF}_d(S)$ ou la valeur de n_τ. ■

3.4 La bisimulation de branchement

Bien que la bisimulation de branchement ne puisse pas être définie en terme de relation de bisimulation, les algorithmes de vérification relatifs à ces équivalences reposent sur des principes similaires. En effet, J.F Groote et F. Vaandrager ont énoncé une variante du *RCP problem* dont la solution coïncide avec la partition associé à la bisimulation de branchement. De même que dans le cas des équivalences de bisimulation, cette partition peut être construite en calculant des raffinements successifs d'une partition initiale jusqu'a obtention d'une partition compatible avec une certaine relation de transition. Nous décrivons ici uniquement le principe de cet algorithme, et en particulier les opérateurs de raffinements et la notion de compatibilité considérée. Une présentation plus détaillée peut-être trouvée dans [GV90].

En premier lieu, nous associons à toute partition ρ définie sur les états d'un système de transitions étiquetées S une relation de transitions T^ρ, construite à partir de la relation de transition de S. Nous montrons alors que ρ est compatible avec T^ρ (au sens usuel du terme, *cf.* définition 3.1-1) si et seulement si la relation d'équivalence associée à ρ est une bisimulation de branchement.

Définition 3.4-1
Soit $S = (Q, a, T, q_0)$ un système de transitions étiquetées, et soit ρ une partition sur Q, telle que $\rho = \{B_i \mid i \in \mathcal{I}\}$. On note T^ρ la relation de transition étiquetée incluse dans $Q \times A \times Q$ définie par :

$\forall B \in \rho \,.\, \forall p \in B \,.$

$$\begin{aligned} \forall a \in \mathcal{A} \,.\, T_a^\rho[p] &= \{q \mid \exists (p_1, \ldots, p_n) \in B \,.\, p \xrightarrow{\tau}_T p_1 \cdots \xrightarrow{\tau}_T p_n \xrightarrow{a}_T q\} \\ T_\tau^\rho[p] &= \{p\} \cup \{q \mid \exists (p_1, \ldots, p_n) \in B \,.\, p \xrightarrow{\tau}_T p_1 \cdots \xrightarrow{\tau}_T p_n \xrightarrow{\tau}_T q\} \end{aligned}$$

■

La proposition 3.4-1 établit le lien entre une bisimulation de branchement et une partition compatible avec la relation T^ρ.

Proposition 3.4-1
Soit $S = (Q, A, T, q_0)$ un système de transitions étiquetées, et soit R une relation d'équivalence sur $Q \times Q$, et soit $\rho = \{B_i \mid i \in \mathcal{I}\}$ la partition associée à R. R est la plus grande bisimulation de branchement si et seulement si ρ est compatible avec la relation T^ρ :

$$\forall (B_i, B_j) \in \rho,\ \forall (p, q) \in B_i,\ \forall a \in A_\tau,\ R \subseteq \mathcal{B}^{br} \ - \ (T_a^\rho[p] \cap B_j \neq \emptyset \ - \ T_a^\rho[q] \cap B_j \neq \emptyset)$$

■

Preuve :

sens '⇒' : L'implication à montrer est logiquement équivalente à :

$\forall (B_i, B_j) \in \rho,\ \forall (p, q) \in B_i,\ \forall a \in A_\tau,$

(i) $(R \subseteq \mathcal{B}^{br} \ \wedge \ T_a^\rho[p] \cap B_j \neq \emptyset) \ \Rightarrow \ T_a^\rho[q] \cap B_j \neq \emptyset$, et

(ii) $(R \subseteq \mathcal{B}^{br} \ \wedge \ T_a^\rho[q] \cap B_j \neq \emptyset) \ \Rightarrow \ T_a^\rho[p] \cap B_j \neq \emptyset$.

On montre uniquement l'implication (i), l'implication (ii) pouvant être justifiée de façon similaire. Soient (p, q) tels que $p \ R \ q$. Par définition de ρ, il existe $B_i \in \rho$ tel que $\{p, q\} \subseteq B_i$. On

a alors :

$$T_a^\rho[p] \cap B_j \neq \emptyset \;\Rightarrow$$

$$(\exists(p', p") \,.\, p \xrightarrow{\tau^*}_T p" \xrightarrow{a}_T p' \;\wedge\; p" \in B_i \;\wedge\; p' \in B_j) \;\vee\; (a = \tau \;\wedge\; B_i = B_j)$$

Or, d'une part :

$$\begin{aligned} R \subseteq \mathcal{B}^{br}(R) \;\wedge\; p" \; R \; q \;\wedge\; p" \xrightarrow{a}_T p' \;\Rightarrow\; & \exists(q_1, \ldots q_n, q') \,.\, q \xrightarrow{\tau^*}_T q_1 \cdots \xrightarrow{\tau^*}_T q_n \xrightarrow{a}_T q' \\ & \wedge (\forall k \;.\; p" \; R \; q_k) \;\wedge\; (p' \; R \; q') \\ \Rightarrow\; & \exists(q_1, \ldots q_n, q') \,.\, q \xrightarrow{\tau^*}_T q_1 \cdots \xrightarrow{\tau^*}_T q_n \xrightarrow{a}_T q' \\ & \wedge (\forall k \;.\; q_k \in B_i) \;\wedge\; (q' \in B_j) \\ \Rightarrow\; & T_a^\rho[q] \cap B_j \neq \emptyset \end{aligned}$$

et, d'autre part, par définition de T^ρ :

$$(a = \tau \;\wedge\; B_i = B_j) \;\Rightarrow\; T_a^\rho[q] \cap B_j \neq \emptyset.$$

sens '$\Leftarrow$' : Soient (p, q) tel que $p \; R \; q$, d'où $\{p, q\} \subseteq B_i$ pour $B_i \in \rho$.
Soit $a \in A_\tau$, soit $B_j \in \rho$ et soit $p' \in B_j$ tel que $p \xrightarrow{a}_T p'$, d'où $T_a^\rho[p] \cap B_j \neq \emptyset$. Or,

$$\begin{aligned} T_a^\rho[p] \cap B_j \neq \emptyset \;\Rightarrow\; & T_a^\rho[q] \cap B_j \neq \emptyset \\ \Rightarrow\; & (\exists(q", q') \,.\, q \xrightarrow{\tau^*}_T q" \xrightarrow{a}_T q' \;\wedge\; q" \in B_i \;\wedge\; q' \in B_j) \\ & \vee (a = \tau \;\wedge\; B_i = B_j) \\ \Rightarrow\; & (\exists(q", q') \,.\, q \xrightarrow{\tau^*}_T q" \xrightarrow{a}_T q' \;\wedge\; (p \; R \; q") \;\wedge\; (p' \; R \; q')) \\ & \vee (a = \tau \;\wedge\; p' \; R \; q) \end{aligned}$$

On montrerait de la même façon :

$$\forall a \in A_\tau \,.\, \forall q' \,.\, q \xrightarrow{a}_T q' \;\Rightarrow\; (\exists(p", p') \,.\, p \xrightarrow{\tau^*}_T p" \xrightarrow{a}_T p' \;\wedge\; (p" \; R \; q) \;\wedge\; (p' \; R \; q')) \vee (a = \tau \;\wedge\; p \; R \; q')$$

Par suite, on a bien $R \subseteq \mathcal{B}^{br}$.

□.

De même que dans le cas de la bisimulation, et d'après la proposition 3.4-1, le calcul des classes d'équivalence pour la bisimulation de branchement se ramène à la recherche de la plus grande partition ρ qui soit compatible avec la relation T^ρ, conformément à l'énoncé suivant :

> Etant donné une partition initiale ρ_{init} définie sur un ensemble Q, une relation de transition sur $Q \times Q$ étiquetée par les élément d'un ensemble A, trouver le plus grand raffinement ρ de ρ_{init} qui soit compatible avec T^ρ.

Cette variante du *RCP problem*, est généralement désigné sous le nom de *Generalized Relational Coarsest Partition with Stuttering problem* (ou *GRCPS problem*) [GV90].

L'algorithme solution proposé par Groote et Vaandrager est similaire dans son principe aux algorithmes relatifs au *RCP problem* : Il consiste à calculer des raffinements successifs (ρ_i) de ρ_{init} tels que pour tout i, ρ_{i+1} soit *stable* par rapport à ρ_i, la partition finale obtenue étant le plus grand raffinement de ρ_{init} *stable* par rapport à lui-même.

Nous suivons une démarche analogue à celle retenue dans le cas des équivalences de bisimulation (section 3.1) : On précise tout d'abord la notion de *stabilité* que l'on considère, puis on définit les opérateurs *Split*, *Ref* et *Refine* qui décrivent l'opération de raffinement. On montre alors que ces opérateurs permettent bien de construire un raffinement de ρ_{init} qui soit stable par rapport à lui-même.

Définition 3.4-2
Soit $S = (Q, A, T, q_0)$ un système de transitions étiquetées, et soient ρ et ρ' des partitions sur Q. On définit les prédicats π^B et π par :

$$\begin{aligned} \pi^B(\rho) &= \forall a \in A_\tau \,.\, \forall B \in \rho \,.\, (X \subseteq (T_a^\rho)^{-1}[B] \ \vee \ X \cap (T_a^\rho)^{-1}[B] = \emptyset) \\ \pi(\rho, \rho') &= \bigwedge_{B \in \rho'} \pi^B(\rho) \end{aligned}$$

On dit alors que la partition ρ est *stable par rapport à la classe* B si et seulement si $\pi^B(\rho)$ est valide, et que ρ est *stable par rapport à la partition* ρ' si et seulement si $\pi(\rho, \rho')$ est valide. ■

D'après la définition précédente, la partition ρ est compatible avec la relation de transition T^ρ si et seulement si elle est stable par rapport à elle même (i.e, $\pi(\rho, \rho)$ est valide). L'opération de raffinement est définie à l'aide des opérateurs suivants ([FM91b]) :

Définition 3.4-3
Soit $S = (Q, A, T, q_0)$ un système de transitions étiquetées, soient ρ et ρ' des partitions sur Q et soient X et Y des sous-ensembles de Q. On définit :

$$\begin{aligned} \forall a \in A_\tau \,.\, \mathcal{F}_a(X, Y) &= \mu Z \,.\, (X \cap T_\tau^{-1}[Z] \cup X \cap T_a^{-1}[Y]) \\ Split(X, Y) &= \prod_{a \in A_\tau} \{\mathcal{F}_a(X, Y), X \setminus \mathcal{F}_a(X, Y)\} \ \wedge \ (X \neq Y) \\ Split(X, X) &= \prod_{a \in A} \{\mathcal{F}_a(X, Y), X \setminus \mathcal{F}_a(X, Y)\} \\ Ref(\rho, B) &= \bigcup_{X \in \rho} Split(X, B) \\ Refine(\rho, \rho') &= \prod_{X \in \rho'} Ref(\rho, X) \end{aligned}$$

■

Remarque 3-5

1. L'opérateur $\mu X \,.\, F(X)$ représente le plus petit point-fixe (par rapport à l'inclusion $\subseteq$) de la fonction F définie de 2^Q vers 2^Q. Par suite, les fonctions T_a^{-1} étant monotones et $\cup$-continues (lorsque $|T_a^{-1}[p]|$ est finie pour tout p), on a :

   $$\forall a \in A_\tau \,.\, \mathcal{F}_a(X, Y) = \bigcup_{i \geq 0} Z_i$$

   avec

   $$\begin{aligned} Z_0 &= X \cap T_a^{-1}[Y] \\ Z_1 &= X \cap T_\tau^{-1}[Z_0] \\ &\vdots \\ Z_{i+1} &= X \cap T_\tau^{-1}[Z_i] \end{aligned}$$

2. La différence entre les définitions de $Split(X, Y)$ pour $X \neq Y$ et de $Split(X, X)$ sera justifiée par la suite (*cf.* proposition 3.4-2).

■

La correction des opérateurs de raffinement est justifiée par la proposition 3.4-2, similaire à la proposition 3.1-3 pour les équivalences de bisimulation, et dont la preuve repose sur le lemme suivant :

Lemme 3.4-1
Soit $S = (Q, A, T, q_0)$ un système de transitions étiquetées, soit ρ une partition sur Q, et soient X

et Y des éléments de ρ.

$$\forall a \in A_\tau \ . \ \mathcal{F}_a(X,Y) \ = \ X \cap (T_a^\rho)^{-1}[Y] \text{ avec } X \neq Y \tag{3.1}$$

$$\forall a \in A \ . \ \mathcal{F}_a(X,X) \ = \ X \cap (T_a^\rho)^{-1}[X] \tag{3.2}$$

■

Preuve :

- On montre tout d'abord la relation 3.1

  sens $\subseteq$: Soit $a \in A_\tau$. En reprenant les notations introduites dans la remarque 3-5, on a :

  $$\mathcal{F}_a(X,Y) = \bigcup_{i \geq 0} Z_i$$

  On montre alors par induction sur i :

  $$\forall i \ . \ Z_i \subseteq X \cap (T_a^\rho)^{-1}[Y].$$

  En effet, d'une part

  $$T_a^{-1}[Y] \subseteq (T_a^\rho)^{-1}[Y] \ \Rightarrow \ Z_0 \subseteq X \cap (T_a^\rho)^{-1}[Y]$$

  et d'autre part,

  $$\begin{aligned} \forall i \ . \ Z_i \subseteq X \cap (T_a^\rho)^{-1}[Y] \quad &\Rightarrow \quad X \cap T_\tau^{-1}[Z_i] \subseteq X \cap (T_a^\rho)^{-1}[Y] \\ &\Rightarrow \quad Z_{i+1} \subseteq X \cap (T_a^\rho)^{-1}[Y] \end{aligned}$$

  sens $\supseteq$: Soit $p \in X \cap (T_a^\rho)^{-1}[Y]$. On a alors,

  $$\exists q \in Y \ . \ p = p_0 \xrightarrow{\tau}_T p_1 \cdots \xrightarrow{\tau}_T p_n \xrightarrow{a}_T q \ \wedge \ \forall i \ . \ p_i \in X$$

  Or, par définition des Z_i :

  $$\forall i, 0 \leq i \leq n \ . \ p_i \in Z_{n-i}.$$

  Par suite, $p \in Z_n$, d'où $p \in \mathcal{F}_a(X,Y)$.

- La relation 3.2 se montre de façon similaire.

□.

Proposition 3.4-2

Soient ρ et ρ' des partitions sur un ensemble Q. L'opérateur Ref vérifie les propriétés suivantes :

(i) $Ref(\rho, B) \sqsubseteq \rho$

(ii) $\pi^B(Ref(\rho, B))$

(iii) $\pi^B(\rho) \ - \ \rho = Ref(\rho, B)$

De même, l'opérateur $Refine$ vérifie :

(iv) $Refine(\rho, \rho') \sqsubseteq \rho$

(v) $\pi(Refine(\rho, \rho'), \rho')$

(vi) $\pi(\rho, \rho') \ - \ \rho = Refine(\rho, \rho')$

■

Preuve :

(i) Par définition de $Ref(\rho, B)$: Toute classe X de ρ est soit dans $Ref(\rho, B)$ soit partitionnée en différentes sous-classes X_i toutes dans $Ref(\rho, B)$.

(ii) Soit $X \in \rho$. D'après le lemme 3.4-1, on a :

$$\begin{aligned} Split(X,B) &= \prod_{a \in A_\tau} \{X \cap (T_a^\rho)^{-1}[B], X \setminus (X \cap (T_a^\rho)^{-1}[B])\} \text{ si } X \neq B \\ Split(B,B) &= \prod_{a \in A_\tau} \{B \cap (T_a^\rho)^{-1}[B], B \setminus (B \cap (T_a^\rho)^{-1}[B])\} \end{aligned}$$

D'autre part, par définition de T^ρ,

$$\forall X \in \rho \,.\, X \subseteq (T_\tau^\rho)^{-1}[X],$$

et conséquent chaque classe de ρ est toujours stable par rapport à elle même vis à vis de l'action τ. Par suite, on a bien $\pi^B(Ref(\rho,B))$ par construction de $Ref(\rho,B)$.

(iii) **sens $\Rightarrow$:** D'après (i), il suffit de montrer $\rho \subseteq Ref(\rho,B)$. Soit X une classe de ρ et a un élément de A_τ tels que $(a \neq \tau) \vee (X \neq Y)$. On a :

$$\pi^B(X) \Rightarrow (X \subseteq (T_a^\rho)^{-1}[B]) \vee (X \cap (T_a^\rho)^{-1}[B] = \emptyset)$$

Or,

$$\begin{aligned} X \subseteq (T_a^\rho)^{-1}[B] &\Rightarrow X = X \cap (T_a^\rho)^{-1}[B] \\ &\Rightarrow X = \mathcal{F}_a(X,B) \wedge X \setminus \mathcal{F}_a(X,B) = \emptyset \end{aligned}$$

et,

$$\begin{aligned} X \cap (T_a^\rho)^{-1}[B] = \emptyset &\Rightarrow X = X \setminus (X \cap (T_a^\rho)^{-1}[B]) \\ &\Rightarrow \mathcal{F}_a(X,B) = \emptyset \wedge X = X \setminus \mathcal{F}_a(X,B). \end{aligned}$$

On en déduit alors que $X \in Ref(\rho,B)$.

sens $\Leftarrow$: C'est une conséquence directe de (ii).

Les propriétées (iv) — (vi) se déduisent alors des propriétés de l'opérateur Ref. □.

Enfin, de même que dans le cas des équivalences de bisimulation, on définit à l'aide de l'opérateur $Refine$ un nouvel opérateur Φ, monotone sur le treillis des partitions, et tel que le plus grand point-fixe ρ_Φ de Φ coïncide avec la partition solution du *GRCPS problem*, ou, de façon équivalente, avec la partition associée à la bisimulation de branchement :

Proposition 3.4-3
Soit $S = (Q, A, T, q_0)$ un système de transitions étiquetées. Soit Φ l'opérateur défini sur l'ensemble des partitions de Q par $\Phi = Refine(\rho,\rho)$. On note ρ_Φ et $\sim_{br}$ les plus grands point fixes des opérateurs Φ et $\mathcal{B}^{br}$ (*cf.* définition 2.4-2) :

$$\begin{aligned} \rho_\Phi &= \nu\rho.\,(\{Q\} \sqcap \Phi_(\rho)) \\ \sim_{br} &= \nu R.\,((Q \times Q) \cap \mathcal{B}^{br}(R)) \end{aligned}$$

ρ_Φ est la partition associée à $\sim_{br}$. ■

D'autre part, lorsque les systèmes de transitions considérés sont à branchement fini, l'opérateur Φ est $\sqcap$-continue et son plus grand point-fixe peut être obtenu en calculant la limite de la suite $(\rho_i)_{i \in \mathcal{N}}$ définie par :

- $\rho_0 = \{Q\}$
- $\forall i \geq 0 \,.\, \rho_{i+1} = \Phi(\rho_i)$

L'algorithme proposé par Groote et Vaandrager permet de calculer la limite de cette suite avec une complexité de l'ordre de $O(m.n)$ en temps et $O(m)$ en mémoire, où m et n représentent respectivement les nombres d'états et de transitions du système. Par conséquent, toujours de façon similaire au cas des équivalences de bisimulation, on peut d'une part définir une *procédure de décision* pour la bisimulation

de branchement (en calculant cette partition sur l'*union* des deux systèmes de transitions étiquetées, *cf.* définition 3.1-4), et d'autre part minimiser un système de transitions étiquetées en construisant son *système quotient*.

En pratique, l'intérêt de cet algorithme par rapport à ceux dédiés au bisimulations faibles est qu'il ne nécessite pas le calcul préalable d'une forme pré-normale, dont le coût est généralement de l'ordre de celui d'une fermeture transitive. Par suite, il est permet de traiter des systèmes de transitions étiquetées de tailles moyenne avec une efficacité supérieure à celle obtenue dans le cas de la τ^*a-bisimulation.

3.5 L'équivalence de sûreté

De même que la bisimulation de branchement, l'équivalence de sûreté $\approx_s$ n'est pas une équivalence de bisimulation. En effet, cette relation a été définie dans le chapitre précédent comme l'équivalence de simulation associée au préordre $\sqsubseteq_s$ ($\approx_s = \sqsubseteq_s \cap (\sqsubseteq_s)^{-1}$), où $\sqsubseteq_s$ est le préordre de simulation obtenu pour l'ensemble de langages $\Lambda_s = \{\tau^* a \mid a \in \mathcal{A}\}$.

Par conséquent, pour comparer modulo l'équivalence de sûreté deux systèmes de transitions étiquetées S_1 et S_2, une solution immédiate consisterait à comparer leurs états initiaux respectifs q_{01} et q_{02} vis-à-vis de la relation $\sqsubseteq_s$: $S_1 \approx_s S_2$ si et seulement si $q_{01} \sqsubseteq_s q_{02}$ et $q_{02} \sqsubseteq_s q_{01}$. A ce jour, deux différents algorithmes ont été proposés pour effectuer ces comparaisons. En fait, il s'agit de procédures de décision pour une relation plus générale, la *pré-bisimulation* [CPS89], dont le préordre $\sqsubseteq_s$ peut être vue comme une instance particulière.

- La première de ces procédures de décision ressemble dans son principe à l'algorithme de comparaison modulo une équivalence de bisimulation : elle consiste à calculer des "raffinements" successifs d'une relation de préordre initiale sur les ensembles d'états des deux systèmes, soit jusqu'a obtenir la relation $\sqsubseteq_s$, soit jusqu'a ce que les états initiaux des deux systèmes ne soient plus reliés. Toutefois, les relations de préordres ayant une structure plus faible que les relations d'équivalence, il est n'est pas possible de les représenter à l'aide de partitions et il est nécessaire d'utiliser des structures de données plus complexes.

  Plus précisément, si Q_1 et Q_2 sont les ensembles d'états des deux systèmes, le préordre R défini sur $Q_1 \times Q_2$ peut être représenté par un ensemble P de $|Q_1|$ *parties* de Q_2 défini par :

  $$P = \{B(p) \mid \forall p \in Q_1 \; . \; B(p) = \{q \mid pRq\}\}.$$

  L'algorithme consiste alors à "raffiner" l'ensemble P_0, pour lequel chacun des $B(p)$ est égal à Q_2, soit jusqu'a *stabilité*, soit jusqu'a ce que q_{02} n'appartienne plus à $B(q_{01})$. En calculant au préalable les formes pré-normales des deux systèmes pour la τ^*a-bisimulation, cet algorithme peut être mis en œuvre avec une complexité de $O(m.n^4)$ en temps et $O(n^2)$ en mémoire [CPS89] (où m et n représentent les nombres d'états et de transitions des formes pré-normales).

- La seconde procédure de décision pour la relation de pre-bisimulation repose sur un algorithme de *model-checking* pour une variante du μ-calcul [CS91] : intuitivement, pour tout couple (p, q) de $Q_1 \times Q_2$, $p \sqsubseteq_s q$ si et seulement si q *satisfait* un ensemble de formules associé à p. La complexité de l'algorithme obtenu est alors de $O(m^2)$ en temps et $O(m)$ en mémoire, là encore après calcul des formes pré-normales pour la τ^*a-bisimulation.

Dans la cas de l'équivalence de sûreté, il existe néanmoins une autre approche qui consiste à s'intéresser *globalement* à la relation d'équivalence $\approx_s$, sans passer par l'intermédiaire du préordre $\sqsubseteq_s$. Cette solution est basée sur le fait que des équivalences de simulation et de bisimulation qui sont définies pour un même ensemble de langages coïncident lorsque les systèmes considérés sont dépourvues de

transitions redondantes (*cf.* définition 2.6-1 et proposition 2.6-2, chapitre 2). On en déduit alors la conséquence suivante :

> Comparer (*resp.* minimiser) des systèmes de transitions étiquetées modulo l'*équivalence de sûreté* revient à comparer (*resp.* minimiser) ces mêmes systèmes *dépourvues de leurs transitions redondantes* modulo la τ^*a*-bisimulation.*

En pratique, l'intérêt de cette solution par rapport aux méthodes précédentes est double : D'une part elle permet de résoudre à la fois le problème de la comparaison et celui de la minimisation, et d'autre part les algorithmes obtenus ont une complexité moindre. Nous en donnons une présentation plus détaillée dans le reste de la section.

De même que dans le cas des bisimulations faibles, on construit pour tout système de transitions étiquetées S une forme pré-normale pour l'équivalence de sûreté, notée $\mathrm{PNF}_s(S)$, qui vérifie les propriétés suivantes :

- Deux systèmes de transitions étiquetées sont équivalents modulo l'équivalence de sûreté si et seulement si leurs formes pré-normales sont équivalente modulo la bisimulation forte ;
- Le quotient d'un système de transitions étiquetées modulo l'équivalence de sûreté est le quotient de sa forme pré-normale modulo la bisimulation forte.

Plus formellement, $\mathrm{PNF}_s(S)$ est définie à l'aide d'un opérateur Red, qui permet de transformer un système de transitions étiquetées donné en système dépourvu de transitions redondantes, de la manière suivante :

Définition 3.5-1
Soit $S = (Q, A, T, q_0)$ un système de transitions étiquetées. On note $Red(S) = (Q, A, T', q_0)$ le système de transitions étiquetées défini par :

$$p \xrightarrow{a}_{T'} p' \;-\; p \xrightarrow{a}_{T} p' \;\wedge\; \forall p''\ .\ (p \xrightarrow{a}_{T} p'' \;\wedge\; p' \neq p'' \;\Rightarrow\; p' \not\sqsubseteq_s p'')$$

La forme pré-normale et le quotient de S pour l'équivalence de sûreté sont respectivement définis par :

- $\mathrm{PNF}_s(S) = Red(\mathrm{PNF}_{\tau^*a}(S))$,
- $S/\approx_s = \mathrm{PNF}_s(S)/\sim$.

■

La proposition suivante justifie ces définitions :

Proposition 3.5-1
Soient S, S_1 et S_2 des systèmes de transitions étiquetées. On a :

(*i*) $\mathrm{PNF}_s(S) \approx_s S$

(*ii*) $S_1 \approx_s S_2 \;-\; \mathrm{PNF}_s(S_1) \sim \mathrm{PNF}_s(S_2)$.

(*iii*) $S/\approx_s$ est le plus petit système de transitions étiquetées équivalent à S modulo $\approx_s$.

(*iv*) $S_1 \approx_s S_2 \;-\; S_1/\approx_s \;=\; S_2/\approx_s$.

■

Preuve :

(*i*) On a $\mathrm{PNF}_{\tau^*a}(S) \approx_{\tau^*a} S$, d'où, puisque $\approx_{\tau^*a} \subseteq \approx_s$, $\mathrm{PNF}_{\tau^*a}(S) \approx_s S$. Il est alors facile de vérifier que l'élimination des transitions redondantes préserve l'équivalence de sûreté.

(*ii*) D'après la proposition 3.3-1, et puisque $\approx_{\tau^*a} \subseteq \approx_s$:

$$S_1 \approx_s S_2 \quad - \quad \mathrm{PNF}_{\tau^*a}(S_1) \approx_s \mathrm{PNF}_{\tau^*a}(S_2)$$
$$- \quad Red(\mathrm{PNF}_{\tau^*a}(S_1)) \approx_s Red(\mathrm{PNF}_{\tau^*a}(S_2))$$

Puisque les relations $\approx_s$ et $\approx_{\tau^*a}$ coïncident sur des systèmes dépourvus de transitions redondantes (proposition 2.6-2, chapitre 2), on en déduit :

$$S_1 \approx_s S_2 \quad - \quad Red(\mathrm{PNF}_{\tau^*a}(S_1)) \approx_{\tau^*a} Red(\mathrm{PNF}_{\tau^*a}(S_2)).$$

Par suite, comme $\mathrm{PNF}_{\tau^*a}(S_1)$ et $\mathrm{PNF}_{\tau^*a}(S_2)$ sont des systèmes τ-*free*, et que les relations $\approx_{\tau^*a}$ et $\sim$ coïncident pour de tels systèmes (proposition 2.6-8, chapitre 2), on a :

$$S_1 \approx_s S_2 \quad - \quad Red(\mathrm{PNF}_{\tau^*a}(S_1)) \sim Red(\mathrm{PNF}_{\tau^*a}(S_2))$$
$$- \quad \mathrm{PNF}_s(S_1) \sim \mathrm{PNF}_s(S_2)$$

(*iii*) **et** (*iv*) On a $\mathrm{PNF}_s(S) \approx_s S$. La preuve de ces propriétés est alors immédiate en remarquant que $\mathrm{PNF}_s(S)$ est τ-*free* et dépourvu de transitions redondantes, et que par conséquent ce système admet des quotients identiques pour les relations $\approx_s$ et $\sim$.

□.

Nous terminons en présentant brièvement l'algorithme de calcul de la forme pré-normale d'un système de transitions étiquetées $S = (Q, A, T, q_0)$ pour l'équivalence de sûreté, dont une présentation plus détaillée peut être trouvée dans [Mou89] :

- Le calcul de la forme pré-normale pour la τ^*a-bisimulation a été décrit dans la section 3.3.3. Sa complexité est de $O(n_a.m)$ en temps et $O(m)$ en mémoire où est m est le nombre de transitions du système initial S et n_a le nombre de ses états accessibles par une action visible.
- L'élimination des transitions redondantes (opérateur Red) repose alors sur les deux étapes suivantes :
  - On calcule tout d'abord la relation $\sqsubseteq_s$ sur $\mathrm{PNF}_\Lambda(S)$, en tenant compte du fait qu'il s'agit d'un système τ-*free* (ce qui simplifie la définition de la relation). Ce calcul peut être mis en œuvre en construisant une matrice booléenne M sur $Q \times Q$ telle que :

    $$\forall (p,q) \in Q \times Q \ . \ M(p,q) = 1 \ - \ p \sqsubseteq_s q$$

    Le coût de cet algorithme est de $O(m'^2)$ en temps et $O(n'^2)$ en mémoire, où m' et n' représentent les nombres d'états et de transitions de $\mathrm{PNF}_\Lambda(S)$. Par suite, il est souvent préférable en pratique de minimiser au préalable $\mathrm{PNF}_\Lambda(S)$ modulo la bisimulation forte.
  - On peut alors construire le système $Red(\mathrm{PNF}_\Lambda(S))$ à l'aide de la matrice M par un simple parcours de $\mathrm{PNF}_\Lambda(S)$, soit un coût en temps de $O(m')$.

3.6 Discussion

Nous avons rappelé dans ce chapitre le principe des algorithmes classiques de comparaison et de minimisation des systèmes de transitions étiquetées pour chacune des relations d'équivalence ou de préordre qui ont été présentées au chapitre précédent.

Dans le cas des équivalences de bisimulation $\sim_\Lambda$, tous ces algorithmes sont basées sur une même approche, qui consiste en deux phases :

- Le calcul d'une forme pré-normale, obtenue en modifiant les relations de transition des systèmes initiaux, et qui est fonction de la relation d'équivalence considérée ;

- La construction des classes d'équivalence pour la bisimulation forte sur cette forme pré-normale, obtenues en itérant jusqu'à stabilité un calcul de raffinements successifs d'une partition initiale définie sur l'ensemble de ses états. En pratique, cette seconde phase peut être mise en œuvre de manière efficace en utilisant un algorithme de raffinements dû à Paige & Tarjan.

La partition finale qui est construite en appliquant cet algorithme contient alors les classes d'équivalence pour la relation $\sim_\Lambda$ sur les systèmes de transitions étiquetées initiaux. Par suite, à partir de cette partition, il est possible de générer des systèmes quotients ou de décider si deux systèmes de transitions étiquetés donnés sont ou non équivalents.

Dans le cas de la bisimulation de branchement, l'utilisation d'un algorithme de raffinement particulier (et plus coûteux que l'algorithme de Paige & Tarjan) permet d'éviter l'étape intermédiaire du calcul des formes pré-normales. Par conséquent, les classes d'équivalence pour cette relation sont directement construites à partir des systèmes de transitions étiquetées initiaux.

Enfin, si des solutions similaires existent en théorie pour comparer des systèmes modulo un préordre de simulation, il n'est pas possible de mettre en œuvre des algorithmes de raffinement aussi efficaces, en particulier du fait de la structure plus faible de ce type de relations par rapport à celle d'une relation d'équivalence.

A l'heure actuelle, l'ensemble de ces algorithmes ont déjà fait l'objet d'une implémentation au sein d'un ou plusieurs outils de vérification, et il est ainsi possible d'une part de comparer leurs comportements en pratique de manière objective, et d'autre part de déceler leurs principales limitations lorsqu'ils sont appliqués à des systèmes de transitions étiquetées de grandes tailles. D'une façon générale, on aboutit alors aux constat suivant :

- Tout d'abord, la construction des formes pré-normales, basée sur des calculs de fermeture transitive, s'avère très coûteuse en temps et en mémoire, et augmente largement le cardinal de la relation de transition du système que l'on considère. Ainsi, dans le cas de l'équivalence observationnelle, elle limite généralement la taille des systèmes pouvant être traités sur une station de travail à quelques dizaines de milliers d'états (quelques centaines de milliers pour l'équivalence de sûreté ou la bisimulation de branchement).

- De plus, si les algorithmes de raffinements utilisés dans la seconde phase restent très efficace tant que la mémoire vive disponible est suffisante (i.e, pour des systèmes de quelques centaines de miliers d'états), ils deviennent rapidement inopérants dès que la taille des systèmes augmente davantage (même si la mémoire virtuelle de la machine est importante). En effet, le principe même de ces algorithmes requiert un accès aléatoire aux relations de transition des systèmes, ce qui impose en pratique de les mémoriser dans leur ensemble.

Chapitre 4

Diagnostic

La conception des programmes parallèles étant une activité intrinsèquement difficile, il parait indispensable que le processus de vérification permette non seulement d'assurer que le fonctionnement du programme proposé est correct, mais fournisse également le cas échéant une aide à la localisation des erreurs. Plus précisément, lorsqu'une implémentation donnée ne satisfait pas ses spécifications, il est souhaitable de pouvoir en expliquer la cause en proposant un *diagnostic*, exprimé dans un formalisme approprié en fonction de la méthode de vérification mise en œuvre. Du point de vue pratique, l'approche choisie pour le diagnostic doit également respecter un certain nombre de contraintes supplémentaires :

- D'une part, le formalisme utilisé doit permettre de fournir des renseignements les plus précis possibles non seulement sur les *causes* des erreurs, mais aussi sur leur *localisation* dans le programme source.
- D'autre part, en fonction de ce formalisme, la taille du diagnostic engendré doit être telle que celui-ci reste toujours exploitable par l'utilisateur.
- Enfin, le calcul du diagnostic ne doit pas augmenter de façon exagérée les complexités des algorithmes mis en œuvre pour la vérification, et en particulier il ne doit pas restreindre la taille des programmes qui peuvent être traités.

Dans le cadre de la vérification de spécifications comportementales, nous nous intéressons dans ce chapitre aux méthodes de diagnostic qui permettent d'expliquer la non-équivalence entre deux systèmes de transitions étiquetées. Après un bref rappel de l'une des approches existantes, qui consiste à exprimer la non-équivalence à l'aide de formules de *logique temporelle*, nous présentons une solution basée sur un formalisme qui nous semble plus approprié en pratique et qui basé sur les *séquences d'exécutions* : dans le cas des équivalences de bisimulation, on construit un couple de séquences d'exécutions qui, partant des états initiaux des deux systèmes, conduisent, par des actions observables identiques, à un couple d'états à partir desquels les ensembles d'actions qu'il est possible d'effectuer diffèrent (et où la non-équivalence apparaît donc clairement). Par conséquent, ce type de séquences permet bien, d'une part de remettre en cause l'existence d'une bisimulation entre les deux systèmes, et d'autre part, de localiser l'erreur commise.

Nous considérons tout d'abord le cas des équivalences de bisimulation, en définissant la notion de *séquences diagnostiques minimales* et en proposant un algorithme efficace de calcul, puis nous montrons comment cette solution peut également être étendue aux préordres et équivalences de simulation.

4.1 Diagnostic en termes de formules logiques

Une première approche pour exprimer la non-équivalence entre deux systèmes de transitions étiquetées consiste à utiliser une logique temporelle qui soit *adéquate* pour la relation d'équivalence considérée : chaque classe d'équivalence est *caractérisée* par un ensemble de formules particulières de cette logique, ce qui implique que deux états sont équivalents si et seulement si ils satisfont exactement les mêmes formules. Par conséquent, il est possible d'exprimer la non-équivalence entre deux états donnés en exhibant une formule qui soit satisfaite uniquement par l'un d'entre-eux.

Une méthode de diagnostic basée sur ce principe a été proposée par Cleaveland [Cle90] dans le cas de la bisimulation forte à l'aide de la logique HML (Hennessy-Milner Logic), qui est adéquate pour cette relation [HM85]. Plus précisément, la sémantique de HML est obtenue en associant à chaque formule Φ, définie par rapport à un système de transitions étiquetées $S = (Q, A, T, q_0)$, l'ensemble $[\![\Phi]\!]$ des éléments de Q qui satisfont Φ. Pour tout état q, élément de Q, on note alors $H(q)$ l'ensemble des formules satisfaites par q :

$$H(q) = \{\Phi \mid q \in [\![\Phi]\!]\}.$$

Par suite, l'adéquation de la logique HML avec la bisimulation forte peut s'énoncer de la façon suivante :

$$\forall q_1, q_2 \in Q \,.\, q_1 \sim q_2 \;-\; H(q_1) = H(q_2)$$

On définit alors les formules exprimant la non-bisimulation entre les états de deux systèmes de transitions étiquetées :

Définition 4.1-1
Soient S_1 et S_2 deux systèmes de transitions étiquetées et soit $S = (Q, A, T, q_0)$ leur système de transitions étiquetées union ($S = S_1 \cup S_2$). Soient B_1 et B_2 des parties de Q ($B_1 \subseteq Q$, $B_2 \subseteq Q$). Une formule HML Φ *distingue* (modulo la bisimulation forte) B_1 de B_2 si et seulement si :

- $B_1 \subseteq [\![\Phi]\!]$
- $B_2 \cap [\![\Phi]\!] = \emptyset$

■

La méthode de diagnostic proposée par Cleaveland permet de générer ces formules diagnostiques à partir de la procédure de décision classique pour la bisimulation forte en appliquant les deux phases suivantes :

- On calcule tout d'abord la plus grande bisimulation sur l'union des deux systèmes de transitions étiquetées en appliquant une variante de l'algorithme de raffinement de Kanellakis et Smolka (*cf.* chapitre 3, section 3.1). La différence avec l'algorithme original consiste à maintenir des informations supplémentaires sur la façon dont ont été construit les éléments des partitions successives obtenues lors du raffinement, ce qui peut se faire sans modifier la complexité de cet algorithme, qui est de $O(m)$ en mémoire et $O(m.n)$ en temps (n et m désignant les nombres d'états et de transitions du système union).

- Lorsque les deux systèmes ne se bisimulent pas, c'est à dire lorsque leurs états initiaux q_1 et q_2 appartiennent à des classes différentes B_1 et B_2 de la partition finale, on génère à l'aide des informations obtenues pendant la première phase une formule HML qui *distingue* B_1 de B_2 (définition 4.1-1). Bien que ce principe de construction assure que les formules obtenues soient *minimales* (dans le sens où elle ne contiennent pas de sous-formules inutiles [Cle90]), leur taille peut devenir exponentielle par rapport au nombre d'états des systèmes. Il est toutefois possible d'en calculer une représentation polynomiale moyennant un traitement supplémentaire. Les coûts en temps et en mémoire de cette deuxième phase sont alors respectivement

de $O(n.p^2.log(p))$ et $O(p^2)$, où p désigne le nombre de classes d'équivalence de la plus grande bisimulation sur le système union ($p \leq n$).

Cette méthode de diagnostic peut également être étendue au cas des autres équivalences de bisimulation en comparant modulo la bisimulation forte les états des formes pré-normales des systèmes de transition originaux. (*cf.* chapitre 3, section 3.3). Enfin, une méthode identique pour la bisimulation de branchement est présentée dans [Kor91a], dans laquelle les formules diagnostiques sont exprimées à l'aide de la logique HMLU, qui est une extension de HML adéquate pour cette équivalence [NV90]. De même que dans le cas de la bisimulation forte, les formules sont construites à partir de la procédure de décision classique qui repose également sur un algorithme de raffinement de partitions. Les coûts en temps et en mémoire du nouvel algorithme sont alors similaires à ceux obtenus pour la bisimulation forte.

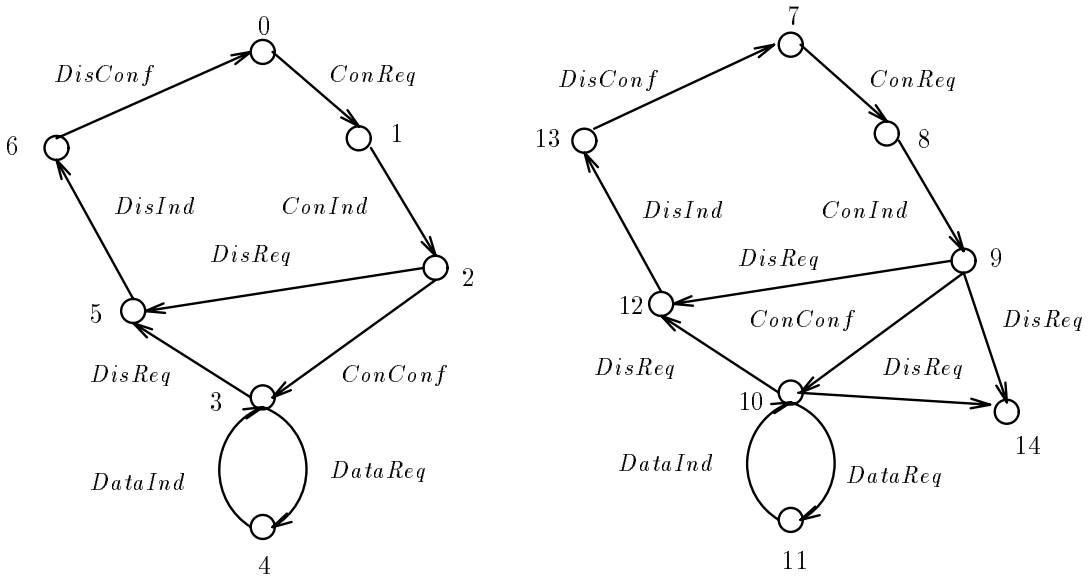
Bien que cette approche soit élégante et qu'elle présente un intérêt certain sur le plan théorique, elle ne nous paraît pas résoudre entièrement le problème de la génération de diagnostics tel qu'il se pose en pratique, en particulier du point de vue de l'aide à la localisation des erreurs. En effet, si les logiques temporelles s'avèrent bien adaptées en tant que langage de spécification, il semble préférable de représenter les diagnostics à l'aide d'un formalisme moins abstrait, et surtout moins éloigné des formalismes utilisés pour représenter le programme source. Enfin, cette méthode de diagnostic n'ayant pas encore été implémentée au sein d'un outil de vérification, il est difficile d'apprécier le comportement des algorithmes mis en œuvre dans le cas d'exemples réalistes, particulièrement en ce qui concerne la taille des formules engendrées.

4.2 Diagnostic en termes de séquences d'exécution

Nous proposons dans cette section une seconde approche qui consiste exprimer le diagnostic en termes de séquences d'exécutions sur les systèmes de transitions. A l'origine, cette solution est inspirée de la méthode de diagnostic utilisée pour la vérification de spécifications exprimées à l'aide de la logique temporelle LTAC, et qui a donné lieu à l'outil CLÉO([Ras90, Ras91]), intégré à l'évaluateur de formules de XESAR [GRRV89].

Intuitivement, le principe de CLÉO consiste à expliquer pourquoi un programme, représenté par un système de transitions étiquetées S, ne satisfait pas une formule LTAC appartenant à sa spécification en construisant un ensemble de séquences d'exécution de S qui contredisent cette formule. L'examen de ces séquences indique alors à l'utilisateur de façon précise quelles sont les suites d'actions que peut effectuer le programme et qui ne satisfont pas sa spécification, permettant ainsi de retrouver l'origine de l'erreur dans le programme source. De manière analogue, il est possible d'exprimer la non-équivalence entre deux systèmes de transitions étiquetées S_1 et S_2 (représentant respectivement un programme et sa spécification comportementale) en construisant un ensemble de séquences d'exécutions de ces deux systèmes qui contredit la notion d'équivalence entre les deux systèmes : On exhibe ainsi d'une part les comportements du programme qui ne sont pas admis par sa spécification, et d'autre part les comportements décrits par la spécification et qui ne sont pas implémentés dans le programme.

Exemple 4-1
Nous reprenons l'exemple du protocole de transport présenté dans le chapitre 2 (exemple 2-3). Sur la figure suivante, le système de transitions étiquetées S_1 décrit le service attendu du protocole, alors que S_2 donne une représentation erronée de ce service dans laquelle les *demandes de déconnexion* ($DisReq$) peuvent conduire de manière non-déterministes à un blocage (dans l'état 14) :



S_1 S_2

Les couples suivants de séquences d'exécutions de S_1 et S_2 montrent de façon explicite que les comportements de ces deux systèmes diffèrent : il existe une séquence d'actions observables qui peut être effectuée par les deux systèmes et qui conduit nécessairement à des états à partir desquels seul l'un des deux systèmes peut évoluer par une action observable donnée (l'action $DisInd$ sur l'exemple) :

$$\begin{cases} 0 \xrightarrow{ConReq} 1 \xrightarrow{ConInd} 2 \xrightarrow{DisReq} 5 \\ 7 \xrightarrow{ConReq} 8 \xrightarrow{ConInd} 9 \xrightarrow{DisReq} 14 \end{cases}$$

$$\begin{cases} 0 \xrightarrow{ConReq} 1 \xrightarrow{ConInd} 2 \xrightarrow{ConConf} 3 \xrightarrow{DisReq} 5 \\ 7 \xrightarrow{ConReq} 8 \xrightarrow{ConInd} 9 \xrightarrow{ConConf} 10 \xrightarrow{DisReq} 14 \end{cases}$$

$$\begin{cases} 0 \xrightarrow{ConReq} 1 \xrightarrow{ConInd} 2 \xrightarrow{ConConf} 3 \xrightarrow{DataReq} 4 \xrightarrow{DataInd} 3 \xrightarrow{DisReq} 5 \\ 7 \xrightarrow{ConReq} 8 \xrightarrow{ConInd} 9 \xrightarrow{ConConf} 10 \xrightarrow{DataReq} 11 \xrightarrow{DataInd} 10 \xrightarrow{DisReq} 14 \end{cases}$$

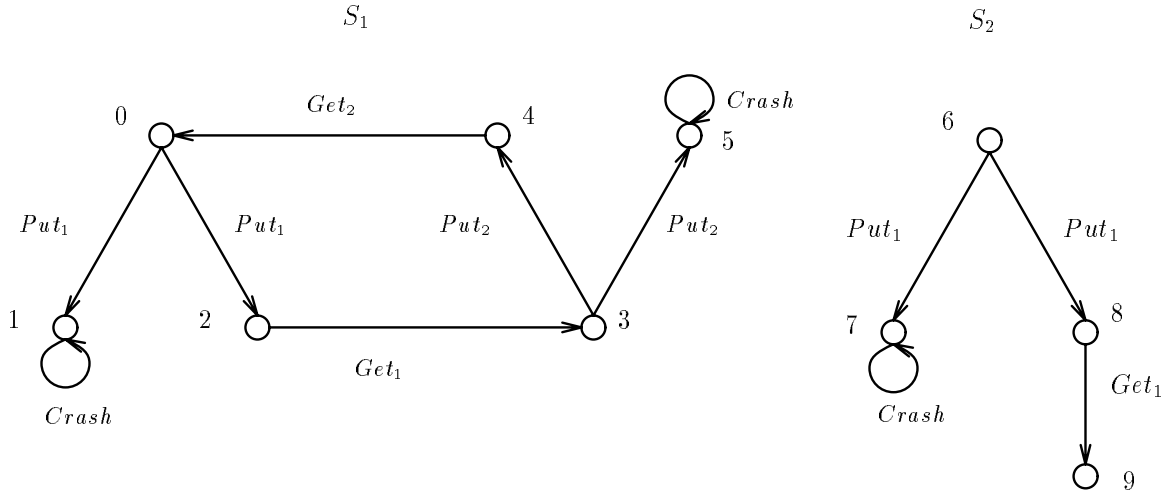
■

Exemple 4-2

On considère dans cet exemple une abstraction d'un protocole de diffusion, similaire à celui présenté au chapitre 2 (exemple 2-2), et dans lequel un émetteur E *non fiable*, envoie séquentiellement des messages à deux stations réceptrices R_1 et R_2, qui sont elles supposées *fiables*. On utilise les actions observables Put_1 et Put_2 pour représenter les émission de messages vers les stations R_1 et R_2, Get_1 et Get_2 pour représenter les réceptions effectuées par ces dernières, et $Crash$ pour représenter les pannes de l'émetteur.

Sur la figure suivante,

- le système de transitions S_1 décrit une spécification correcte de ce protocole, dans laquelle, en l'absence de pannes, les messages sont toujours émis par E et reçus par R_1 et R_2,
- et le système de transitions S_2 en décrit une implémentation erronée (où seule les actions observables apparaissent) dans laquelle l'émetteur se bloque après l'émission du premier message vers R_1.



Les couples de séquences d'exécutions de S_1 et S_2 qui montrent que ces deux systèmes ne sont pas équivalents modulo la bisimulation forte sont alors :

$$\begin{cases} 0 \xrightarrow{Put_1} 1 \xrightarrow{Get_1} 2 \\ 6 \xrightarrow{Put_1} 8 \xrightarrow{Get_1} 8 \end{cases}$$

$$\begin{cases} 0 \xrightarrow{Put_1} 2 \\ 6 \xrightarrow{Put_1} 7 \end{cases}$$

$$\begin{cases} 0 \xrightarrow{Put_1} 1 \\ 6 \xrightarrow{Put_1} 8 \end{cases}$$

■

Remarque 4-1

Sur l'exemple 4-2, les deux derniers couples de séquences ne sont pas strictement nécessaires pour expliquer la non-bisimulation entre S_1 et S_2, puisque 1 et 7 se bisimulent, et seuls 2 et 8 ne se bisimulent pas (le premier couple de séquences serait donc à priori suffisant).

Toutefois, nous avons *choisi* de prendre en compte ce type de séquences lors du diagnostic, choix qui n'a pas posé de problèmes en pratique du point de vue de l'utilisateur. Notons que ce cas de figure se produit uniquement lorsque *les deux* systèmes de transitions que l'on compare sont non déterministes.

■

Les sections suivantes présentent de façon plus formelle la notion de *séquences diagnostiques* pour la non-équivalence de deux systèmes de transitions étiquetées. On donne tout d'abord une définition générale, que l'on justifie, puis on propose différents critères pour obtenir des séquences diagnostiques *minimales*.

4.3 Séquences diagnostiques pour la bisimulation

Nous définissons dans un premier temps un *produit synchrone* $S_1 \times_\Lambda S_2$ entre deux systèmes de transitions étiquetées S_1 et S_2 modulo un ensemble d'actions observables. Nous montrons alors comment construire des séquences diagnostiques pour la non-équivalence entre S_1 et S_2 à partir d'un sous-ensemble des séquences d'exécution de $S_1 \times_\Lambda S_2$.

4.3.1 Un produit synchrone

Définition 4.3-1
Soient $S_i = (Q_i, A_i, T_i, q_{0i})_{(i=1,2)}$ deux systèmes de transitions étiquetées, et soit Λ un ensemble de langages sur A. Le produit synchrone de S_1 et S_2 pour Λ, noté $S_1 \times_\Lambda S_2$, est le système de transitions étiquetées $S = (Q, A_1 \cap A_2, T, (q_{01}, q_{02}))$, où $Q \subseteq Q_1 \times Q_2$ et $T \subseteq T_1 \times T_2$ sont définis par la règle suivante :

$$\frac{(q_1, q_2) \in Q,\ \lambda \in \Lambda,\ q_1 \xrightarrow{\lambda}_{T_1} q_1',\ q_2 \xrightarrow{\lambda}_{T_2} q_2'}{(q_1', q_2') \in Q,\ (q_1, q_2) \xrightarrow{\lambda}_{T} (q_1', q_2')}$$

■

La proposition suivante justifie le fait qu'à chaque séquence d'exécution de $S_1 \times_\Lambda S_2$ peut être associé un couple de séquences d'exécution de S_1 et de S_2 définies sur les mêmes actions observables :

Proposition 4.3-1
Soient $S_i = (Q_i, A_i, T_i, q_{0i})_{(i=1,2)}$ deux systèmes de transitions étiquetées et soit $S = (Q, A, T, q_0)$ leur produit synchrone ($S = S_1 \times_\Lambda S_2$). On a alors :

$$\forall (p_0, q_0) \in Q\ .\ ((p_0, q_0), \lambda_0, (p_1, q_1), \ldots, \lambda_{k-1}, (p_k, q_k)) \in Ex_\Lambda((p_0, q_0)) \ -$$
$$(p_0, \lambda_0, p_1, \ldots, \lambda_{k-1}, p_k) \in Ex(p_0) \ \wedge\ (q_0, \lambda_0, q_1, \ldots, \lambda_{k-1}, q_k) \in Ex(q_0)$$

■

Preuve : Par induction sur la longueur des séquences.

□.

4.3.2 Définition des séquences diagnostiques

On considère les systèmes de transitions étiquetées S_1 et S_2 d'états initiaux respectifs q_{01} et q_{02}, et l'équivalence de bisimulation $\sim_\Lambda$ définie sur un ensemble de langages Λ.

D'après la définition de relation de bisimulation entre systèmes de transitions étiquetées, S_1 et S_2 ne sont pas équivalents modulo $\sim_\Lambda$ si et seulement si $q_{01} \not\sim_\Lambda q_{02}$, c'est à dire si et seulement si il existe un entier k tel que $q_{01} \not\sim^k_\Lambda q_{02}$. Par conséquent, expliquer que deux systèmes ne se bisimulent pas revient à expliquer pourquoi leurs états initiaux ne sont pas équivalent modulo une relation $\sim^k_\Lambda$. La proposition suivante explicite la relation $\not\sim^k_\Lambda$ à partir de laquelle seront définies les séquences diagnostiques :

Proposition 4.3-2

Soient $S_i = (Q_1, A_i, T_i, q_{0i})_{(i=1,2)}$ deux systèmes de transitions étiquetées, et soient $\sim_\Lambda$ une équivalence de bisimulation définie sur un ensemble de langages Λ. Pour tout $(q_1, q_2) \in Q_1 \times Q_2$, on a :

(i) $q_1 \not\sim^1_\Lambda q_2 \;-\; act_\Lambda(q_1) \neq act_\Lambda(q_2)$

(ii) $\forall k > 1 \,.\, q_1 \not\sim^k_\Lambda q_2 \;-$

$$(\exists \lambda \in \Lambda \,.\, \exists q'_1 \,.\, (q_1 \xrightarrow{\lambda}_{T_1} q'_1 \;\wedge\; \forall q"_2 \,.\, q_2 \xrightarrow{\lambda}_{T_2} q"_2 \;\Rightarrow\; q'_1 \not\sim^{k-1}_\Lambda q"_2))$$
$$\vee$$
$$(\exists \lambda \in \Lambda \,.\, \exists q'_2 \,.\, (q_2 \xrightarrow{\lambda}_{T_2} q'_2 \;\wedge\; \forall q"_1 \,.\, q_1 \xrightarrow{\lambda}_{T_1} q"_1 \;\Rightarrow\; q"_1 \not\sim^{k-1}_\Lambda q'_2))$$

(iii) $\forall k > 1 \,.\, q_1 \not\sim^k_\Lambda q_2 \;\wedge\; q_1 \sim^{k-1}_\Lambda q_2 \;\Rightarrow$

$$\exists \lambda \,.\, \exists (q'_1, q'_2) \,.\, q_1 \xrightarrow{\lambda}_{T_1} q'_1 \;\wedge\; q_2 \xrightarrow{\lambda}_{T_2} q'_2 \;\wedge\; q'_1 \not\sim^{k-1}_\Lambda q'_2 \;\wedge\; q'_1 \sim^{k-2}_\Lambda q'_2$$

■

Preuve : Ce sont des conséquences immédiates de la définition de $\sim^k_\Lambda$ (*cf.* chapitre 2).
□.

Les séquences diagnostiques seront alors construites de la façon suivante :

- Si $k = 1$, expliquer que $q_1 \not\sim^1_\Lambda q_2$ revient à exhiber l'ensemble des λ de Λ pour lesquels seul q_1 ou q_2 admet un successeur (proposition 4.3-2 (i)).
- Pour $k > 1$, d'après la proposition 4.3-2 (ii) $q_1 \not\sim^k_\Lambda q_2$ si et seulement si il existe des successeurs q'_1 de q_1 (*resp.* q'_2 de q_2) par une action observable λ tels qu'il n'y ait pas de successeurs de q_2 (*resp.* de q_1) par la même action λ qui leur soit équivalents modulo $\sim^{k-1}_\Lambda$.

Plus formellement, ces états sont définis à partir des ensembles suivants :

Définition 4.3-2
Soient $S_i = (Q_i, A_i, T_i, q_{0i})_{(i=1,2)}$ deux systèmes de transitions étiquetées et soit l'équivalence de bisimulation $\sim_\Lambda$. Pour tout couple (q_1, q_2) de $Q_1 \times Q_2$, et pour tout $k > 1$, on considère les ensembles suivants :

$$\mathrm{Fail1}^k_\lambda(q_1, q_2) \;=\; \{q'_1 \mid q_1 \xrightarrow{\lambda}_{T_1} q'_1 \;\wedge\; \forall q"_2 \,.\, (q_2 \xrightarrow{\lambda}_{T_2} q"_2 \;\Rightarrow\; q'_1 \not\sim^{k-1}_\Lambda q"_2)\}$$
$$\mathrm{Fail2}^k_\lambda(q_1, q_2) \;=\; \{q'_2 \mid q_2 \xrightarrow{\lambda}_{T_2} q'_2 \;\wedge\; \forall q"_1 \,.\, (q_1 \xrightarrow{\lambda}_{T_1} q"_1 \;\Rightarrow\; q"_1 \not\sim^{k-1}_\Lambda q'_2)\}$$

et

$$\mathrm{Fail}^k_\lambda(q_1, q_2) = (\mathrm{Fail1}^k_\lambda(q_1, q_2) \;\times\; T_{2_\lambda}[q_2]) \;\cup\; (T_{1_\lambda}[q_1] \;\times\; \mathrm{Fail2}^k_\lambda(q_1, q_2))$$

■

A l'aide des notations introduites dans la définition 4.3-2, expliquer pourquoi $q_1 \not\sim^k_\Lambda q_2$ revient à montrer que l'union des ensembles $\mathrm{Fail1}^k_\lambda(q_1, q_2)$ et $\mathrm{Fail2}^k_\lambda(q_1, q_2)$ est non vide, ou encore que l'ensemble $\mathrm{Fail}^k_\lambda(q_1, q_2)$ est non vide.

Plus précisément, pour tout état (q_1, q_2), l'ensemble $\mathrm{Fail}^k_\lambda(q_1, q_2)$ satisfait les propriétés suivantes :

Lemme 4.3-1
Soient $Si = (Q_i, A_i, T_i, q_{0i})_{(i=1,2)}$ deux systèmes de transitions étiquetées, et soit $S = (Q, A, T, q_0)$ leur produit synchrone. Pour tout état (q_1, q_2) de Q, on a :

(i) $\forall k > 1 \,.\, \forall \lambda \in \Lambda \,.\, \mathrm{Fail}^k_\lambda(q_1, q_2) \;\subseteq\; T_\lambda[(q_1, q_2)]$

(ii) $\forall k > 1 \,.\, q_1 \not\sim^k_\Lambda q_2 \;\Rightarrow\; (\forall (q'_1, q'_2) \,.\, (q'_1, q'_2) \in \mathrm{Fail}^k_\lambda(q_1, q_2) \;\Rightarrow\; q'_1 \not\sim^{k-1}_\Lambda q'_2)$

(iii) $\forall k > 1 \,.\, q_1 \not\sim^k_\Lambda q_2 \;\wedge\; q_1 \sim^1_\Lambda q_2 \;-\; \exists \lambda \in \Lambda \,.\, \mathrm{Fail}^k_\lambda(q_1, q_2) \neq \emptyset$

■

Preuve :

(i) **et** (ii) sont des conséquences directes de la définition de Fail_λ^k.

(iii) **sens '⇒' :** On a, par définition de Fail_λ^k :

$$q_1 \not\sim_\Lambda^k q_2 \;\Rightarrow\; \exists \lambda \in \Lambda \,.\, \text{Fail1}_\lambda^k(q_1,q_2) \cup \text{Fail2}_\lambda^k(q_1,q_2) \neq \emptyset.$$

D'autre part, $\text{Fail1}_\lambda^k(q_1,q_2) \subseteq T_{1_\lambda}[q_1]$, $\text{Fail2}_\lambda^k(q_1,q_2) \subseteq T_{2_\lambda}[q_2]$, et

$$q_1 \sim_\Lambda^1 q_2 \;\Rightarrow\; (\forall \lambda \in \Lambda \,.\, T_{1_\lambda} \neq \emptyset \;-\; T_{2_\lambda} \neq \emptyset).$$

On déduit alors

$$q_1 \not\sim_\Lambda^k q_2 \;\wedge\; q_1 \sim_\Lambda^1 q_2 \;\Rightarrow\; (\text{Fail1}_\lambda^k(q_1,q_2) \times T_{2_\lambda}[q_2]) \;\cup\; (T_{1_\lambda}[q_1] \times \text{Fail2}_\lambda^k(q_1,q_2)) \neq \emptyset.$$

sens '⇐' : immédiat, à partir de la proposition 4.3-2 (ii) et de la définition de Fail_λ^k.

□.

Le lemme 4.3-1 justifie alors l'existence de séquences d'exécution du produit synchrone de deux systèmes de transitions étiquetées S_1 et S_2 qui permettent d'*expliquer* la non-bisimulation entre les états initiaux q_{01} et q_{02} de ces deux systèmes, construite en appliquant l'algorithme informel suivant :

1. Si $q_{01} \not\sim_\Lambda^1 q_{02}$ alors les ensembles d'actions que peuvent effectuer ces deux états sont différents, et la non-bisimulation apparaît clairement.

2. Dans le cas contraire, il existe un entier $k > 1$ tel que $q_{01} \not\sim_\Lambda^k q_{02}$. Par conséquent, il existe un successeur (q_1, q_2) de (q_{01}, q_{02}) dans $S_1 \times_\Lambda S_2$ qui appartient à $\text{Fail}_\lambda^k(q_1,q_2)$ (lemme 4.3-1, (i) et (iii)). Or, nécessairement, $q_1 \not\sim_\Lambda^{k-1} q_2$ (lemme 4.3-1, (ii)), et ce processus peut ainsi être répété récursivement à partir de (q_1, q_2) jusqu'à obtention d'une séquence d'exécution de $S_1 \times_\Lambda S_2$ terminé par un couple d'états non-équivalents modulo $\sim_\Lambda^1$.

L'existence de ces séquences sera justifiée plus formellement dans la section 4.3.3.

Définition 4.3-3
Soient S_1 et S_2 deux systèmes de transitions étiquetées, soit $S = (Q, A, T, q)$ leur produit synchrone, et soit $\sim_\Lambda$ une équivalence de bisimulation.

Pour tout entier $k > 1$, et pour tout $(p_0, q_0) \in Q$ tel que $p_0 \not\sim_\Lambda^k q_0$, l'ensemble $Diag_\Lambda{}^k$ est défini par :

$$\begin{aligned} Diag_\Lambda{}^k(p_0,q_0) \;=\; & \{\sigma \in Ex_\Lambda(p_0,q_0) \mid \sigma = ((p_0,q_0), \lambda_0, (p_1,q_1), \ldots, \lambda_{k-2}, (p_{k-1},q_{k-1})) \\ & \wedge\, \forall i,\; 0 \leq i < k-1, \exists n_i \,.\, 1 \leq n_i \leq k \,.\, (p_{i+1},q_{i+1}) \in \text{Fail}_{\lambda_i}^{n_i}(p_i,q_i) \\ & \wedge\, p_{k-1} \not\sim_\Lambda^1 q_{k-1}\} \end{aligned}$$

■

Ces séquences diagnostiques pour la non équivalence entre états de S_1 et S_2 sont alors étendues aux systèmes de transitions étiquetées eux-mêmes :

Définition 4.3-4
Soient S_1 et S_2 deux systèmes de transitions étiquetées d'états initiaux respectifs q_{01} et q_{02}, et soit $\sim_\Lambda$ une équivalence de bisimulation. L'ensemble $Diag_\Lambda$ des séquences diagnostiques pour la non-équivalence modulo $\sim_\Lambda$ entre S_1 et S_2 est défini par :

$$Diag_\Lambda(S_1,S_2) = \bigcup_{k>1} Diag_\Lambda{}^k(q_{01},q_{02})$$

■

Remarque 4-2

- Intuitivement, les états *terminaux* des séquences diagnostiques de $Diag_\Lambda(S_1, S_2)$ correspondent aux états du produit synchrone à partir desquels les comportements des systèmes S_1 et S_2 divergent (en termes d'actions observables pouvant être effectuées). Par conséquent, l'ensemble de ces états représente bien des "erreurs" qui sont présentes dans S_1 ou S_2 et qui remettent en cause l'existence d'une équivalence de bisimulation entre ces deux systèmes. Dans la suite, nous désignerons ces états sous le terme d'*états erreurs*.

- Le choix de considérer ou non comme séquences diagnostiques les séquences dues au non-déterminisme (*cf.* remarque 4-1) dépend en fait uniquement de la manière dont est défini l'ensemble Fail^k_λ (définition 4.3-2).

  Plus précisément, la définition suivante permettrait de ne pas prendre en compte ce type de séquences dans la définition de $Diag_\Lambda(S_1, S_2)$:

$$\begin{aligned}\mathrm{Fail}^k_\lambda(q_1,q_2) \quad = \quad & (\mathrm{Fail1}^k_\lambda(q_1,q_2) \times (T_{2_\lambda}[q_2] - \mathrm{Fail2}^k_\lambda(q_1,q_2))) \cup \\ & ((T_{1_\lambda}[q_1] - \mathrm{Fail1}^k_\lambda(q_1,q_2)) \times \mathrm{Fail2}^k_\lambda(q_1,q_2))\end{aligned}$$

  L'ensemble des résultats établis dans ce chapitre resterait alors valide avec cette définition de Fail^k_λ.

■

4.3.3 Existence des séquences diagnostiques

Pour justifier les définitions 4.3-3 et 4.3-4, il suffit de montrer que, pour tout couple (p_0, q_0), $p_0 \not\sim_\Lambda q_0$ si et seulement si l'ensemble des séquences diagnostiques est non vide :

Proposition 4.3-3
Soient S_1 et S_2, deux systèmes de transitions étiquetées, $S = S_1 \times_\Lambda S_2$, et $\sim_\Lambda$ une équivalence de bisimulation.

Pour tout état (p_0, q_0) de S, on a :

$$\forall k > 1 \,.\, p_0 \not\sim^k_\Lambda q_0 \;-\; {Diag_\Lambda}^k(p_0,q_0) \neq \emptyset$$

■

Preuve :

sens '⇒' : On montre alors la proposition par induction sur k :

Pour $k = 2$, d'après le lemme 4.3-1 (iii),

$$p_0 \not\sim^2_\Lambda q_0 \;\Rightarrow\; \exists (p_1,q_1) \,.\, (p_1,q_1) \in \mathrm{Fail}^2_\lambda(p_0,q_0) \;\wedge\; p_1 \sim^0_\Lambda q_1.$$

d'autre part, $(p_1, q_1) \in T_\lambda[(p_0, q_0)]$ (lemme 4.3-1 (i)), et par suite :

$$((p_0,q_0),\lambda,(p_1,q_1)) \in Ex_\Lambda(p_0,q_0).$$

On suppose alors la proposition vérifiée pour tout m inférieur à un entier k fixé avec $k > 2$. On a alors :

$$\begin{aligned} p_0 \not\sim^k_\Lambda q_0 \;\wedge\; p_0 \sim^{k-1}_\Lambda q_0 \quad & \Rightarrow \quad \exists (p_1,q_1) \,.\, (p_1,q_1) \in \mathrm{Fail}^k_\lambda(p_0,q_0) \quad \text{(lemme 4.3-1 } (iii)) \\ & \Rightarrow \quad p_1 \not\sim^{k-1}_\Lambda q_1 \quad \text{(lemme 4.3-1 } (ii))\end{aligned}$$

Par suite, d'après l'hypothèse d'induction, il existe une séquence σ' de $Ex_\Lambda(p_1, q_1)$ qui appartient à ${Diag_\Lambda}^{k-1}(p_1, q_1)$. D'autre part, puisque $(p_1, q_1) \in T_\lambda[(p_0, q_0)]$ (lemme 4.3-1, (i)),

$$\sigma = ((p_0, q_0), \lambda, \sigma') \in Ex_\Lambda(p_0, q_0)$$

et par conséquent :

$$\sigma \in {Diag_\Lambda}^{k-1}(p_0, q_0).$$

sens '$\Leftarrow$' : Par définition de ${Diag_\Lambda}^{k}(p_0, q_0)$, $(p_1, q_1) \in \mathrm{Fail}^{n_0}_{\lambda_0}(p_0, q_0)$, avec $1 < n_0 \leq k$. Or,

$$\mathrm{Fail}^{n_0}_{\lambda_0}(p_0, q_0) \neq \emptyset \quad \Rightarrow \quad p_0 \not\sim^{n_0}_\Lambda q_0$$
$$\Rightarrow \quad p_0 \not\sim^{k}_\Lambda q_0.$$

□.

Le corollaire suivant justifie alors les définitions 4.3-3 et 4.3-4 :

Corollaire 4.3-1
Soient S_1 et S_2, deux systèmes de transitions étiquetées, $S = S_1 \times_\Lambda S_2$, et $\sim_\Lambda$ une équivalence de bisimulation. Pour tout état (p_0, q_0) de S, on a :

$$(p_0 \not\sim_\Lambda q_0 \ \wedge \ act_\Lambda(p_0) = act_\Lambda(q_0)) \ - \ \exists k \ . \ {Diag_\Lambda}^{k}(p_0, q_0) \neq \emptyset$$

■

4.4 Des séquences diagnostiques minimales

Bien que la définition de séquences diagnostiques proposée dans la section 4.3 soit satisfaisante du point de vue théorique (dans le sens où chaque élément de $Diag_\Lambda(S_1, S_2)$ permet bien d'expliquer pourquoi les systèmes de transitions étiquetées S_1 et S_2 ne sont pas équivalents modulo $\sim_\Lambda$), elle présente toutefois un certain nombre d'inconvénients en pratique, essentiellement en raison du *nombre* et de la *longueur* des séquences qui sont prises en compte. En effet :

- D'une part, lorsque le système produit $S_1 \times_\Lambda S_2$ contient des circuits, l'ensemble de ses séquences d'exécution est *infini*, et il peut par conséquent en être de même pour $Diag_\Lambda(S_1, S_2)$ (*cf.* exemple 4-1). Un tel ensemble n'est donc pas calculable en pratique.

- D'autre part, certaines séquences de $Diag_\Lambda(S_1, S_2)$ peuvent contenir des arguments qui ne sont pas strictement nécessaires à l'explication de la non-équivalence. D'une façon générale, ces séquences *redondantes* peuvent être minorées sans perte d'information par des séquences plus courtes et par conséquent plus facilement exploitables par l'utilisateur. Enfin, lorsque le nombre de séquences diagnostiques engendrées est très élevé, il peut être préférable de ne construire que les séquences les plus courtes qui conduisent à un état erreur donné.

Afin d'obtenir une définition de séquences diagnostiques qui soit plus intéressante en pratique, nous proposons dans cette section un certain nombre de critères permettant de construire des *séquences minimales*. Plus précisément, on montre dans un premier temps qu'il est toujours possible de se restreindre à des séquences *élémentaires*, ce qui permet d'obtenir un ensemble de séquences diagnostique toujours fini. On propose alors différentes relations d'ordre sur l'ensemble des séquences d'exécution d'un système de transitions étiquetées et on discute de leurs intérêts pour la définition de séquences diagnostiques minimales.

4.4.1 Séquences diagnostiques élémentaires

Les propositions suivantes justifient le fait que lorsque deux systèmes de transitions étiquetées ne sont pas équivalents il est toujours possible de construire des séquences diagnostiques élémentaires, c'est à dire des séquences de $S_1 \times_\Lambda S_2$ qui ne contiennent pas deux fois le même état (*cf.* définition 1.2-6, chapitre 1).

Proposition 4.4-1
Soient S_1 et S_2 deux systèmes de transitions étiquetées, soit $\sim_\Lambda$ une équivalence de bisimulation, et soit σ une séquence de $Diag_\Lambda(S_1, S_2)$ vérifiant :

- $\sigma = ((p_0, q_0), \lambda_0, (p_1, q_1), \ldots, \lambda_{k-2}, (p_{k-1}, q_{k-1}))$
- $\forall i,\ 0 \leq i < k-1 \ .\ p_i \not\sim_\Lambda^{k-i} q_i \ \wedge\ p_i \sim_\Lambda^{k-i-1} q_i$

On a alors :

$$\forall (i,j)\ .\ i \neq j \ \Rightarrow\ \sigma(i) \neq \sigma(j).$$

■

Preuve : Soient (i,j) tels que $0 \leq i < j \leq k-1$. D'une part, on a, par définition de σ,

$$p_j \not\sim_\Lambda^{k-j} q_j \ \text{ et } \ p_i \sim_\Lambda^{k-i-1} q_i,$$

et, d'autre part,

$$\begin{aligned} i < j \quad &\Rightarrow \quad k - i > k - j \\ &\Rightarrow \quad k - i - 1 \geq k - j \end{aligned}$$

Par conséquent, (p_i, q_i) et (p_j, q_j) ne peuvent être identiques. □.

Corollaire 4.4-1
Soient S_1 et S_2 deux systèmes de transitions étiquetées d'états initiaux respectifs p_0 et q_0, et soit $\sim_\Lambda$ une équivalence de bisimulation.

$$p_0 \not\sim_\Lambda q_0 \ \Rightarrow\ \exists \sigma \in Diag_\Lambda(S_1, S_2)\ .\ \forall (i,j)\ .\ i \neq j \ \Rightarrow\ \sigma(i) \neq \sigma(j)$$

■

Preuve : On a tout d'abord :

$$p_0 \not\sim_\Lambda q_0 \ \Rightarrow\ -\ \ \exists k\ .\ p_0 \not\sim_\Lambda^k q_0 \ \wedge\ p_0 \sim_\Lambda^{k-1} q_0$$

Par suite, en utilisant la proposition 4.3-2 (*iii*), on peut montrer par induction qu'il existe une séquence σ de $Diag_\Lambda(S_1, S_2)$ telle que :

$$\forall i\ .\ 0 \leq i < |\sigma| - 1 \ \wedge\ \sigma(i+1) = (p_i, q_i) \ \Rightarrow\ p_i \not\sim_\Lambda^{k-i} q_i \ \wedge\ p_i \sim_\Lambda^{k-i-1} q_i$$

σ satisfait donc les conditions énoncées dans la proposition 4.4-1, et, par conséquent, σ est bien une séquence élémentaire. □.

4.4.2 Relations d'ordre sur les séquences d'exécution

Plusieurs relations d'ordre peuvent être définies sur l'ensemble des séquences d'exécution d'un système de transitions étiquetées S. Pour tout ensemble quelconque E de séquences d'exécution, nous considérons dans la suite trois relations particulières qui donneront lieu à trois notions de séquences diagnostiques minimales :

- La relation $<_l$ prend en compte la longueur des séquences : une séquence σ de E sera minimale pour cette relation si et seulement si E ne contient pas de séquence plus courte (en nombre d'états) que σ.

- La relation $<_p$ est la relation classique de *préfixe* : une séquence σ de E sera minimale pour cette relation si et seulement si E ne contient aucun des préfixes propres de σ.
- La relation $<_\cap$ prend en compte la longueur des séquences et l'ensemble des états qu'elles définissent : une séquence σ de E sera minimale pour cette relation si et seulement si E ne contient pas de séquence σ' plus courte que σ et telle que σ et σ' comportent des états communs autres que l'état initial p_0 de S.

Définition 4.4-1
Soit S un système de transitions étiquetées, soit p_0 un état de S, et soient σ et σ' deux séquences d'exécution de $Ex(p_0)$ telles que :

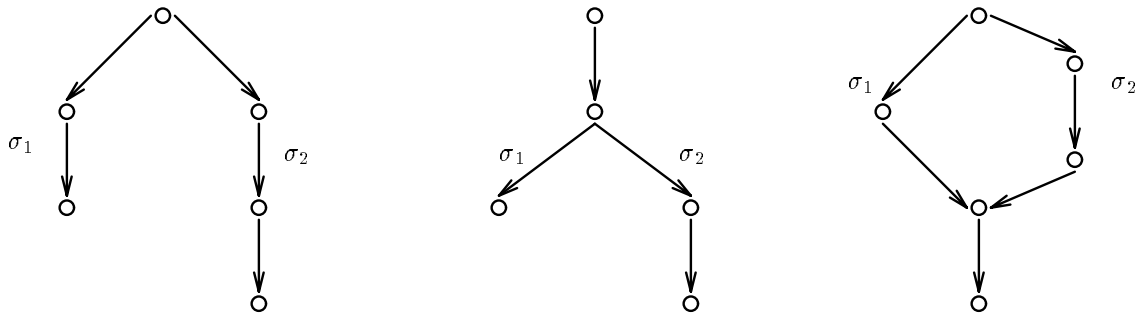
$$\sigma = (p_0, a_0, p_1, \ldots a_{n-1}, p_n)$$
$$\sigma' = (p_0, a'_0, p'_1, \ldots a'_{n'-1}, p'_{n'}).$$

On munit alors l'ensemble $Ex(p_0)$ des relations $<_l$, $<_p$ et $<_\cap$ définis de la façon suivante :

- $\sigma' <_l \sigma \; - \; n' < n$
- $\sigma' <_p \sigma \; - \; (n' < n) \wedge (\forall i,\ 0 \leq i \leq n' \,.\, a_i = a'_i \wedge p_i = p'_i)$
- $\sigma' <_\cap \sigma \; - \; (n' < n) \wedge (\exists i, i',\ 0 < i' \leq n', 0 < i \leq n \,.\, p_i = p'_{i'})$

■

Exemple 4-3



S_1 S_2 S_3

Sur les systèmes de transitions de la figure précédente, les ensembles de séquences minimales pour les relations $<_l$, $< p$ et $<_\cap$ (respectivement notés $M^{<_l}$, $M^{<_p}$ et $M^{<_\cap}$) sont données dans le tableau suivant :

système	$M^{<_l}$	$M^{<_p}$	$M^{<_\cap}$
S_1	$\{\sigma_1\}$	$\{\sigma_1, \sigma_2\}$	$\{\sigma_1, \sigma_2\}$
S_2	$\{\sigma_1\}$	$\{\sigma_1, \sigma_2\}$	$\{\sigma_1\}$
S_3	$\{\sigma_1\}$	$\{\sigma_1, \sigma_2\}$	$\{\sigma_1\}$

■

Pour toute relation d'ordre $<$ définie sur l'ensemble des séquences d'exécution de $S_1 \times_\Lambda S_2$, on définit l'ensemble des séquences diagnostiques minimales modulo $<$ de la manière suivante :

Définition 4.4-2
Soient S_1 et S_2 deux systèmes de transitions étiquetées d'états initiaux respectifs q_{01} et q_{02}, soit $\sim_\Lambda$

une équivalence de bisimulation et soit $<$ une relation d'ordre définie sur $Ex_\Lambda(q_{01}, q_{02})$. L'ensemble des séquences diagnostiques minimales modulo $<$ pour $\sim_\Lambda$ est défini par :

$$Diag^{<}_{\Lambda}(S_1, S_2) = \{\sigma \in Diag_\Lambda(S_1, S_2) \mid \sigma \text{ élémentaire } \wedge \nexists \sigma' . (\sigma' \in Diag_\Lambda(S_1, S_2) \wedge \sigma' < \sigma)\}$$

■

Remarque 4-3

1. Les relations $<_l$, $<_p$ et $<_\cap$ sont ordonnées vis-à-vis de l'inclusion de la façon suivante :

   $$<_p \subset <_\cap \subset <_l .$$

   On en déduit les inclusions suivantes entre les ensembles de séquences diagnostiques minimales qui leur sont associées :

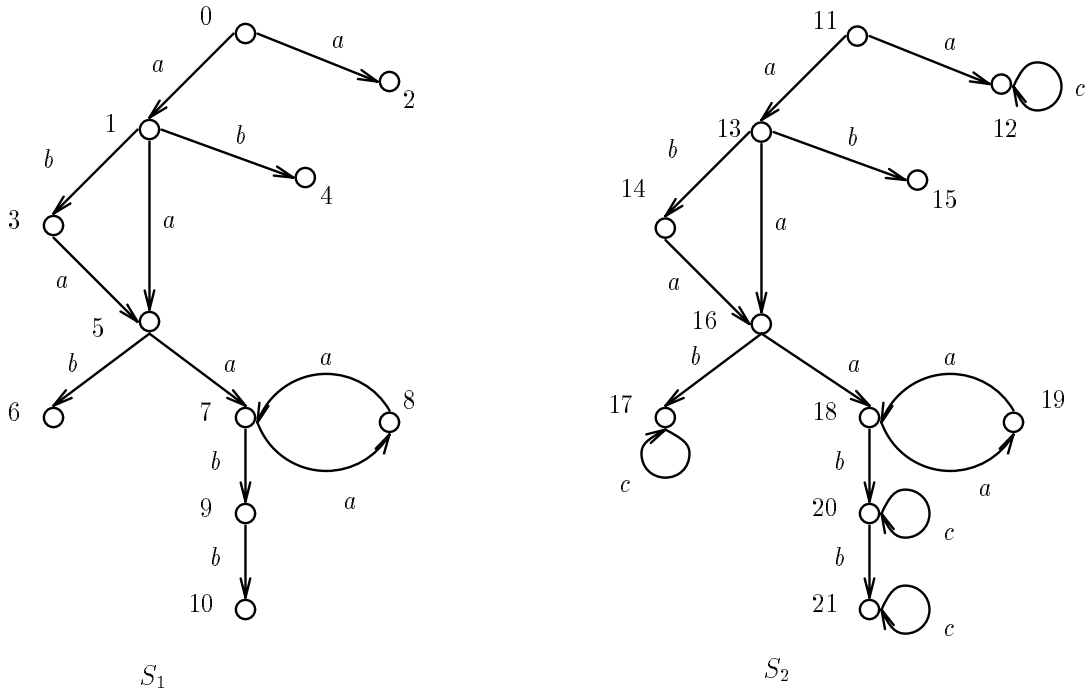
   $$Diag^{<_l}_{\Lambda}(S_1, S_2) \subset Diag^{<_\cap}_{\Lambda}(S_1, S_2) \subset Diag^{<_p}_{\Lambda}(S_1, S_2) \subset Diag_\Lambda(S_1, S_2).$$

2. Du fait que l'ensemble des séquences d'exécution élémentaires d'un système de transitions étiquetées est toujours fini, il en est de même pour $Diag^{<}_{\Lambda}(S_1, S_2)$. Notons d'autre part que toute séquence minimale pour les relations $<_l$ ou $<_\cap$ est nécessairement élémentaire.

■

On donne sur l'exemple suivant les différents ensembles de séquences diagnostiques minimales obtenues à l'aide des relations $<_l$, $<_p$ et $<_\cap$.

Exemple 4-4



Les systèmes de transitions étiquetées S_1 et S_2 de la figure ci-dessus ne sont pas équivalents modulo la bisimulation forte : les états 12, 17, 20 et 21 de S_2 peuvent effectuer une transition étiquetée par l'action c contrairement aux états 2, 6, 9 et 10 de S_1 qui leur "correspondent".

$Diag_\Lambda(S_1, S_2)$ contient les séquences diagnostiques suivantes :

$\sigma_1 = ((0,11) \xrightarrow{a} (2,12))$

$\sigma_2 = ((0,11) \xrightarrow{a} (1,13) \xrightarrow{a} (5,16) \xrightarrow{b} (6,17))$

$\sigma_3 = ((0,11) \xrightarrow{a} (1,13) \xrightarrow{b} (3,14) \xrightarrow{a} (5,16) \xrightarrow{b} (6,17))$

$\sigma_4 = ((0,11) \xrightarrow{a} (1,13) \xrightarrow{a} (5,16) \xrightarrow{a} (7,18) \xrightarrow{b} (9,20))$

$\sigma_5 = ((0,11) \xrightarrow{a} (1,13) \xrightarrow{b} (3,14) \xrightarrow{a} (5,16) \xrightarrow{a} (7,18) \xrightarrow{b} (9,20))$

$\sigma_6 = ((0,11) \xrightarrow{a} (1,13) \xrightarrow{a} (5,16) \xrightarrow{a} (7,18) \xrightarrow{b} (9,20) \xrightarrow{b} (10,21))$

$\sigma_7 = ((0,11) \xrightarrow{a} (1,13) \xrightarrow{a} (5,16) \xrightarrow{a} (7,18) \xrightarrow{a} (8,19) \xrightarrow{a} (7,18) \xrightarrow{b} (9,20))$

$\vdots$

Pour les relations d'ordre $<_l$, $<_\cap$ et $<_p$, les ensembles de séquences diagnostiques minimales obtenus sont alors respectivement :

$$Diag_\Lambda^{<_l}(S_1,S_2) = \{\sigma_1\},\ \ Diag_\Lambda^{<_\cap}(S_1,S_2) = \{\sigma_1,\sigma_2\},\ \ Diag_\Lambda^{<_p}(S_1,S_2) = \{\sigma_1,\sigma_2,\sigma_3,\sigma_4,\sigma_5\}.$$

■

En pratique, l'intérêt d'une relation d'ordre donnée pour définir des séquences diagnostiques minimales dépend en fait du nombre de séquences obtenues par rapport au nombres d'erreurs distinctes détectées à l'aide de ces séquences sur le système de transitions représentant l'implémentation à vérifier. On donne quelques éléments de comparaison entre les relation $<_l$, $<_p$ et $<_\cap$ en fonction de ce critère :

relation $<_l$: l'ensemble des séquences minimales pour la relation $<_l$ ne contient par définition que les séquences les plus courtes (en nombre d'états) de $Diag_\Lambda^{<}(S_1,S_2)$, ce qui limite nécessairement le nombre d'erreurs détectées. Toutefois, construire des séquences qui soient minimales pour cette relation peut être intéressant lorsque le nombre et la longueur des séquences diagnostiques sont très élevées, et que par conséquent seules les séquences les plus courtes s'avèrent exploitables.

relation $<_p$: cette relation est similaire à la relation d'ordre proposée dans [Ras90] pour définir des séquences diagnostiques minimales dans le cas de spécifications décrites en logique temporelle LTAC. Dans notre cas, un sous-ensemble intéressant de séquences minimales pour la relation $<_p$ est l'ensemble des séquences de $Diag_\Lambda^{<}(S_1,S_2)$ qui ne contiennent qu'un seul état erreur (ce sous-ensemble est défini plus formellement dans la section 4.6). Néanmoins, l'inconvénient principal de cette relation est le nombre assez élevé de séquences minimales obtenues, dont la plupart correspondent en fait souvent à la même erreur.

relation $<_\cap$: les séquences diagnostiques minimales obtenues à l'aide de cette relation sont les plus courtes séquences qui conduisent à une erreur donnée, et son utilisation est donc justifiée lorsque le nombre de séquences menant à une même erreur est élevé (ce qui est par exemple le cas lorsque le produit synchrone entre les deux systèmes contient des cycles). En contrepartie, en fonction de la forme des systèmes de transitions à comparer, un certain nombre d'erreurs peuvent ne pas être accessibles par une séquence diagnostique minimale pour la relation $<_\cap$.

4.5 Mise en œuvre à partir des algorithmes de vérification

On s'intéresse dans cette section à la mise en œuvre du calcul des séquences diagnostiques à partir des algorithmes de vérification présentés au chapitre précédent. On montre tout d'abord comment

les différentes équivalences de bisimulation peuvent être prises en compte, puis on décrit les modifications à apporter à la procédure de décision de la bisimulation forte pour permettre l'élaboration du diagnostic.

Comme il a été rappelé au précédent chapitre (chapitre 3, section 3.3), l'algorithme classique utilisé pour comparer des systèmes de transitions étiquetées modulo une bisimulation faible $\sim_\Lambda$ comporte deux phases :

1. On calcule tout d'abord une forme pré-normale pour chacun des systèmes, qui dépend de la relation d'équivalence considérée.

2. On compare ensuite ces formes pré-normales modulo la bisimulation forte, en utilisant un algorithme de raffinement de partitions.

Par conséquent, pour expliquer la non-équivalence entre les deux systèmes de transitions, il suffit en pratique de ne considérer que la deuxième phase de l'algorithme, ce qui revient à construire des séquences diagnostiques pour la bisimulation forte directement sur les formes pré-normales. L'intérêt de cette approche est qu'elle ne requiert qu'un seul algorithme de diagnostic (pour la bisimulation forte), et qu'elle est complètement indépendante de la relation d'équivalence considérée. En contrepartie, son principal inconvénient est que les séquences diagnostiques obtenues ne se réfèrent pas au systèmes de transitions originaux mais à leurs formes pré-normales. Par suite, si la correspondance entre les *états* qui apparaissent dans ces séquences diagnostiques et les états des systèmes originaux est facilement envisageable, il n'en est pas de même pour les *actions*, qui ont été obtenues sur les formes pré-normales par renommage des éléments des langages λ_i de Λ en leur représentant canonique $[\lambda_i]$. Des solutions à ce problème seront envisagées dans la section 4.9.

Dans le cas de la bisimulation forte, les procédures de décision classiques reposent sur des algorithmes de raffinement de partitions qui permettent de calculer les classes d'équivalence pour cette relation sur l'ensemble des états d'un système de transitions étiquetées. Comparer deux systèmes S_1 et S_2, revient alors à calculer ces classes d'équivalence sur leur système union $S_1 \cup S_2$ et à vérifier si leurs états initiaux q_{01} et q_{02} appartiennent ou non à la même classe.

On propose dans la suite des algorithmes qui permettent de construire des séquences diagnostiques minimales à l'aide de ces classes d'équivalence lorsque les deux systèmes ne se bisimulent pas. La procédure de décision pour la bisimulation forte (*cf.* chapitre 3, algorithme 3-1) est alors légèrement modifié de manière itérer les opérations de raffinement jusqu'a obtention des classes d'équivalence (même si au cours du raffinement les états initiaux des deux systèmes apparaissent dans des classes différentes de la partition courante) :

Algorithme 4-1

Q_1 et Q_2 sont les ensembles d'états des systèmes S_1 et S_2,
ρ et ρ' sont des partitions sur $Q_1 \cup Q_2$,
Φ_Λ est un opérateur de raffinement de partition.

début
  $\rho := \{Q_1 \cup Q_2\}$;
  $\rho' := \Phi_\Lambda(\rho)$;
  tantque $(\rho' \neq \rho)$ **faire**
    $\rho := \rho'$;
    $\rho' := \Phi_\Lambda(\rho)$;
  ftantque
  si $\exists B \in \rho' \,.\, \{q_{01}, q_{02}\} \subseteq B$ **alors**
    afficher $(S_1 \sim_\Lambda S_2)$

sinon
afficher ($S_1 \not\sim_\Lambda S_2$) ;
construire_séquences_diagnostiques (S_1, S_2, ρ)
fsi.
fin.

■

Plusieurs algorithmes seront envisagés pour la procédure *construire_séquences_diagnostiques* en fonction de la relation d'ordre choisie pour définir les séquences minimales. Toutefois, afin de limiter le coût en mémoire de la procédure de décision, on ne considère que des algorithmes qui permettent d'afficher les séquences "à la volée", au fur et à mesure de leur construction, sans avoir à mémoriser de manière globale l'ensemble des séquences obtenues. Par conséquent, les solutions proposées seront toutes basés sur des parcours en profondeur du produit synchrone $S_1 \times S_2$.

4.6 Calcul de séquences minimales pour la relation $<_p$

On définit en premier lieu le sous-ensemble des séquences minimales qui seront calculées, puis on propose un algorithme qui permet de les construire effectivement.

4.6.1 Caractérisation des séquences minimales

Le lemme suivant établit une condition nécessaire et suffisante pour qu'une séquence diagnostique soit minimale pour la relation $<_p$.

Lemme 4.6-1
Soient S_1 et S_2 deux systèmes de transitions étiquetées d'états initiaux respectifs p_0 et q_0, et soit $\sim_\Lambda$ une équivalence de bisimulation.

Pour tout $k > 1$ et pour toute séquence *élémentaire* σ de ${Diag_\Lambda}^k(p_0, q_0)$, on a :

$$\exists \sigma' \,.\, \sigma' \in Diag_\Lambda(S_1, S_2) \;\wedge\; \sigma' <_p \sigma \;-\; \exists i \,.\, 0 \leq i < k-1 \;\wedge\; (\sigma(i) = (p_i, q_i) \;\Rightarrow\; p_i \not\sim^1_\Lambda q_i)$$

■

Preuve : On pose

$$\sigma = ((p_0, q_0), \lambda_0, (p_1, q_1), \ldots, \lambda_{k-2}, (p_{k-1}, q_{k-1}))$$

sens '⇒' : Par définition de la relation $<_p$, on a : Pour tout $\sigma' <_p \sigma$,

$$\sigma' <_p \sigma \;\Rightarrow\; \exists i \,.\, 0 \leq i < k-1 \;\wedge\; \sigma' = ((p_0, q_0), \lambda_0, (p_1, q_1), \ldots, \lambda_{i-1}, (p_i, q_i)).$$

et $(p_i, q_i) \neq (p_{k-1}, q_{k-1})$ puisque σ est élémentaire.

De plus, par définition de $Diag_\Lambda(S_1, S_2)$,

$$\sigma' \in Diag_\Lambda(S_1, S_2) \;\Rightarrow\; p_i \not\sim^1_\Lambda q_i.$$

sens '⇐' : On a

$$\sigma' = ((p_0, q_0), \lambda_0, (p_1, q_1), \ldots, \lambda_{i-1}, (p_i, q_i)) \;\wedge\; p_i \not\sim^1_\Lambda q_i$$

Or,

$$\sigma \in {Diag_\Lambda}^k(p_0, q_0) \;\Rightarrow\; \forall j,\; 0 \leq j < i,\; \exists n_j \,.\, 1 \leq n_j \leq k \,.\, (p_{j+1}, q_{j+1}) \in \mathrm{Fail}^{n_j}_{\lambda_j}(p_j, q_j)$$

d'où

$$\sigma' \in Diag_\Lambda(S_1, S_2)$$

□.

La proposition 4.6-1 définit alors l'ensemble des séquences diagnostiques minimales pour la relation $<_p$:

Proposition 4.6-1
Soient S_1 et S_2 deux systèmes de transitions étiquetées, et soit $\sim_\Lambda$ une équivalence de bisimulation. On a alors :

$$Diag_\Lambda^{<_p}(S_1, S_2) = \{\sigma \in Diag_\Lambda(S_1, S_2) \mid \forall i \,.\, 0 \leq i < |\sigma| \;\Rightarrow\; (\sigma(i) = (p_i, q_i) \;\Rightarrow\; p_i \not\sim_\Lambda^1 q_i)\}$$

■

Preuve : c'est une conséquence directe du lemme 4.6-1. □.

4.6.2 Algorithme de construction des séquences minimales

On considère deux systèmes de transitions étiquetées $S_i = (Q_i, A_i, T_i, q_{0i})_{i=(1,2)}$, $\sim_\Lambda$ une équivalence de bisimulation, et $S = (Q, A, T, (q_{01}, q_{02})$ le produit synchrone $S_1 \times_\Lambda S_2$.

On note ρ, la partition associée à $\sim_\Lambda$ ($\rho = \{B_i\}$) et, pour tout couple d'états (q_1, q_2) de $Q_1 \times Q_2$, on note $\text{Fail}_\Lambda(q_1, q_2)$ l'ensemble défini par :

$$\text{Fail}_\Lambda(q_1, q_2) = \{(\lambda, (q_1', q_2')) \in T\,[(q_1, q_2)] \mid \exists k \,.\, (q_1', q_2') \in \text{Fail}_\lambda^k(q_1, q_2)\}$$

En utilisant les résultats établis dans la proposition 4.6-1, la procédure informelle suivante, appliquée à l'état initial de S, permet de construire une séquence diagnostique minimale pour la relation $<_p$:

Soit σ la séquence en cours de construction, et (q_1, q_2) le dernier état ajouté à σ :

- Si $Act(q_1) \neq Act(q_2)$, alors (q_1, q_2) termine la séquence diagnostique σ.
- Si $Act(q_1) = Act(q_2)$, alors on poursuit la construction de σ en lui ajoutant un couple $(\lambda, (q_1', q_2'))$ choisi arbitrairement dans $\text{Fail}_\Lambda(q_1, q_2)$, et tel que (q_1', q_2') n'appartienne pas à σ. On réapplique alors cette procédure à partir du nouvel état courant (q_1', q_2').

L'algorithme précédent peut directement être mis en œuvre à partir d'un algorithme classique de parcours en profondeur de S. Les structures de données nécessaires sont alors :

- Une pile *StState* (*resp. StLabel*), pour mémoriser les états (*resp.* les action) de la séquence diagnostique en cours de construction.
- Une pile *StTrans*, pour mémoriser les transitions qui sont exécutables à partir des états de la séquence courante et qui n'ont pas encore été tirées.

L'algorithme de construction des séquences diagnostiques minimales pour la relation $<_p$ est alors le suivant :

Algorithme 4-2

procédure construire_séquences_diagnostiques (S_1, S_2, ρ)
début
  $StState := \emptyset$; $StTrans := \emptyset$; $StLabel := \emptyset$;
  empiler $((q_{01}, q_{02})$, *StState*) ;
  empiler ($\text{Fail}_\Lambda(q_{01}, q_{02})$, *StTrans*) ; **(1)**
  tantque $StState \neq \emptyset$ **faire**
    (q_1, q_2) := sommet (*StState*) ;
    si (sommet (*StTrans*) $\neq \emptyset \wedge Act(q_1) = Act(q_2)$) **alors**
      choisir $(\lambda, (q_1', q_2'))$ **dans** sommet (*StTrans*) ;

```
            sommet (StTrans) := sommet (StTrans) −{(λ, (q'1, q'2))} ;
            si (q'1, q'2) ∉ StState alors
                  empiler (λ, StLabel) ;
                  empiler ((q'1, q'2), StState) ;
                  empiler (Fail_Λ(q'1, q'2), StTrans)   (2)
            fsi
      sinon
            (* StState et StLabel contiennent les états et les actions
              d'une séquence diagnostique minimale *)
            afficher StState et StLabel ;
            dépiler (StLabel) ;
            dépiler (StState) ;
            dépiler (StTrans)
      fsi
   ftantque
fin.
```

■

De plus, lorsque l'équivalence de bisimulation considérée est la bisimulation forte, et en supposant que la partition ρ est donnée sous la forme d'une fonction *inclasse* qui, pour tout état de $Q_1 \cup Q_2$ retourne l'indice de sa classe, le calcul de $\mathrm{Fail}_\Lambda(q_1, q_2)$ (lignes notées **(1)** et **(2)** dans l'algorithme) peut se faire en un temps de l'ordre de $|T\ [(q_1, q_2)]|$ en appliquant l'algorithme suivant :

Algorithme 4-3

```
fonction calculer_Fail_Λ (q1, q2)
Soit l est un ensemble d'états, dont le cardinal est majoré par |T1λ[q1] + T2λ[q2]|
début
   Fail_Λ := ∅ ;
   pour tout λ dans Λ
         l := ∅ ;
         pour tout (q'1, q'2) dans Tλ[(q1, q2)]
               si inclasse(q'1) = inclasse(q'2) alors
                     l := l ∪ {q'1, q'2}
               fsi
         fpour
   Fail_Λ := Fail_Λ ∪ {(λ, (T1λ[q1] − l) × T2λ[q2])} ∪ {(λ, T1λ[q1] × (T2λ[q2] − l))} ;
   retourner (Fail_Λ)
fin.
```

■

Par conséquent, s'agissant d'un algorithme de parcours en profondeur classique, et puisque chaque état peut être traité en un temps constant, les coûts en temps et en mémoire de l'algorithme 4-2 pour la recherche des séquences diagnostiques sont respectivement de $O(c^d)$ et $O(d)$ [Hol89], où c est le facteur de branchement de S (i.e, le nombre moyen de successeurs par état) et d la longueur de sa plus longue séquence élémentaire (d est donc majoré par le nombre d'états n de S).

Toutefois, du fait que seul l'ensemble des états non équivalents de S est exploré, le comportement de l'algorithme en pratique est généralement bien meilleur que le laissent supposer ces valeurs théoriques obtenues dans le cas le plus défavorable (i.e, lorsque tous les états de S sont non équivalents). Il nous parait donc plus intéressant (et plus représentatif) d'exprimer le coût en temps de cet algorithme en fonction du nombre s de séquences effectivement construites. Par suite, puisque chaque séquence est

calculée en un temps qui est de l'ordre du nombre d'états qui la composent, la complexité obtenue est de $O(s.n)$.

4.7 Calcul de séquences minimales pour la relation $<_\cap$

De façon similaire à ce qui a été fait dans la section précédente, nous définissons en premier lieu un sous-ensemble de séquences qui sont minimales pour la relation $<_\cap$, puis nous proposons un algorithme qui permettra de les construire de manière efficace à l'aide d'un parcours en profondeur de $S_1 \times_\Lambda S_2$. Là encore, si cet algorithme peut en théorie s'appliquer à n'importe quelle équivalence de bisimulation, nous donnons sa complexité uniquement dans le cas de la bisimulation forte puisqu'il est suffisant de considérer cette seule relation en pratique (*cf.* section 4.5).

4.7.1 Caractérisation des séquences minimales

En notant d la longueur de la plus longue séquence acyclique de $S_1 \times_\Lambda S_2$, l'algorithme informel suivant calcule les séquences diagnostiques qui sont minimales pour la relation $<_\cap$:

Algorithme 4-4

procédure construire_séquences_diagnostiques (S_1, S_2)
début
  pour k allant de 2 à d **faire**
    pour toute séquence *élémentaire* σ de $Diag_\Lambda{}^k(S_1, S_2)$ **faire**
      si pour tout $i \in [1, k-1]$ $\sigma(i)$ n'est pas marqué **alors**
        marquer $\sigma(i)$;
        afficher σ (* σ est minimale pour la relation $<_\cap$ *)
      fsi
    fpour
  fpour
fin.

■

La validité de l'algorithme 4-4 repose sur le fait que les ensembles $Diag_\Lambda{}^k(S_1, S_2)$ sont successivement calculés pour des valeurs *croissantes* de k, ce qui parait difficile à mettre en œuvre lorsque les séquences sont obtenues à l'aide d'un parcours en profondeur de $S_1 \times_\Lambda S_2$. Toutefois, nous montrons dans la suite qu'il est possible de définir un sous-ensemble de séquences diagnostiques minimales pour la relation $<_\cap$ qui soit calculable en utilisant ce type de parcours.

Définition 4.7-1
Soient S_1 et S_2 deux systèmes de transitions étiquetées d'états initiaux respectifs q_{01} et q_{02}, tels que S_1 et S_2 soient non équivalents modulo l'équivalence de bisimulation $\sim_\Lambda$.

On note k_0 le plus petit entier tel que q_{01} et q_{02} ne soient pas équivalents modulo $\sim_\Lambda^{k_0}$, et k_m le plus petit entier k tel que $\sim_\Lambda^{k+1} = \sim_\Lambda^k$:

$$k_0 = min\{k \mid q_{01} \not\sim_\Lambda^k q_{02}\}$$
$$k_m = min\{k \mid \forall (q_1, q_2) \in Q_1 \times Q_2 \,.\, q_1 \sim_\Lambda^k q_2 \Rightarrow q_1 \sim_\Lambda q_2\}.$$

On définit la suite d'ensembles $(D_\Lambda^k(S_1, S_2))$ la façon suivante :

- $D^{k_0}(S_1, S_2) = Diag_\Lambda{}^{k_0}(S_1, S_2)$

- $\forall k,\ k_0 < k \leq k_m + 1,$
$$D^k_\Lambda(S_1, S_2) = \{\sigma \in {Diag_\Lambda}^k(S_1, S_2) \mid \forall i,\ 1 \leq i \leq k-1\ .\ \sigma(i) = (p_i, q_i) \Rightarrow p_i \sim_\Lambda^{k-i-1} q_i \wedge \forall k'\ .\ k' < k \Rightarrow \nexists \sigma'\ .\ \sigma' \in D^{k'}_\Lambda(S_1, S_2) \wedge \sigma(i) \in \sigma'\}$$

■

Proposition 4.7-1
Soient S_1 et S_2 deux systèmes de transitions étiquetées, et soit $\sim_\Lambda$ une équivalence de bisimulation. On a alors :

(i) $S_1 \not\sim_\Lambda S_2 \Rightarrow \exists k\ .\ D^k_\Lambda(S_1, S_2) \neq \emptyset$.

(ii) $\forall k,\ D^k_\Lambda(S_1, S_2) \subseteq Diag^{<\cap}_\Lambda(S_1, S_2)$.

■

Preuve : Elle est immédiate :

- *(i)* est une conséquence des propositions 4.3-2 *(iii)* et 4.3-3.
- *(ii)* est obtenu par construction des ensembles D^k_Λ.

□.

Dans le cas de la bisimulation forte, on propose un algorithme qui permet de construire les séquences diagnostiques appartenant aux ensembles D^k_Λ et qui seront donc minimales pour la relation $<_\cap$ (proposition 4.7-1). Plus précisément, cet algorithme consiste en deux phases distinctes qui sont détaillées dans la suite :

1. Le calcul des classes d'équivalence pour les relations $\sim^k_\Lambda$, qui sont obtenues en modifiant l'opérateur de raffinement de partition utilisé dans la procédure de décision classique ;
2. La construction des séquences appartenant aux ensembles D^k_Λ, qui est effectuée lors du parcours en profondeur de certaines séquences de $S_1 \times_\Lambda S_2$ déterminées à l'aides des classes d'équivalences calculées en 1.

4.7.2 Calcul des relations $\sim^k_\Lambda$

Au chapitre 3, nous avons décrit une procédure de décision pour une équivalence de bisimulation $\sim_\Lambda$ entre deux systèmes de transitions étiquetées S_1 et S_2 dont le principe consiste à calculer jusqu'à stabilité des raffinements successifs d'une partition initiale définie sur l'ensemble Q des états du système $S_1 \cup S_2$. La partition finale ρ_{Φ_Λ} ainsi obtenue correspond alors à la partition associée à la relation d'équivalence $\sim_\Lambda$ sur Q.

Plus formellement, la partition ρ_{Φ_Λ} a été définie comme la limite d'une suite $(\rho_i)_{i \in \mathcal{N}}$ telle que (*cf.*chapitre 3, section 3.1) :

- $\rho_0 = \{Q\}$,
- $\forall k \geq 0, \rho_{k+1} = Refine_\Lambda(\rho_k, \rho_k)$.

Nous montrons dans cette section que, moyennant une organisation particulière de ces raffinements, il est possible lors du calcul de ρ_{Φ_Λ} de construire l'ensemble des partitions associées au relations $\sim^k_\Lambda$ (pour tout $k \geq 0$). Nous décrivons alors comment ce calcul peut être efficacement mis en œuvre à l'aide de l'algorithme de raffinement proposé par Paige & Tarjan (*cf.* chapitre 3, section 3.2.2) et intégré ainsi à la procédure de décision classique de la bisimulation forte.

Notons que cette section peut être évitée en première lecture sans que cela nuise à la compréhension de l'algorithme de calcul des séquences diagnostiques minimales qui est présenté dans la section suivante.

Proposition 4.7-2
Soit $S = (Q, A, T, q_0)$ un système de transitions étiquetées, et $\sim_\Lambda$ une équivalence de bisimulation. On considère la suite $(\rho_i)_{i \in \mathcal{N}}$ de partitions sur Q définies par :

- $\rho_0 = \{Q\}$,
- $\rho_1 = Refine_\Lambda(\rho_0, \rho_0)$,
- $\forall k \geq 1, \rho_{k+1} = Refine_\Lambda(\rho_k, \rho_k - \rho_{k-1})$.

On a alors :

Pour tout $i \geq 0$, ρ_i est la partition associée à la relation $\sim_\Lambda^i$

■

Preuve :

En premier lieu, il est facile de voir que, pour tout couple de partition (ρ_k, ρ_{k+1}) sur Q et pour tout entier $k \geq 0$, ρ_{k+1} est la partition associée à la relation $\sim_\Lambda^{k+1}$ si et seulement si,

- ρ_k est la partition associée à la relation $\sim_\Lambda^k$,
- et ρ_{k+1} est *stable* par rapport à ρ_k (*cf.* chapitre 3, définition 3.4-2).

Plus formellement, cette propriété s'écrit :

$$\rho_{k+1} = \sim_\Lambda^{k+1} - \rho_k = \sim_\Lambda^k \wedge \pi_\Lambda(\rho_{k+1}\rho_k) \quad \textbf{(1)}$$

La preuve de **(1)** est obtenue en comparant les définitions de $\sim_\Lambda^{k+1}$ et π_Λ.

On montre alors par induction que, pour toutes partitions ρ_i de $(\rho_i)_{i \in \mathcal{N}}$, on a :

$$\rho_{i+1} = Refine_\Lambda(\rho_i, \rho_i) \quad \textbf{(2)}$$

- Elle est immédiate pour $k = 1$.
- On suppose alors que pour un entier k fixé, la propriété **(2)** est vérifiée pour toute partition (ρ_i) avec $i \leq k$. Or, en utilisant la proposition 3.2-1,

$$\begin{aligned} \rho_k = Refine_\Lambda(\rho_{k-1}, \rho_{k-1}) \quad &\Rightarrow \quad Refine_\Lambda(\rho_k, \rho_k - \rho_{k-1}) = Refine_\Lambda(\rho_k, \rho_k) \\ &\Rightarrow \quad \rho_{k+1} = Refine_\Lambda(\rho_k, \rho_k) \end{aligned}$$

Par conséquent, d'après la proposition 3.1-3 (vi), on a

$$\forall i \geq 0 \;.\; \pi_\Lambda(\rho_{i+1}\rho_i)$$

d'où, en utilisant **(1)**,

$$\forall i \geq 0 \;.\; \rho_{i+1} = \sim_\Lambda^{i+1} .$$

□.

Il reste alors à décrire comment les partitions ρ_i de la suite définie dans la proposition 4.7-2 peuvent être obtenues en utilisant l'algorithme de calcul de raffinements proposé par Paige & Tarjan :

Nous avons vu au chapitre 3 (section 3.2.2) que le principe de cet algorithme consiste à maintenir une liste de *partitionneurs* W_i, constituée de *blocs simples* et de *blocs arbres*. A chaque pas de l'algorithme, un élément de cette liste est sélectionné pour raffiner la partition courante ρ_i (et calculer ainsi une nouvelle partition ρ_{i+1}) et la liste de partitionneurs est remise à jour (en une liste W_{i+1}).

Formellement, on calcule ainsi la limite d'une suite $(\rho_i, W_i)_{i\in\mathcal{N}}$, définie à partir d'un opérateur Φ'_Λ (définition 3.2-1, 3.2-2 et 3.2-4) en utilisant l'algorithme suivant :

début
$(\rho, W) := (\{Q\}, \{Q\})$;
répéter
choisir B **dans** W ;
$(\rho, W) := \Phi"_\Lambda(\rho, W, B)$
jusqua $W = \emptyset$
fin.

De façon intuitive, la solution que nous proposons pour construire les partitions ρ_i associées aux relations $\sim^i_\Lambda$ consiste à ordonner les raffinements nécessaires au calcul de ρ_i en distinguant les *nouveaux* partitionneurs obtenues lors de ce calcul (et qui correspondent donc aux classes obtenues par décomposition d'une classe de ρ_i) de ceux qui ont été obtenues lors du calcul de ρ_{i-1} (et qui correspondent donc à la décomposition d'une classe de cette partition).

Plus précisément on définit ainsi un opérateur $\theta"_\Lambda$ en remplacement de l'opérateur $\Phi"_\Lambda$ qui permet de raffiner une partition à l'aide d'un partitionneur sélectionné dans un ensemble initial W_1 tout en construisant l'ensemble W_2 des nouveaux partitionneurs obtenus :

Définition 4.7-2
Soit ρ une partition sur un ensemble Q, et W_1 et W_2 deux ensembles de partitionneurs constitués de blocs simples et de blocs arbres. Soit x un élément de W_1.

$$\theta_\Lambda"(\rho, W_1, W_2, x) = (\rho', W'_1, W'_2)$$

avec,

- $\rho' = Ref_\Lambda(\rho, x)$ (*cf.* définition 3.2-3),
- Soit $X \in \rho$. W'_1 et W'_2 sont définis de la manière suivante :

  1) Si $X \in W_1$ (*resp.* $X \in W_2$), $X \neq x$, et X n'est pas partitionné alors X est laissée inchangée dans W'_1 (*resp.* dans W'_2).

  2) Si $X \in W_1$ et X est partitionnée en $\{X_i\}$, alors X est *laissé inchangé* dans W'_1 (si $x \neq X$) et les X_i *sont ajoutés* à W'_2. Si $X \in W_2$ et X est partitionnée en $\{X_i\}$ alors X est remplacé par $\{X_i\}$ dans W'_2. dans W.

  3) Si $X \notin W_1$ et $X \notin W_2$ (ni en tant que bloc simple, ni en tant que feuille d'un bloc arbre), et X est partitionnée en $\{X_i\}$, alors le bloc arbre représentant cette décomposition est inséré dans W'_2.

  4) Les blocs arbres n de W_1 (*resp.* de W_2) sont remplacés dans W_1 (*resp.* dans W_2) par les blocs arbres n', obtenus à partir de n en remplaçant toutes les feuilles de n correspondant à des classes partitionnées par le bloc arbre représentant ces décompositions.

  De plus, si x est un bloc arbre n alors

  5) Les fils gauches et droits non feuilles de n sont insérés dans W'_2.

■

Proposition 4.7-3
Soit ρ une partition définie sur un ensemble Q, et soit W un ensemble fini de partitionneur (blocs simples ou blocs arbres). On note $\mathcal{S} = (\rho_i, W'_i, W"_i)_{(i\in\mathcal{N})}$ la suite définie par :

- $(\rho_0, W'_0, W"_0) = (\rho, W, \emptyset)$

- $\forall i \geq 0$, $(\rho_{i+1}, W'_{i+1}, W"_{i+1}) = \theta"_\Lambda(\rho_i, W'_i, W"_i, x)$ avec $x \in W'_i$.

  $\mathcal{S}$ converge vers une limite $(\rho_r, \emptyset, W"_r)$.

■

Preuve : Elle est immédiate puisque, par définition de $\theta"_\Lambda$,

- aucun élément n'est ajouté aux ensembles W'_i,
- au pas i, le partitionneur courant est systématiquement enlevé de W'_i.

Par suite, pour tout i $|W'_{i+1}| < |W'_i|$ et la suite converge. □.

La proposition 4.7-3 permet alors de définir l'opérateur θ'_Λ, qui remplace Φ'_Λ :

Définition 4.7-3
Soient ρ une partition sur un ensemble Q et W un ensemble de partitionneurs. L'opérateur θ'_Λ est défini de la manière suivante :

$$\theta'_\Lambda(\rho, W) = (\rho_r, W"_r)$$

où $(\rho_r, \emptyset, W"_r)$ est la limite de la suite $(\rho_i, W'_i, W"_i)_{(i \in \mathcal{N})}$ définie par :

- $(\rho_0, W'_0, W"_0) = (\rho, W, \emptyset)$
- $\forall i \geq 0$, $(\rho_{i+1}, W'_{i+1}, W"_{i+1}) = \theta"_\Lambda(\rho_i, W'_i, W"_i, x)$ avec $x \in W'_i$.

On note alors $Part(\rho_r)$ l'ensemble des *classes* effectivement utilisées comme partitionneurs pour calculer ρ_r :

$$Part(\rho_r) = \{Classe(x) \mid \exists i,\ 0 \leq i < r\ .\ (\rho_{i+1}, W'_{i+1}, W"_{i+1}) = \theta"_\Lambda(\rho_i, W'_i, W"_i, x)\}$$

■

Remarque 4-4

1. Si x est un *bloc arbre*, alors
   $$\theta"_\Lambda(\rho, W_1, W_2, x) = (\rho', W'_1, W'_2) \ - \ \Phi"_\Lambda(\rho, W_1 \cup W_2, x) = (\rho', W'_1 \cup W'_2)$$
2. Par conséquent, si W ne contient que des blocs arbres, alors il n'y a pas de différences entre les opérateurs θ'_Λ et Φ'_Λ et l'opérateur $\theta"_\Lambda$ sert uniquement à garantir que les raffinements sont effectués en préservant l'ordre spécifié :
   $$\theta'_\Lambda(\rho, W) = (\rho', W') \ - \ \Phi'_\Lambda(\rho, W) = (\rho', W')$$
3. Enfin, lorsque la partition initiale est la partition universelle $\{Q\}$, il est facile de voir par induction que, pour $i \geq 1$, les ensembles W_i obtenus par application de θ'_Λ ne contiennent que des bloc arbres :
   - Si W_1 est non vide, alors il contient un bloc arbre unique obtenu par décomposition du seul bloc simple $\{Q\}$ contenu dans W_0 (définition de $\theta_\Lambda"$).
   - Si W_i contient que des blocs arbres pour i fixé, alors W_{i+1} ne contiendra que des blocs arbres (toujours par définition de $\theta_\Lambda"$).

■

Il reste alors à montrer que, ainsi défini, l'opérateur θ'_Λ vérifie bien dans tous les cas les propriétés requises. Les deux lemmes suivants permettent successivement d'établir que, lors du calcul de la partition ρ_{k+1} :

- toute classe de ρ_k qui n'est pas une classe de ρ_{k-1} est utilisé pour raffiner ρ_k lors du calcul de ρ_{k+1},
- toute classe utilisé pour raffiner ρ_k lors du calcul de ρ_{k+1} et soit une classe de ρ_k soit une union de classes de ρ_k. On garantit ainsi que les éléments de ρ_{k+1} ne sont pas calculés à l'aide de partitionneurs issues de la décomposition d'une classe de ρ_k.

Lemme 4.7-1
Soit $S = (Q, A, T, q_0)$ un système de transitions étiquetées et soit $\sim_\Lambda$ une équivalence de bisimulation. Soit $(\rho_i, W_i)_{(i \in \mathcal{N})}$ la suite définie par :

- $(\rho_0, W_0) = (\{Q\}, \{Q\})$,
- $\forall i \geq 0$, $(\rho_{i+1}, W_{i+1}) = \theta'_\Lambda(\rho_i, W_i)$.

Pour tout $k \geq 0$, toute classe de ρ_k issue de la décomposition d'une classe de ρ_{k-1} servira de partitionneur lors du calcul de ρ_{k+1} :

$$(\forall X \,.\, X \in \rho_k \ \wedge \ X \notin \rho_{k-1}) \ \Rightarrow \ X \in Part\,(\rho_{k+1}).$$

■

Lemme 4.7-2
Soit $S = (Q, A, T, q_0)$ un système de transitions étiquetées et soit $\sim_\Lambda$ une équivalence de bisimulation. Soit $(\rho_i, W_i)_{(i \in \mathcal{N})}$ la suite définie par :

- $(\rho_0, W_0) = (\{Q\}, \{Q\})$,
- $\forall i \geq 0$, $(\rho_{i+1}, W_{i+1}) = \theta'_\Lambda(\rho_i, W_i)$.

Pour tout $k \geq 0$, toute classe utilisée pour raffiner la partition ρ_k lors du calcul de ρ_{k+1} est soit une classe de ρ_k, soit une union de classes de ρ_k :

$$X \in Part\,(\rho_{k+1}) \ \Rightarrow \ \exists X_1, \ldots, X_n \in \rho_k \,.\, X = \bigcup_{i=1}^{n} X_i.$$

■

Les preuves des lemmes 4.7-1 et 4.7-2 sont données dans l'annexe A.

Proposition 4.7-4
Soit $S = (Q, A, T, q_0)$ un système de transitions étiquetées et soit $\sim_\Lambda$ une équivalence de bisimulation. Soit $(\rho_i, W_i)_{(i \in \mathcal{N})}$ la suite définie par :

- $(\rho_0, W_0) = (\{Q\}, \{Q\})$,
- $\forall i \geq 0$, $(\rho_{i+1}, W_{i+1}) = \theta'_\Lambda(\rho_i, W_i)$.

On a alors :

Pour tout $i \geq 0$, ρ_i est la partition associée à la relation $\sim^i_\Lambda$.

■

Preuve : C'est une conséquence de la proposition 4.7-2 et des lemmes 4.7-1 et 4.7-2. □

On donne alors l'algorithme de la nouvelle procédure de décision pour la bisimulation forte $\sim_\Lambda$, obtenue à partir de l'algorithme général 4-1 en utilisant l'opérateur $\theta"_\Lambda$, et qui permet de construire des séquences diagnostiques minimales pour la relation $<_\cap$:

Algorithme 4-5

Soient ρ une partition sur un ensemble Q,

W_1 et W_2 des ensembles de partitionneurs (blocs simples ou blocs arbres),
i, k_0 et k_m des entiers.
début
$(\rho, W_1, W_2) := (\{Q\}, \{Q\}, \emptyset)$;
$\sim^0_\Lambda := \rho$;
$i := 0$;
répéter
$i := i + 1$;
répéter (* calcul de $\theta'_\Lambda(\rho, W_1)$ *) **(1)**
choisir x **dans** W_1 ;
$(\rho, W_1, W_2) := \theta"_\Lambda(\rho, W_1, W_2, x)$
jusqua $W_1 = \emptyset$;
$\sim^i_\Lambda := \rho$;
si $\nexists B \in \rho \,.\, \{q_{01}, q_{02}\} \subseteq B$ **alors**
$k_0 :=$ i
fsi ;
$W_1 := W_2$;
$W_2 := \emptyset$
jusqua $W_1 = \emptyset$;
$k_m := i$;
si $\exists B \in \rho \,.\, \{q_{01}, q_{02}\} \subseteq B$ **alors**
afficher $(S_1 \sim_\Lambda S_2)$
sinon
afficher $(S_1 \not\sim_\Lambda S_2)$;
construire_séquences_diagnostiques (S_1, S_2, $\{\sim^k_\Lambda\}$, k_0, k_m)
fsi.
fin.

■

Avant de préciser la complexité de ce nouvel algorithme, on propose tout d'abord une structure de données adéquate pour représenter les relations $\sim^k_\Lambda$:

Après chaque application de l'opérateur θ'_Λ (**(1)** dans l'algorithme), il est possible de représenter la partition courante ρ_i par un arbre $\mathcal{A}$ dont les feuilles correspondent aux éléments de ρ_i (i.e., les classes d'équivalence des relations $\sim^i_\Lambda$). Plus précisément, cette arbre est construit de la façon suivante :

- A la partition initiale ρ_0 est associé l'arbre $\mathcal{A}(0)$ constitué d'un nœud unique représentant la partition universelle.
- Pour tout $i \geq 0$, l'arbre $\mathcal{A}(i+1)$ associé à la partition ρ_{i+1} est obtenu à partir de $\mathcal{A}(i)$ en remplaçant chacune de ses feuilles (i.e., les éléments de ρ_i) qui ont été partitionnées par des nœuds représentant cette décomposition (i.e., dont les fils représentent les classes obtenues lors du partitionnement).

Pour garder la trace de l'ensemble des partitions obtenues lors de la décomposition de la partition initiale, il suffit alors de *dater* chaque nouveau nœud ajouté à $\mathcal{A}$ en lui associant un attribut indiquant à quelle partition ρ_i les unions des feuilles de chacun de ses fils appartiennent (par exemple en utilisant la variable i définie dans l'algorithme).

La taille mémoire utilisée par $\mathcal{A}$ est de l'ordre du nombre de ses feuilles, soit de l'ordre du nombre de classes d'équivalence p que comporte la partition finale (avec $p \leq n$, où n est le nombre d'états contenus dans Q). Par ailleurs,

- d'une part, lorsque la partition initiale est la partition universelle, nous avons vu que le nombre total de raffinements effectués en utilisant l'opérateur θ'_Λ et identiques à celui obtenu à l'aide de l'opérateur Φ'_Λ (*cf.* remarque 4-4),
- d'autre part, la construction d'un nouveau nœud dans l'arbre $\mathcal{A}$ peut être effectué en un temps qui est de l'ordre de celui nécessaire pour partitionner une classe.

Par conséquent, sans tenir compte de la procédure *construire_séquences_diagnostiques* qui est discuté dans la section suivante, les complexités en temps et en mémoire de l'algorithme 4-5 sont identiques à celles de la procédure de décision classique, soit $O(m.log(n))$ en temps et $O(m)$ en mémoire (où m et n représentent respectivement les nombres d'états et de transitions du système union).

4.7.3 Algorithme de construction des séquences minimales

On considère les deux systèmes de transitions étiquetées $S_i = (Q_i, A_i, T_i, q_{0i})_{i=(1,2)}$, l'équivalence de bisimulation $\sim_\Lambda$, et $S = (Q, A, T, (q_{01}, q_{02}))$ le produit synchrone $S_1 \times_\Lambda S_2$. Soit $\{\rho_k\}$ l'ensemble des partitions associées aux relations $\sim_\Lambda^k$, obtenues comme indiqué dans la section précédente.

L'algorithme que nous proposons consiste à construire les séquences appartenant aux ensembles $D_\Lambda^k(S_1, S_2)$ pour k croissant de k_0 à k_m en appliquant, pour k fixé la procédure informelle suivante :

> Soit σ une séquence diagnostique en cours de construction, et soit i l'indice du dernier état ajouté à σ, avec $\sigma(i) = (q_1, q_2)$.
>
> - Si $i = k$, alors (q_1, q_2) termine la séquence diagnostique σ.
> - Si $i \neq k$, alors on cherche un couple $(\lambda, (q'_1, q'_2))$ dans $\mathrm{Fail}_\Lambda^{k-i}(q_1, q_2)$, tel que d'une part (q'_1, q'_2) n'appartienne à aucune séquence de longueur inférieure à k déjà construite, et que d'autre part q_1 et q_2 soient équivalents modulo $\sim_\Lambda^{k-i-1}$. Si un tel couple existe, on l'ajoute à σ et on réapplique cette procédure à la nouvelle séquence ainsi obtenue, sinon σ n'est pas préfixe d'une séquence diagnostique minimale.

Là encore, cet algorithme peut être mis en œuvre à partir d'un algorithme classique de parcours en profondeur de $S_1 \times_\Lambda S_2$. Néanmoins, il est possible d'améliorer l'efficacité de l'implémentation obtenue en tenant compte des remarques suivantes :

- D'une part, les séquences parcourues sont nécessairement élémentaires (proposition 4.4-1), et il est donc inutile d'utiliser une pile pour détecter si un nouvel état appartient ou non à la séquence courante.
- D'autre part, du fait que la longueur des séquences qui vont être parcourues soit connue à l'avance (elle varie entre k_0 et k_m), il est possible d'optimiser la taille des structures de données utilisées pour les mémoriser. En particulier, les états (*resp.* les actions) de ces séquences seront mémorisés dans un tableau *State* (*resp. Label*) à k_m entrées. On utilisera également un ensemble M, pour marquer les états de $Q_1 \times Q_2$ qui ont été déjà visités ($|M|$ est donc théoriquement majoré par $|Q_1 \times Q_2|$).

On obtient ainsi une version récursive de l'algorithme de construction des séquences diagnostiques minimales pour la relation $<_\cap$:

Algorithme 4-6

procédure construire_séquences_diagnostiques (S_1, S_2, $\{\rho_k\}$)
début
  pour k allant de k_0 à k_{max} **faire** **(1)**

$Label[k] := \emptyset$;
$State[k] := (q_{01}, q_{02})$;
construire_séquence $((q_1, q_2), k, k)$
fpour
fin.

procédure construire_séquence $((q_1, q_2), i, k)$
début
si $i = 1$ **alors**
(* *State* et *Label* contiennent les états et les actions
d'une séquence diagnostique minimale *)
pour j allant de 1 à k **faire**
$M := M \cup State$;
afficher *State* et *Label*
sinon
$succ := \{(\lambda, (q_1', q_2')) \mid (q_1', q_2') \in \mathrm{Fail}^i_\Lambda(q_1, q_2) \wedge q_1' \sim^{i-2}_\Lambda q_2' \wedge (q_1', q_2') \notin M\}$; **(2)**
pour tout $(\lambda, (q_1', q_2'))$ **dans** *succ* **faire** **(3)**
$Label[i-1] := \lambda$;
$State[i-1] := (q_1', q_2')$;
construire_séquence $((q_1', q_2'), i-1, k)$
fpour
fsi
fin.

■

Il reste alors à expliciter le calcul de l'ensemble *succ* pour un état (q_1, q_2) donné (**(2)** dans l'algorithme). En pratique, par rapport à l'algorithme 4-2, la seule difficulté consiste en fait à décider si, pour un successeur (q_1', q_2') de (q_1, q_2), les états q_1' et q_2' satisfont, pour un i fixé :

$q_1' \not\sim^i_\Lambda q_2' \wedge q_1' \sim^{i-1}_\Lambda q_2'$ **(A)**

Pour ce faire, on utilise d'une part une fonction *inclasse*, qui, pour tout état, retourne le numéro de la classe qui le contient dans la partition finale ρ_{k_m}, et, d'autre part, l'arbre $\mathcal{A}$ défini dans la section précédente, en procédant de la manière suivante :

Deux états q_1 et q_2 satisfont **(A)** si et seulement si il existe un nœud n de $\mathcal{A}$ tel que

- l'attribut associé à n soit égal à i (i.e, n représente la décomposition d'une classe de $\sim^{i-1}_\Lambda$)
- $inclasse(q_1)$ et $inclasse(q_2)$ appartiennent à l'union des feuilles de deux fils *distincts* de n (i.e, q_1 et q_2 sont non-équivalents modulo $\sim^i_\Lambda$).

Par suite, cette vérification peut se faire en un parcours de $\mathcal{A}$, soit en un temps de l'ordre de $O(p)$.

Par conséquent, les coûts théoriques en temps et en mémoire de l'algorithme 4-6 pour rechercher les séquences diagnostiques sont respectivement de $O(p.c^{k_m})$ et $O(n)$, où c est le facteur de branchement de $S_1 \times_\Lambda S_2$ et n le nombre de ses états. Toutefois, là encore il est préférable d'exprimer ces valeurs en fonctions du nombre s de séquences effectivement calculées, ce qui donne un coût en temps de $O(s.p.k_m)$ (k_m étant la longueur maximale d'une séquence diagnostique).

Remarque 4-5
k_m est toujours inférieur ou égal à la longueur de la plus longue séquence élémentaire d de $S_1 \times_\Lambda S_2$.

■

Enfin, un certain nombre de variantes sont envisageables pour l'algorithme 4-6 :

- En remplaçant l'itération **(1)** par l'instruction $\{k := k_0\}$, on obtient l'ensemble des séquences diagnostiques minimales pour la relation $<_l$.
- En remplaçant l'itération **(3)** par l'instruction $\{$**choisir** $(\lambda,(q'_1,q'_2))$ **dans** $succ\}$, on ne calcule plus *toutes* les séquences minimales pour la relation $<_\cap$, mais *une* séquence minimale pour chaque entier k. Le coût en temps de ce nouvel algorithme dans le cas le plus défavorable est alors de $O(p.k_m^2)$.

4.8 Préordres et équivalences de simulation

Nous montrons dans cette section que les séquences diagnostiques définies dans ce chapitre pour expliquer la non-équivalence entre deux systèmes de transitions étiquetées modulo une équivalence de bisimulation peuvent également être étendues aux relations de simulation, et en particulier à l'équivalence de sûreté.

4.8.1 Préordres de simulation

Les relations de simulation et de bisimulation ayant des "structures" identiques, la plupart des résultats obtenus du point de vue du diagnostic dans le cas d'une équivalence de bisimulation $\sim_\Lambda$ peuvent être directement transposés au préordre de simulation $\sqsubseteq_\Lambda$. En particulier, à tout couple de systèmes de transitions étiquetées S_1 et S_2 tels que $S_1 \not\sqsubseteq_\Lambda S_2$ peut être associé un ensemble non vide (noté $Diag_\Lambda(S_1,S_2)$) de séquences d'exécution de $S_1 \times_\Lambda S_2$.

De manière similaire à ce qui a été fait dans la section 4.3, nous donnons une définition de $Diag_\Lambda(S_1,S_2)$ obtenue à partir des relations $\sqsubseteq^k_\Lambda$.

Proposition 4.8-1
Soient $S_i = (Q_1, A_i, T_i, q_{0i})_{(i=1,2)}$ deux systèmes de transitions étiquetées, et soient $\sqsubseteq_\Lambda$ un préordre de simulation défini sur un ensemble de langages Λ. On a :

(i) $q_1 \not\sqsubseteq^1_\Lambda q_2 \;-\; act_\Lambda(q_1) \not\subseteq act_\Lambda(q_2)$

(ii) $\forall k > 1 \,.\, q_1 \not\sqsubseteq^k_\Lambda q_2 \;-$
$\quad (\exists \lambda \in \Lambda \,.\, \exists q'_1 \,.\, (q_1 \xrightarrow{\lambda}_{T_1} q'_1 \;\wedge\; \forall q"_2 \,.\, q_2 \xrightarrow{\lambda}_{T_2} q"_2 \;\Rightarrow\; q'_1 \not\sqsubseteq^{k-1}_\Lambda q"_2))$

■

Preuve : C'est une conséquence immédiate de la définition de $\sqsubseteq^k_\Lambda$. □

De même que dans le cas des équivalences de bisimulation, on définit l'ensemble Fail^k_Λ pour tout états (q_1,q_2) de $S_1 \times_\Lambda S_2$:

Définition 4.8-1
Soient $S_i = (Q_i, A_i, T_i, q_{0i})_{(i=1,2)}$ deux systèmes de transitions étiquetées et soit le préordre de simulation $\sqsubseteq_\Lambda$. Pour tout $(q_1,q_2) \in Q_1 \times Q_2$, et pour tout $k > 1$, on définit les ensembles

$$\mathrm{Fail1}^k_\lambda(q_1,q_2) = \{q'_1 \mid q_1 \xrightarrow{\lambda}_{T_1} q'_1 \;\wedge\; \forall q"_2 \,.\, (q_2 \xrightarrow{\lambda}_{T_2} q"_2 \;\Rightarrow\; q'_1 \not\sqsubseteq^{k-1}_\Lambda q"_2)\},$$

et

$$\mathrm{Fail}^k_\lambda(q_1,q_2) = (\mathrm{Fail1}^k_\lambda(q_1,q_2) \;\times\; T_{2_\lambda}[q_2]).$$

■

On peut alors établir un lemme identique au lemme 4.3-1. Les séquences diagnostiques pour le préordre de simulation $\sqsubseteq_\Lambda$ sont alors définies de la façon suivante :

Définition 4.8-2
Soient S_1 et S_2 deux systèmes de transitions étiquetées, soit $S = (Q, A, T, q)$ leur produit synchrone, et soit $\sqsubseteq_\Lambda$ un préordre de simulation. Pour tout entier $k \geq 1$, et pour tout $(p_0, q_0) \in Q$ tel que $p_0 \not\sqsubseteq^k_\Lambda q_0$, l'ensemble ${Diag_\Lambda}^k$ est défini par :

$$\begin{aligned} {Diag_\Lambda}^k(p_0,q_0) \quad = \quad & \{\sigma \in Ex_\Lambda(p_0,q_0) \mid \sigma = ((p_0,q_0),\lambda_0,(p_1,q_1),\ldots,\lambda_{k-2},(p_{k-1},q_{k-1})) \\ & \wedge\, \forall i,\ 0 \leq i < k-1, \exists n_i \,.\, 1 \leq n_i \leq k \,.\, (p_{i+1},q_{i+1}) \in \mathrm{Fail}^{n_i}_{\lambda_i}(p_i,q_i) \\ & \wedge\, p_{k-1} \not\sqsubseteq^1_\Lambda q_{k-1}\} \end{aligned}$$

■

On a alors la proposition suivante, identique au corollaire 4.3-1 :

Proposition 4.8-2
Soient S_1 et S_2, deux systèmes de transitions étiquetées, $S = S_1 \times_\Lambda S_2$, et $\sqsubseteq_\Lambda$ une équivalence de bisimulation. Pour tout état (p_0, q_0) de S, on a :

$$p_0 \not\sqsubseteq_\Lambda q_0 \ \wedge\ p_0 \sqsubseteq^1_\Lambda q_0 \ - \ \exists k \,.\, {Diag_\Lambda}^k(p_0,q_0) \neq \emptyset$$

■

A partir de cette définition générale, comme dans le cas de l'équivalence de bisimulation, il est possible de définir divers sous-ensembles de séquences minimales, en utilisant par exemple les relations d'ordre $<_l$, $<_p$ ou $<_\cap$. Ainsi, dans le cas de la relation $<_p$, l'algorithme proposé pour construire des séquences minimales pour une équivalence de bisimulation (algorithme 4-2) peut directement être étendu au cas des préordres de simulation.

4.8.2 Equivalences de simulation

Du fait qu'une équivalence de simulation $\approx_\Lambda$ est définie comme l'intersection de deux préordres de simulation ($\approx_\Lambda = \sqsubseteq_\Lambda \cap (\sqsubseteq_\Lambda)^{-1}$), expliquer pourquoi deux systèmes de transitions étiquetées ne sont pas équivalents modulo $\approx_\Lambda$ revient à expliquer pourquoi ils ne sont pas en relation modulo l'un ou l'autre de ces deux préordres.

Toutefois, dans le cas de l'équivalence de sûreté $\approx_s$, nous avons décrit au chapitre 3 (section 3.5) une procédure de décision qui consiste à comparer modulo la bisimulation forte des systèmes de transitions dépourvues de *transitions redondantes*. Plus formellement, nous avons défini pour tout systèmes de transitions étiquetées S, une forme pré-normale pour l'équivalence de sûreté, notée $\mathrm{PNF}_s(S)$, telle que :

$$S_1 \approx_s S_2 \ - \ \mathrm{PNF}_s(S_1) \sim \mathrm{PNF}_s(S_2).$$

Par conséquent, conformément au cas des bisimulations faibles, construire des séquences diagnostiques pour l'équivalence de sûreté revient à construire les séquences diagnostiques obtenues en comparant leur formes pré-normales modulo la bisimulation forte.

4.9 Discussion

Nous avons présenté dans ce chapitre une méthode de diagnostic pour la vérification de spécifications comportementales à l'aide de relations de simulation ou de bisimulation qui consiste à exhiber

un ensemble de séquences d'exécutions *erronées* du programme. Plus précisément, lorsque les deux systèmes de transitions étiquetées représentant le programme et sa spécification ne sont pas équivalents, le diagnostic obtenu est un ensemble de séquences d'exécution du produit synchrone de ces deux systèmes, éventuellement restreint à ses éléments minimaux modulo une relation d'ordre sur les séquences. De plus, pour trois relations d'ordre particulières, des algorithmes de génération de séquences diagnostiques minimales ont été proposés, dont les principales caractéristiques sont, d'une part, de ne pas mémoriser l'ensemble des séquences obtenues, et, d'autre part, de ne pas modifier les complexités des algorithmes de vérification dans le cas où les deux systèmes sont équivalents.

L'un de ces algorithmes, dédié à la relation d'ordre $<_\cap$ (algorithme 4-6), a été implémenté dans le cadre de l'outil de vérification ALDÉBARAN, et il peut être appliqué à l'ensemble des relations d'équivalence actuellement prises en compte par ce logiciel. Ainsi, l'utilisation d'ALDÉBARAN (en conjonction avec le compilateur CÆSAR) lors de la vérification de programmes LOTOS de tailles réalistes a pu montrer en pratique l'intérêt de cette approche pour le diagnostic :

- En premier lieu, il apparaît que, dans le cas d'exemples réels, la taille et le nombre des séquences minimales pour la relation $<_\cap$ restent raisonnables, et par conséquent celles-ci sont exploitables par un utilisateur.

- De plus, en précisant exactement quels sont les actions observables effectuées et les numéros des états parcourus avant d'atteindre un couple d'états "erreur", ce type de diagnostic permet de renforcer la compréhension du programme et de localiser la cause de l'erreur.

- Enfin, du point de vue du coût, la phase de génération du diagnostic s'est toujours avérée négligeable devant la phase de vérification proprement dite.

Néanmoins, si le choix des séquences d'exécution semble être un bon formalisme pour générer un diagnostic à partir des algorithmes de vérification, un certain nombre d'améliorations pourraient encore être apportées au prototype réalisé, notamment en ce qui concerne le lien entre les séquences diagnostiques obtenues et le programme source :

- D'une part, dans le cas des bisimulations faibles et de l'équivalence de sûreté, nous avons choisi de construire les séquences diagnostiques à partir des formes pré-normales des systèmes de transitions étiquetées. Si ce choix offre l'intérêt de ne pas modifier les algorithmes de vérification et d'unifier la phase de génération du diagnostic pour toutes les relations d'équivalence, il rend également plus difficile le lien avec le programme source puisque seules les états atteints par une action observable apparaissent dans le diagnostic. Une solution possible pour retrouver les états intermédiaires et les actions internes présents entre deux actions observables consisterait à reparcourir le système de transition original en se guidant à l'aide des séquences diagnostiques calculées sur la forme pré-normale.

- D'autre part, il serait également souhaitable de pouvoir enrichir les séquences d'exécutions obtenues en ajoutant sur les états et les actions des informations produites par l'outil utilisé pour la génération du système de transition représentant le programme (comme les valeurs des variables d'états, les noms des actions internes avant le renommage en τ, la ligne du programme source où elles sont déclarées, ...). En particulier, ces modifications seraient envisageables dans le cas des systèmes de transition produits par CÆSAR, puisque toutes ces informations sont disponibles en sortie de ce compilateur.

Enfin, d'autres améliorations pourraient également êtres envisagées, comme le choix interactif des séquences diagnostiques à afficher (qui serait facile à mettre en œuvre du fait que les séquences sont générées en profondeur d'abord), où l'étude d'autres relations d'ordres pour construire des séquences minimales (par exemple en identifiant les séquences qui conduisent au même état "erreur" et qui peuvent se déduire l'une de l'autre par permutation de certaines de leurs actions).

Partie III

Vérification “à la volée”

Chapitre 5

Un algorithme de comparaison "à la volée"

Nous présentons dans ce chapitre un nouvel algorithme pour comparer deux systèmes de transitions étiquetées modulo une relation de simulation ou de bisimulation. Cet algorithme a été conçu de manière à satisfaire simultanément deux contraintes :

- Il doit pouvoir être mise en œuvre au fur et à mesure de la génération du système de transition étiquetées représentant le programme à vérifier (*vérification "à la volée"*), sans construire explicitement sa relation de transition.
- Son coût en mémoire doit rester raisonnable en pratique et permettre ainsi la comparaison de systèmes de transition étiquetées de tailles réalistes.

Nous montrons tout d'abord comment l'existence d'une relation de bisimulation entre deux systèmes de transitions étiquetées S_1 et S_2 peut être exprimée comme une propriété sur l'ensemble des séquences d'exécution élémentaires de leur produit synchrone S, qui soit décidable lors d'un parcours en profondeur de ce produit. On obtient ainsi un premier algorithme de vérification basé sur une énumération exhaustive de l'ensemble des séquences d'exécution élémentaires de S et dont la complexité est par conséquent exponentielle en fonction du nombre d'états de ce produit.

Nous proposons alors deux améliorations, qui, toujours sur la base d'un parcours en profondeur de S, permettent de réduire la complexité en temps de cet algorithme à $O(m)$ lorsqu'au moins l'un des deux systèmes à comparer est déterministe, et à $O(m.n)$ dans le cas général (i.e., lorsque les deux systèmes sont non déterministes), où n et m représentent respectivement les nombres d'états et de transitions de S.

Enfin, nous montrons comment cette approche peut être étendue au cas des préordres et équivalences de simulation et nous terminons en indiquant comment le calcul de *séquences diagnostiques* peut être adapté à ces nouvelles procédures de décision.

5.1 Critère de bisimulation sur les séquences d'exécutions

Nous décrivons dans cette section le principe de l'algorithme de vérification "à la volée", qui consiste à exprimer l'existence d'une relation de bisimulation entre deux systèmes de transitions étiquetées S_1 et S_2 à l'aide d'un critère sur l'ensemble des séquences d'exécution élémentaires de leur produit

synchrone.

On propose tout d'abord un premier critère, obtenu en considérant l'ensemble de *toutes* les séquences d'exécutions de $S_1 \times_\Lambda S_2$. On montre alors qu'il est possible de restreindre ce critère aux séquences d'exécution *élémentaires*.

5.1.1 Equivalence de bisimulation entre séquences d'exécution

Définition 5.1-1
Soient $S_i = (Q_i, A_i, T_i, q_{0i})_{(i=1,2)}$ deux systèmes de transitions étiquetées, soit Λ un ensemble de langages sur $\mathcal{A}$ et soit $S = (Q, A, T, (q_{01}, q_{02}))$ le produit synchrone $S = S_1 \times_\Lambda S_2$.

On note Σ_Λ, la fonction sur l'ensemble des parties de $Ex_\Lambda(S)$ définie par :

$$\Sigma_\Lambda \ : \ 2^{Ex_\Lambda(S)} \longrightarrow 2^{Ex_\Lambda(S)}$$

et :

$$\begin{aligned}\Sigma_\Lambda(X) \ = \ & \{\sigma \in Ex_\Lambda(S) \mid last(\sigma) = (p,q) \ \Rightarrow \\ & \forall \lambda \ . \ \forall p' \ . \ p \xrightarrow{\lambda}_{T_1} p' \ \Rightarrow \ \exists q' \ . \ q \xrightarrow{\lambda}_{T_2} q' \ \wedge \ (\sigma, \lambda, (p', q')) \in X \\ & \wedge \\ & \forall \lambda \ . \ \forall q' \ . \ q \xrightarrow{\lambda}_{T_2} q' \ \Rightarrow \ \exists p' \ . \ p \xrightarrow{\lambda}_{T_1} p' \ \wedge \ (\sigma, \lambda, (p', q')) \in X\}\end{aligned}$$

On note alors E le plus grand point fixe de la fonction Σ_Λ :

$$E \ = \ \nu X. \, (Ex_\Lambda(S_1 \times S_2) \ \cap \ \Sigma_\Lambda(X))$$

■

Il reste à établir le lien entre la fonction Σ_Λ et l'équivalence de bisimulation $\sim_\Lambda$:

Lemme 5.1-1
Soient S_1 et S_2 deux systèmes de transitions étiquetées, soit $\sim_\Lambda$ une équivalence de bisimulation et soit S le produit synchrone $S_1 \times_\Lambda S_2$. Soit E le plus grand sous-ensemble de $Ex_\Lambda(S)$ satisfaisant $E \subseteq \Sigma_\Lambda(E)$.

On a alors :

$$\forall (p_0, q_0) \in Q \ . \ p_0 \sim_\Lambda q_0 \ - \ \forall \sigma \in Ex_\Lambda(S) \ . \ last(\sigma) = (p_0, q_0) \ \Rightarrow \ \sigma \in E$$

■

Preuve :

sens "⇒" : Pour toute séquence $\sigma = (\sigma', \lambda, (p_0, q_0))$ telle que $last(\sigma) = (p_0, q_0)$, on note E' l'ensemble défini par :

$$E' = \{(\sigma, \lambda, (p_0, q_0), \lambda_0, (p_1, q_1), \ldots, \lambda_{n-1}, (p_n, q_n)) \in Ex_\Lambda(S) \mid \forall i, \, 0 \leq i \leq n \ . \ p_i \sim_\Lambda q_i\}$$

Alors :

- la séquence $(\sigma, \lambda, (p_0, q_0))$ appartient à E'.
- en notant $(E^k)_{(k \in \mathcal{N})}$ la suite définie par :

$$\begin{aligned} E^0 \ &= \ Ex_\Lambda(S) \\ \forall k \geq 0 \ . \ E^{k+1} \ &= \ \Sigma_\Lambda(E^k), \end{aligned}$$

on peut montrer par induction sur k :

$$\forall k \geq 0 \ . \ last(\sigma) = (p,q) \ \wedge \ p \sim_\Lambda q \ \Rightarrow \ \sigma \in E^k.$$

D'où, $E' \subseteq \Sigma_\Lambda(E)$.

Par suite, on a bien $E' \subseteq E$ et $\sigma \in E$.

sens "$\Leftarrow$" : Soit R la relation définie par :

$$R = \{(p,q) \mid \forall \sigma \in Ex_\Lambda(S) \,.\, last(\sigma) = (p,q) \;\Rightarrow\; \sigma \in E\}.$$

D'après la définition de Σ, on a $R \subseteq \mathcal{B}_\Lambda(R)$ (*cf.* chapitre 2, définition 2.1-1) et, par suite, la relation R est bien une bisimulation. On a donc $p_0 \sim_\Lambda q_0$.

□.

Le lemme 5.1-1 fournit un premier critère sur les séquences d'exécutions de $S_1 \times_\Lambda S_2$ pour décider si deux états se bisimulent. Plus précisément :

- On note R^Σ_Λ la relation binaire constituée des couples (p,q) de $S_1 \times S_2$ qui terminent une séquence de $Ex_\Lambda(S_1 \times S_2)$ contenue dans E.
- On montre alors (proposition 5.1-1) que R^Σ_Λ coïncide avec l'équivalence de bisimulation restreinte à l'ensemble des états de $S_1 \times S_2$.

Définition 5.1-2
Soient S_1 et S_2 deux systèmes de transitions étiquetées, soit S le produit synchrone $S_1 \times_\Lambda S_2$, avec $S = (Q, A, T, (p_0, q_0))$, et soit E le plus grand sous-ensemble de $Ex_\Lambda(S)$ satisfaisant $E \subseteq \Sigma_\Lambda(E)$.

La relation R^Σ_Λ, incluse dans $Q_1 \times Q_2$ est alors définie de la manière suivante :

$$R^\Sigma_\Lambda = \{(p,q) \in Q \mid \exists \sigma \,.\, last(\sigma) = (p,q) \;\wedge\; \sigma \in E\}$$

■

Proposition 5.1-1
Soient S_1 et S_2 deux systèmes de transitions étiquetées, soit $\sim_\Lambda$ une équivalence de bisimulation et soit S le produit synchrone $S_1 \times_\Lambda S_2$ avec $S = (Q, A, T, (q_{01}, q_{02}))$.

On a alors :

$$R^\Sigma_\Lambda = (\sim_\Lambda \cap Q)$$

■

Preuve : Compte-tenu du lemme 5.1-1, il suffit de montrer :

$$\forall (p,q) \in Q \,.\, (\forall \sigma \,.\, last(\sigma) = (p,q) \;\wedge\; \sigma \in E) \;\Rightarrow\; (\exists \sigma' \,.\, last(\sigma') = (p,q) \;\Rightarrow\; \sigma' \in E).$$

Or, par définition le produit $S_1 \times_\Lambda S_2$ ne contient pas d'états inaccessibles. Donc :

$$\forall (p,q) \,.\, (p,q) \in Q \;\Rightarrow\; \exists \sigma \in Ex_\Lambda(S) \,.\, last(\sigma) = (p,q),$$

ce qui termine la preuve de la proposition.
□.

Remarque 5-1
La relation R^Σ_Λ définit en fait une équivalence de bisimulation $\sim^*_\Lambda$ entre les *séquences d'exécution* de S_1 et S_2 :

$$(p_0, \lambda_0, \ldots, \lambda_{n-1}, p_n) \sim^*_\Lambda (q_0, \lambda_0, \ldots, \lambda_{n-1}, q_n) \;-\; (p_n, q_n) \in R^\Sigma_\Lambda$$

Une autre définition de relation de bisimulation entre séquences d'exécution est proposée dans [NMV90]. ■

5.1.2 Restriction aux séquences élémentaires

Bien que le critère établi dans la proposition 5.1-1 permette de décider si les états p et q d'un *couple quelconque* (p,q) de $S_1 \times_\Lambda S_2$ se bisimulent ou non, son application reste néanmoins difficile puisqu'il

repose sur l'examen de toutes les séquences d'exécution de $S_1 \times_\Lambda S_2$, dont certaines peuvent être de longueur infinie. Dans la suite, on s'intéresse à un critère plus faible, qui peut être vérifié en ne prenant en compte que *les séquences élémentaires* de $S_1 \times_\Lambda S_2$ et qui permet de décider si *les états initiaux* de ces deux systèmes se bisimulent.

Ce critère est obtenu à partir d'une nouvelle fonction, Σ^e_Λ, définie sur l'ensemble des séquences d'exécutions élémentaires de $S_1 \times_\Lambda S_2$: les états initiaux q_{01} et q_{02} de S_1 et S_2 se bisimulent si et seulement si la séquence d'exécution élémentaire $((q_{01}, q_{02}))$ de $S_1 \times_\Lambda S_2$ appartient à un ensemble E^e inclus dans $\Sigma^e_\Lambda(E^e)$.

Définition 5.1-3
Soient $S_i = (Q_i, A_i, T_i, q_{0i})_{(i=1,2)}$ deux systèmes de transitions étiquetées, soit Λ un ensemble de langages sur $\mathcal{A}$ et soit S le produit synchrone $S = S_1 \times_\Lambda S_2$.

Soit $\Sigma^e_\Lambda : 2^{Ex_\Lambda(S)} \longrightarrow 2^{Ex_\Lambda(S)}$ définie par :

$$\begin{aligned}\Sigma^e_\Lambda(X) \;=\; & \{\sigma \in Ex_\Lambda(S) \mid last(\sigma) = (p,q) \;\Rightarrow \\ & \forall \lambda \,.\, \forall p' \,.\, p \xrightarrow{\lambda}_{T_1} p' \;\Rightarrow\; \exists q' \,.\, (q \xrightarrow{\lambda}_{T_2} q' \;\wedge\; ((p',q') \in \sigma \;\vee\; (\sigma, \lambda, (p',q')) \in X)) \\ & \wedge \\ & \forall \lambda \,.\, \forall q' \,.\, q \xrightarrow{\lambda}_{T_2} q' \;\Rightarrow\; \exists p' \,.\, (p \xrightarrow{\lambda}_{T_1} p' \;\wedge\; ((p',q') \in \sigma \;\vee\; (\sigma, \lambda, (p',q')) \in X))\}\end{aligned}$$

On note alors E^e le plus grand point fixe de la fonction Σ^e_Λ :

$$E^e \;=\; \nu X.\,(Ex_\Lambda(S_1 \times S_2) \;\cap\; \Sigma^e_\Lambda(X))$$

■

Le lemme suivant permet alors de comparer les fonctions Σ et Σ^e vis-à-vis de la relation d'inclusion :

Lemme 5.1-2
Soit S_1 et S_2 deux systèmes de transitions étiquetées, et soit S le produit synchrone $S_1 \times_\Lambda S_2$. Pour tout état (p,q) de S, et pour tout ensemble X inclus dans $Ex_\Lambda((p,q))$ on a :

$$X \subseteq \Sigma_\Lambda(X) \;\Rightarrow\; X \subseteq \Sigma^e_\Lambda(X)$$

On a donc en particulier :

$$E \subseteq E^e$$

■

Preuve : Par définition des fonctions Σ_Λ et Σ^e_Λ.

□

Pour établir le lien entre la fonction Σ^e_Λ et l'équivalence de bisimulation $\sim_\Lambda$, on définit tout d'abord un ensemble K de séquences d'exécutions de $S_1 \times_\Lambda S_2$ qui n'appartiennent à aucune partie E^e de $Ex_\Lambda(S_1 \times_\Lambda S_2)$ telle que $E^e \subseteq \Sigma^e_\Lambda(E^e)$. On montre alors dans la proposition 5.1-2 que $S_1 \sim_\Lambda S_2$ si et seulement si la séquence $((q_{01}, q_{02}))$ n'appartient pas à K.

La preuve de cette proposition reposera sur le lemme suivant :

Lemme 5.1-3
Soient S_1 et S_2 deux systèmes de transitions étiquetées, et soit $S = S_1 \times_\Lambda S_2$ leur produit synchrone. On note E^e le plus grand ensemble inclus dans $Ex_\Lambda(S_1 \times_\Lambda S_2)$ tel que $E^e \subseteq \Sigma^e_\Lambda(E^e)$.

Pour toute séquence $\sigma \in Ex_\Lambda(S)$ telle que $last(\sigma) = (p,q)$, on a :

$$\sigma \notin E^e \;-\; (p \not\sim^1_\Lambda q) \vee$$

$$(\exists \lambda \,.\, \exists p' \,.\, p \xrightarrow{\lambda}_{T_1} p' \ \wedge\ \forall q' \,.\, (q \xrightarrow{\lambda}_{T_2} q' \ \Rightarrow\ ((p',q') \notin \sigma \ \wedge\ (\sigma,\lambda,(p',q'))) \notin E^e)) \vee$$
$$(\exists \lambda \,.\, \exists p' \,.\, q \xrightarrow{\lambda}_{T_2} q' \ \wedge\ \forall p' \,.\, (p \xrightarrow{\lambda}_{T_1} p' \ \Rightarrow\ ((p',q') \notin \sigma \ \wedge\ (\sigma,\lambda,(p',q'))) \notin E^e)).$$

■

Preuve : E^e étant le plus grand sous-ensemble de $Ex_\Lambda(S_1 \times_\Lambda S_2)$ satisfaisant $E^e \subseteq \Sigma^e_\Lambda(E^e)$, on a :

$$\sigma \notin E^e \ -\ \sigma \notin \Sigma^e_\Lambda(E^e)$$

La preuve est alors immédiate en écrivant la négation de $\sigma \in \Sigma^e_\Lambda(E^e)$.
□.

Proposition 5.1-2
Soient S_1 et S_2 deux systèmes de transitions étiquetées d'états initiaux p_1 et q_1, et soit $\sim_\Lambda$ une équivalence de bisimulation définie sur $Q_1 \times Q_2$. Soit E^e le plus grand ensemble inclus dans $Ex_\Lambda(S_1 \times_\Lambda S_2)$ tel que $E^e \subseteq \Sigma^e_\Lambda(E^e)$.

Pour tout entier $k \geq 1$, on note :

$$K(k) \ = \ \{\sigma = ((p_0,q_0),\lambda_0,(p_1,q_1),\ldots,\lambda_{n-1},(p_n,q_n)) \in Ex_\Lambda(S_1 \times_\Lambda S_2) \mid$$
$$\exists\, k_0,k_1,\ldots,k_n \,.\, (\forall i,\ 0 \leq i \leq n,\ k_{i+1} < k_i \ \wedge\ p_i \not\sim^{k_i}_\Lambda q_i \ \wedge\ p_i \sim^{k_i-1}_\Lambda q_i \ \wedge\ k_n = k)\}$$

On a alors :

$$\forall k_n \geq 1, \forall \sigma \,.\, \sigma \in K(k_n) \ \Rightarrow\ \sigma \notin E^e$$

■

Preuve : par induction sur k :

- Pour $k = 1$, on a :

  $last(\sigma) = (p,q) \ \wedge\ p \not\sim^1_\Lambda q.$

  Par suite, d'après le lemme 5.1-3, $\sigma \notin E^e$.

- Soit un entier k fixé. L'hypothèse d'induction s'écrit :

  $$\forall i,\ 1 \leq i \leq k \ \Rightarrow\ (\forall \sigma \,.\, \sigma \in K(i) \ \Rightarrow\ \sigma \notin E^e) \quad \textbf{(1)}$$

  On montre alors :

  $$\forall \sigma \,.\, \sigma \in K(k+1) \ \Rightarrow\ \sigma \notin E^e$$

  Soit $\sigma \in K(k+1)$. Par définition de l'ensemble $K(k)$,

  $$last(\sigma) = (p,q) \ \Rightarrow\ p \not\sim^{k+1}_\Lambda q \ \wedge\ p \sim^k_\Lambda q$$

  Par suite, d'après la proposition 4.3-2 (chapitre 4),

  $$\exists \lambda \,.\, \exists p' \,.\, p \xrightarrow{\lambda}_{T_1} p' \ \wedge\ \forall q' \,.\, q \xrightarrow{\lambda}_{T_2} q' \ \Rightarrow\ \exists k' \,.\, k' < k+1 \ \wedge\ p' \not\sim^{k'}_\Lambda q' \ \wedge\ p' \sim^{k'-1}_\Lambda q'$$

  ou bien

  $$\exists \lambda \,.\, \exists q' \,.\, q \xrightarrow{\lambda}_{T_2} q' \ \wedge\ \forall p' \,.\, p \xrightarrow{\lambda}_{T_1} p' \ \Rightarrow\ \exists k' \,.\, k' < k+1 \ \wedge\ p' \not\sim^{k'}_\Lambda q' \ \wedge\ p' \sim^{k'-1}_\Lambda q'$$

  Par conséquent, on a, pour l'ensemble des couples (p',q') ainsi obtenus :

  - d'une part, $p' \not\sim^{k'}_\Lambda q'$ avec $k' < k+1$. Or,

    $$\sigma \in K(k+1) \ \Rightarrow\ (\forall i \leq n \,.\, \sigma(i) = (p_i,q_i) \ \Rightarrow\ \exists k_i \geq k+1 \,.\, p_i \sim^{k_i-1}_\Lambda q_i)$$

    d'où nécessairement :

    $$(p',q') \notin \sigma \quad \textbf{(2)}$$

– d'autre part, en notant σ' la séquence $(\sigma, \lambda, (p', q'))$, on a :

$$\sigma' \in K(k') \ \wedge \ k' < k+1$$

d'où, par hypothèse d'induction **(1)**, on déduit :

$$\sigma' \notin E^e \quad \textbf{(3)}$$

On a alors $last(\sigma) = (p, q)$ et, d'après **(2)** et **(3)**,

$$\exists \lambda \,.\, \exists p' \,.\, p \xrightarrow{\lambda}_{T_1} p' \ \wedge \ \forall q' \,.\, (q \xrightarrow{\lambda}_{T_2} q' \ \Rightarrow \ ((p', q') \notin \sigma \ \wedge \ (\sigma, \lambda, (p', q')) \notin E^e))$$

ou

$$\exists \lambda \,.\, \exists p' \,.\, q \xrightarrow{\lambda}_{T_2} q' \ \wedge \ \forall p' \,.\, (p \xrightarrow{\lambda}_{T_1} p' \ \Rightarrow \ ((p', q') \notin \sigma \ \wedge \ (\sigma, \lambda, (p', q')) \notin E^e)).$$

Par conséquent, en utilisant le lemme 5.1-3, on déduit que $\sigma \notin \Sigma_\Lambda(E^e)$, d'où $\sigma \notin E^e$.

□.

Le corollaire suivant fournit un critère de décision pour l'équivalence de bisimulation. Ce critère repose sur l'ensemble des séquences d'exécutions élémentaires du produit synchrone des systèmes de transitions à comparer :

Corollaire 5.1-1
Soient S_1 et S_2 deux systèmes de transitions étiquetées, d'états initiaux respectifs q_{01} et q_{02}, et soit $\sim_\Lambda$ une équivalence de bisimulation définie sur $Q_1 \times Q_2$. En notant E^e le plus grand sous-ensemble de $Ex_\Lambda(S_1 \times_\Lambda S_2)$ inclus dans $\Sigma^e_\Lambda(E^e)$ on a :

$$S_1 \not\sim S_2 \ - \ ((q_{01}, q_{02})) \notin E^e$$

■

Preuve :

sens "⇒" : Clairement,

$$S_1 \not\sim S_2 \ \Rightarrow \ \exists k \,.\, q_{01} \not\sim^k_\Lambda q_{02} \ \wedge \ q_{01} \sim^{k-1}_\Lambda q_{02}.$$

Par suite, en utilisant la notation introduite dans la proposition 5.1-2, on a $((q_{01}, q_{02})) \in K(k)$, d'où $((q_{01}, q_{02})) \notin E^e$.

sens "⇐" : Soit E le plus grand ensemble inclus dans $Ex_\Lambda(S_1 \times_\Lambda S_2)$ tel que $E \subseteq \Sigma_\Lambda(E)$.

On a :

$$\begin{array}{lcll} ((q_{01}, q_{02})) \notin E^e & \Rightarrow & ((q_{01}, q_{02})) \notin E & \text{(d'après le lemme 5.1-2)} \\ & \Rightarrow & S_1 \not\sim S_2 & \text{(d'après la proposition 5.1-1)} \end{array}$$

□.

5.2 Un premier algorithme de vérification "à la volée"

A partir du corollaire 5.1-1, il est possible de construire un premier algorithme de vérification basé sur un parcours en profondeur du produit synchrone de S_1 et de S_2.

Toutefois, avant de décrire cet algorithme plus en détails, on cherche à restreindre davantage l'ensemble des séquences d'exécution élémentaires de $S1 \times_\Lambda S_2$ prises en compte lors de la vérification. Nous définissons donc un nouveau produit synchrone, noté dans la suite $S_1 \otimes_\Lambda S_2$, et nous montrons que l'examen de l'ensemble des séquences d'exécution élémentaires de ce produit suffit pour décider si les deux systèmes sont ou non équivalents.

5.2.1 Le produit synchrone $S_1 \otimes_\Lambda S_2$

Définition 5.2-1
Soient $S_i = (Q_i, A_i, T_i, q_{0i})_{(i=1,2)}$ deux systèmes de transitions étiquetées, soit Λ un ensemble de langages sur $\mathcal{A}$. On note $S_1 \otimes_\Lambda S_2$, le système de transitions étiquetées $S = (Q, A, T, (q_{01}, q_{02}))$ défini par :

- $Q \subseteq (Q_1 \times Q_2) \cup \{fail\}$, avec $fail \notin (Q_1 \cup Q_2)$,
- $A \subseteq \Lambda \cup \{\phi\}$, avec $\phi \notin (A_1 \cup A_2)$,
- $T \subseteq Q \times A \times Q$

et T et Q sont les plus petits ensembles obtenus par application des règles $R0$, $R1$, et $R2$ suivante :

$$(q_{01}, q_{02}) \in Q \qquad [R0]$$

$$\frac{(q_1, q_2) \in Q,\ \mathcal{A}\mathrm{ct}_\Lambda(q_1) = \mathcal{A}\mathrm{ct}_\Lambda(q_2),\ \lambda \in \Lambda,\ q_1 \xrightarrow{\lambda}_{T_1} q_1',\ q_2 \xrightarrow{\lambda}_{T_2} q_2'}{\{(q_1', q_2')\} \in Q, \{(q_1, q_2) \xrightarrow{\lambda}_T (q_1', q_2')\} \in T} \qquad [R1]$$

$$\frac{(q_1, q_2) \in Q,\ \mathcal{A}\mathrm{ct}_\Lambda(q_1) \neq \mathcal{A}\mathrm{ct}_\Lambda(q_2),}{\{fail\} \in Q, \{(q_1, q_2) \xrightarrow{\phi}_T fail\} \in T} \qquad [R2]$$

■

A partir de cette définition, il est facile de voir que, d'une part, la séquence $((q_{01}, q_{02}))$ appartient à $Ex_\Lambda(S_1 \otimes_\Lambda S_2)$, et que, d'autre part, toute séquence d'exécution de $S_1 \otimes_\Lambda S_2$ est bien une séquence d'exécution de $S_1 \times_\Lambda S_2$:

Proposition 5.2-1
Soient S_1 et S_2 deux systèmes de transitions étiquetées et Λ un ensemble de langages sur $\mathcal{A}$. On a :

(i) $((q_{01}, q_{02})) \in Ex_\Lambda(S_1 \otimes_\Lambda S_2)$

(ii) $Ex_\Lambda(S_1 \otimes_\Lambda S_2) \subseteq Ex_\Lambda(S_1 \times_\Lambda S_2)$

■

La proposition suivante établit que, pour toute séquence σ appartenant à $Ex_\Lambda(S_1 \otimes_\Lambda S_2)$, décider si σ appartient à E^e revient à décider si les séquences élémentaires de $Ex_\Lambda(S_1 \otimes_\Lambda S_2)$ obtenues à partir de l'état final de σ appartiennent ou non à E^e :

Proposition 5.2-2
Soient S_1 et S_2 deux systèmes de transitions étiquetées, d'états initiaux respectifs q_{01} et q_{02}, et soit E^e le plus grand sous-ensemble de $Ex_\Lambda(q_{01} \times_\Lambda q_{02})$ inclus dans $\Sigma^e_\Lambda(E^e)$.

En notant $S = (Q, A, T, q_0)$ le produit $S_1 \otimes_\Lambda S_2$, on a :

(i) $\forall (p,q) \in Q \,.\, p \not\sim^1_\Lambda q \ - \ (p,q) \xrightarrow{\phi}_T fail$

(ii) $\forall \sigma \in Ex_\Lambda(S_1 \times S_2) \,.\, \sigma \in E^e \ \Rightarrow \ \sigma \in Ex_\Lambda(S_1 \otimes S_2)$

■

Preuve :

(i) Pour tout état (p,q) de Q, on a :

$p \not\sim^1_\Lambda q$ – $\mathcal{A}\mathrm{ct}_\Lambda(p) \neq \mathcal{A}\mathrm{ct}_\Lambda(q)$

– $(p,q) \xrightarrow{\phi}_T fail$

(ii) Par définition de S, et en utilisant (i), on a :

$$\forall (p,q) \in Q \;.\; p \sim^1_\Lambda q \;\wedge\; p \xrightarrow{\lambda}_{T_1} p' \;\wedge\; q \xrightarrow{\lambda}_{T_1} q' \;-\; (p,q) \xrightarrow{\lambda}_{T} (p',q')$$

La propriété s'établit alors directement à partir du lemme 5.1-3.

□.

5.2.2 Algorithme

D'après le corollaire 5.1-1, les systèmes de transitions étiquetées S_1 et S_2 d'états initiaux respectifs q_{01} et q_{02} ne sont pas équivalents modulo l'équivalence de bisimulation $\sim_\Lambda$ si et seulement si la séquence d'exécution $((q_{01},q_{02}))$ de $S_1 \times_\Lambda S_2$ appartient à E^e. Or, compte-tenu de la proposition 5.2-2, ce critère peut être vérifié par une énumération de l'ensemble des séquences élémentaires du système $S_1 \otimes_\Lambda S_2$.

On réécrit tout d'abord de manière plus algorithmique la proposition 5.2-2, puis on présente un premier algorithme de vérification.

Proposition 5.2-3
Soient $S_i = (Q_i, A_i, T_i, q_{0i})_{(i=(1,2))}$ deux systèmes de transitions étiquetées, soit S le produit $S_1 \otimes_\Lambda S_2$, avec $S = (Q, A, T, (q_{01}, q_{02}))$.

Pour toute séquence $\sigma \in Ex_\Lambda(S)$, telle $last(\sigma) = (q_1,q_2)$, l'algorithme suivant permet de décider si σ appartient ou non à E^e avec un coût en temps et en mémoire de $O(|T_\lambda[(q_1,q_2)]|)$:

Soit $Equiv\_List$ un sous-ensemble de $T_{1_\Lambda}[q_1] \cup T_{2_\Lambda}[q_2]$.
début
  $Equiv\_List\,(q_1,q_2) := \emptyset$;
  si $\neg((q_1,q_2) \xrightarrow{\phi}_T fail)$ **alors**
    pour tout λ dans Λ
      pour tout (q_1',q_2') dans $T_\lambda[(q_1,q_2)]$
        si $((q_1',q_2') \in \sigma)$ **ou** $((\sigma,\lambda,(q_1',q_2')) \in E^e)$ **alors**
          $Equiv\_List := Equiv\_List \cup \{(\lambda,q_1'),(\lambda,q_2')\}$
        fsi
      fpour
    fpour
  fsi
  si $Equiv\_List = T_{1_\Lambda}[q_1] \cup T_{2_\Lambda}[q_2]$ **alors**
    retourner $(\sigma \in E^e)$
  sinon
    retourner $(\sigma \notin E^e)$
  fsi
fin.

■

L'algorithme de vérification qui s'en déduit repose sur un parcours en profondeur de $S_1 \otimes_\Lambda S_2$: pour chaque séquence élémentaire σ de ce produit, on vérifie si σ appartient ou non à E^e après avoir déterminé si l'ensemble de ses séquences "successeurs" dans le parcours en profondeur appartiennent ou non à E^e. Les structures de données nécessaires sont alors :

- Une pile $StState$, pour mémoriser les états de la séquence en cours de vérification (et assurer ainsi la terminaison de l'algorithme).
- Une pile $StTrans$, pour mémoriser les ensembles de transitions restant à exécuter à partir de

chacun des états de la séquence courante,

- Un ensemble de listes $Equiv\_List$ (*cf.* proposition 5.2-3) tel que, pour tout état (q_1, q_2) appartenant à la séquence courante :

$$Equiv\_List\,(q_1, q_2) \subseteq T_{1_\Lambda}[q_1] \cup T_{2_\Lambda}[q_2].$$

Enfin, on utilisera également la fonction "successeur" $succ\_produit_\Lambda$ pour représenter la relation de transition de $S_1 \otimes_\Lambda S_2$ (définition 5.2-1) :

$$\mathcal{A}\mathrm{ct}_\Lambda(q_1) = \mathcal{A}\mathrm{ct}_\Lambda(q_2) \quad \Rightarrow \quad succ\_produit_\Lambda\,(q_1, q_2) = \{(\lambda, (q_1', q_2')) \mid (q_1 \xrightarrow{\lambda}_{T_1} q_1' \ \wedge \ q_2 \xrightarrow{\lambda}_{T_2} q_2')\}$$
$$\mathcal{A}\mathrm{ct}_\Lambda(q_1) \neq \mathcal{A}\mathrm{ct}_\Lambda(q_2) \quad \Rightarrow \quad succ\_produit_\Lambda\,(q_1, q_2) = \{(\phi, fail)\}$$

Le calcul de cette fonction pour différents ensembles de langages Λ sera explicité dans le chapitre 6.

On obtient alors l'algorithme suivant :

Algorithme 5-1

Soit l une liste de couples d'états ($l \subseteq Q_1 \times Q_2$).

début
  $StState := \emptyset$; $StTrans := \emptyset$;
  $Equiv\_List\,(q_{01}, q_{02}) := \emptyset$;
  $l := succ\_produit_\Lambda\,(q_{01}, q_{02})$;
  si $l \neq \{fail\}$ **alors**
    empiler $((q_{01}, q_{02}), StState)$;
    empiler $(l, StTrans)$;
  fsi
  tantque $StState \neq \emptyset$ **faire** **(1)**
    (q_1, q_2) := sommet $(StState)$;
    si sommet $(StTrans) \neq \emptyset$ **alors**
      (* il reste des séquences successeurs à visiter *)
      choisir $(\lambda, (q_1', q_2'))$ **dans** sommet $(StTrans)$;
      sommet $(StTrans)$:= sommet $(StTrans) - \{(\lambda, (q_1', q_2'))\}$;
      si $(q_1', q_2') \notin StState$ **alors**
        $l := succ\_produit_\Lambda\,(q_1', q_2')$;
        si $l \neq \{fail\}$ **alors**
          $Equiv\_List\,(q_1', q_2') := \emptyset$;
          empiler $((q_1', q_2'), StState)$ **(2)** ;
          empiler $(l, StTrans)$
        fsi
      sinon
        (* (q_1', q_2') appartient à la séquence courante *)
        $Equiv\_List\,(q_1, q_2) := Equiv\_List\,(q_1, q_2) \cup \{(\lambda, q_1'), (\lambda, q_2')\}$
      fsi
    sinon
      (* toutes les séquences successeurs ont été visitées *)
      dépiler $(StState)$;
      dépiler $(StTrans)$;
      si $Equiv\_List\,(q_1, q_2) = T_{1_\Lambda}[q_1] \cup T_{2_\Lambda}[q_2]$ **alors**
        (* la séquence courante appartient à E^e *)
        $Equiv\_List\,(\text{sommet}(StState)) := Equiv\_List\,(\text{sommet}(StState)) \cup \{(\lambda, q_1), (\lambda, q_2)\}$
      fsi
    fsi

ftantque
si $Equiv\_List(q_{01}, q_{02}) = T_{1_\Lambda}[q_{01}] \cup T_{2_\Lambda}[q_{02}]$ **alors**
(* $((q_{01}, q_{02})) \in E^e$ *)
afficher $(S_1 \sim_\Lambda S_2)$
sinon
(* $((q_{01}, q_{02})) \notin E^e$ *)
afficher $(S_1 \not\sim_\Lambda S_2)$
fsi
fin.

■

La correction de cet algorithme repose sur les deux arguments suivants :

- La terminaison est assurée car seuls des états n'appartenant pas à *StState* sont empilés (en **(2)**) : par conséquent, à chaque nouvel état empilé correspond une séquence appartenant à l'ensemble *fini* des séquences d'exécution *élémentaires* de $S_1 \otimes_\Lambda S_2$.

- A la fin de chaque itération de la boucle **(1)**, en notant σ la séquence composée des états de *StState*, il est facile de voir, d'après la proposition 5.2-3, que l'algorithme maintient l'invariant suivant :

  $$\text{sommet}\, StTrans = \emptyset \;\Rightarrow\; (Equiv\_List(q_1, q_2) = T_{1_\Lambda}[q_1] \cup T_{2_\Lambda}[q_2] \;-\; (\sigma, (q_1, q_2)) \in E^e)$$

  Par suite, lorsque cette itération termine, on a bien :

  $$Equiv\_List(q_{01}, q_{02}) = T_{1_\Lambda}[q_{01}] \cup T_{2_\Lambda}[q_{02}] \;-\; ((q_{01}, q_{02})) \in E^e$$

Les coûts en temps et en mémoire de l'algorithme 5-1 sont alors les suivants :

- Les tailles des piles *StState* et *StTrans* sont majorées par la longueur d de la plus longue séquence élémentaire de $S_1 \otimes_\Lambda S_2$. De plus, pour chaque état (q_1, q_2) de la séquence courante, la taille de la liste $Equiv\_Listes\,(q_1, q_2)$ est majorée par $|T_{1_\Lambda}[q_1] \cup T_{2_\Lambda}[q_2]|$. Par conséquent, en supposant que chaque état de S_1 (*resp.* de S_2) a un nombre de successeurs borné par une constante c_1 (*resp.* c_2), le coût en mémoire de l'algorithme 5-1 est de $O((c_1 + c_2).d)$.

- On note c le facteur de branchement de $S_1 \otimes_\Lambda S_2$, avec $c \leq c_1.c_2$. Le coût en temps d'une exécution de l'itération **(1)** est de l'ordre de $O(c)$. Or, le nombre d'exécutions de cette itération correspond au nombre total n^* d'états visités lors du parcours en profondeur de $S_1 \otimes_\Lambda S_2$, ce qui donne un coût en temps pour l'algorithme 5-1 de $O(c.n^*)$. Du fait que le système $S_1 \otimes_\Lambda S_2$ puisse contenir des *cycles*, la valeur de n^* est en général largement supérieure au nombre total d'états n de $S_1 \otimes_\Lambda S_2$. On trouve une première approximation de cette valeur dans [Hol89] :

  $$n \leq n^* \leq c^d$$

  Par la suite, un encadrement plus précis, qui ne dépend que de la valeur de n, a été proposé dans [JJ91] :

  $$n \leq n^* \leq e.(n!) + 1$$

Si ce premier algorithme résout bien une partie des objectifs recherchés (puisqu'il peut être mis en œuvre "à la volée" avec un coût en mémoire réduit), son coût en temps reste exponentiel en fonction du nombre d'états du produit synchrone, ce qui est prohibitif en pratique.

Nous proposons donc une amélioration possible qui, toujours sur la base d'un parcours en profondeur de $S_1 \otimes_\Lambda S_2$, consiste à réduire la complexité en maintenant en mémoire l'ensemble des états déjà visités, et permet ainsi d'éviter de les parcourir à nouveau lors de l'examen d'une nouvelle séquence. Nous présenterons au chapitre 6 des moyens pour contrôler ce surcoût en mémoire.

Dans la suite, on considère deux cas de figure en fonction de la structure des systèmes de transitions étiquetées à comparer : on parlera de cas "déterministe" lorsqu'au moins l'un des deux systèmes est déterministe (par rapport à l'ensemble de langages Λ) et de cas "général", lorsqu'aucun de ces deux systèmes n'est déterministe.

5.3 Le cas "déterministe"

On montre que, lorsque l'un des deux systèmes de transitions étiquetées est déterministe, le critère établi dans la section 5.1 peut être simplifié de manière significative.

5.3.1 Un critère d'équivalence plus simple

Lorsque l'un au moins des systèmes de transitions étiquetées que l'on considère est déterministe, la définition des relations $\sim_\Lambda^k$ pour $k \geq 1$ peut être réécrite de la façon suivante :

Lemme 5.3-1
Soient $S_i = (Q_i, A_i, T_i, q_{0i})_{(i=1,2)}$ deux systèmes de transitions étiquetées, tels que S_1 ou S_2 soit *déterministe*. Pour toute équivalence de bisimulation $\sim_\Lambda$, on a :

$$\forall k \geq 1 \,.\, p \sim_\Lambda^k q \;-\; (\mathcal{A}\mathrm{ct}_\Lambda(p) = \mathcal{A}\mathrm{ct}_\Lambda(q) \;\wedge\; \forall p' \,.\, \forall q' \,.\, p \xrightarrow{\lambda}_{T_1} p' \;\wedge\; q \xrightarrow{\lambda}_{T_2} q' \;\Rightarrow\; p' \sim_\Lambda^{k-1} q')$$

■

Preuve : Pour $k \geq 1$, la relation $\sim_\Lambda^k$ est définie comme la conjonction des trois propriétés suivantes :

(i) $p \sim_\Lambda^1 q$.

(ii) $\forall \lambda \in \Lambda \,.\, \forall p' \,.\, (p \xrightarrow{\lambda}_{T_1} p' \;\Rightarrow\; \exists q' \,.\, (q \xrightarrow{\lambda}_{T_2} q' \;\wedge\; p' \sim_\Lambda^{k-1} q'))$.

(iii) $\forall \lambda \in \Lambda \,.\, \forall q' \,.\, (q \xrightarrow{\lambda}_{T_2} q' \;\Rightarrow\; \exists p' \,.\, (p \xrightarrow{\lambda}_{T_1} p' \;\wedge\; p' \sim_\Lambda^{k-1} q'))$.

Si l'on suppose S_1 déterministe, on a

$$\forall \lambda \in \Lambda \,.\, |T_{1_\lambda}| \leq 1$$

d'où :

$$\begin{aligned}
(iii) \quad - \quad & \forall \lambda \in \Lambda \,.\, \forall q' \,.\, (q \xrightarrow{\lambda}_{T_2} q' \;\Rightarrow\; (T_{1_\lambda}[p] \neq \emptyset \;\wedge\; \forall p' \,.\, p \xrightarrow{\lambda}_{T_1} p' \;\Rightarrow\; p' \sim_\Lambda^{k-1} q')) \\
- \quad & \forall \lambda \in \Lambda \,.\, \forall q' \,.\, (q \xrightarrow{\lambda}_{T_2} q' \;\Rightarrow\; T_{1_\lambda}[p] \neq \emptyset) \;\wedge \\
& \forall \lambda \in \Lambda \,.\, \forall q' \,.\, (q \xrightarrow{\lambda}_{T_2} q' \;\Rightarrow\; \forall p' \,.\, p \xrightarrow{\lambda}_{T_1} p' \;\Rightarrow\; p' \sim_\Lambda^{k-1} q')
\end{aligned}$$

Par suite,

$$(i) \;\wedge\; (ii) \;\wedge\; (iii) \;\Rightarrow\; (\mathcal{A}\mathrm{ct}_\Lambda(p) = \mathcal{A}\mathrm{ct}_\Lambda(q) \;\wedge\; \forall p' \,.\, \forall q' \,.\, p \xrightarrow{\lambda}_{T_1} p' \;\wedge\; q \xrightarrow{\lambda}_{T_2} q' \;\Rightarrow\; p' \sim_\Lambda^{k-1} q')$$

□.

A partir de ce lemme, la proposition 5.3-1 [FM90] fournit un critère simple pour décider si un couple d'états (p, q) du produit synchrone $S_1 \otimes_\Lambda S_2$ appartient ou non à la relation de bisimulation $\sim_\Lambda$: $p \sim_\Lambda q$ si et seulement si il n'existe aucune séquence d'exécution sur $S_1 \otimes_\Lambda S_2$ qui, à partir de l'état (p, q), conduise à l'état $fail$.

Proposition 5.3-1
Soient S_1 et S_2 deux systèmes de transitions étiquetées tels que S_1 (ou S_2) soit *déterministe*.

On considère l'équivalence de bisimulation $\sim_\Lambda$ et on note S le produit synchrone $S_1 \otimes_\Lambda S_2$ (avec $S = (Q, A, T, q_0)$).

On a alors :

$$\forall (p,q) \in Q \,.\, (p \not\sim_\Lambda q \;-\; \exists \sigma \in Ex_\Lambda((p,q)) \,.\, last\,(\sigma) = fail)$$

■

Preuve : Pour tout état (p,q) de S,

$$p \not\sim_\Lambda q \;-\; \exists k \geq 1 \,.\, p \not\sim^k_\Lambda q \;\wedge\; p \sim^{k-1}_\Lambda q.$$

On montre alors la proposition par induction sur k :

- Pour $k = 1$, on a, d'après la proposition 5.2-2 (i) :

  $$p \not\sim^1_\Lambda q \;-\; \exists \sigma \in Ex_\Lambda((p,q)) \,.\, \sigma = ((p,q), \phi, fail)$$

- On suppose la proposition vraie pour tout $k < l$, où l est un entier fixé. L'hypothèse d'induction s'écrit alors :

  $$\forall k,\, 1 \leq k < l \,.\, p \not\sim^k_\Lambda q \;\wedge\; p \sim^{k-1}_\Lambda q \;-\; \exists \sigma \in Ex_\Lambda((p,q)) \,.\, \exists n \,.\, \sigma(n) = fail$$

  Soit (p,q) tel que $p \not\sim^l_\Lambda q \;\wedge\; p \sim^{l-1}_\Lambda q$. D'après le lemme 5.3-1,

  $$l > 1 \;\wedge\; p \not\sim^l_\Lambda q \;-\; \exists \lambda \,.\, \exists p' \,.\, \exists q' \,.\, p \xrightarrow{\lambda}_{T_1} p' \;\wedge\; q \xrightarrow{\lambda}_{T_2} q' \;\wedge\; p' \not\sim^{l-1}_\Lambda q' \;\wedge\; p' \sim^{l-2}_\Lambda q'.$$

  Or, par hypothèse d'induction,

  $$\exists \sigma' \in Ex_\Lambda((p',q')) \,.\, last\,(\sigma') = fail.$$

  d'où $last\,(\sigma) = fail$.

□.

5.3.2 Algorithme

Compte-tenu de la proposition 5.3-1, lorsque l'un des deux systèmes de transitions est déterministe, la procédure de décision pour une équivalence de bisimulation $\sim_\Lambda$ peut se ramener à une simple analyse d'accessibilité sur $S_1 \otimes_\Lambda S_2$: il suffit de s'assurer que l'état $fail$ n'est pas un état accessible de ce système.

Plusieurs algorithmes d'analyse d'accessibilité (ou de *parcours exhaustif*) peuvent être mis en œuvre pour effectuer ce type de vérification. Une étude détaillée des algorithmes existants et de leurs complexités respectives est faite dans [Hol89] et [JÉ91].

La solution que nous présentons ici est basée sur un parcours en profondeur "classique" du système $S_1 \otimes_\Lambda S_2$, similaire à celui mis en œuvre dans l'algorithme 5-1. Toutefois, à la différence de cet algorithme, on mémorise, en plus de la séquence d'exécution courante, l'ensemble des états déjà explorés lors de l'examen des séquences d'exécution précédentes, et qui n'ont donc plus à être revisités (puisque tous leurs successeurs ont déjà été visités). Par conséquent, l'algorithme termine soit lorsque l'état $fail$ est atteint, soit lorsqu'il ne reste plus aucun nouvel état à explorer. Le choix du parcours en profondeur sera justifié au chapitre 6.

Il reste à décrire les structures de données nécessaires :

- De même que dans l'algorithme 5-1, le parcours du système de transitions sera géré à l'aide de deux piles : une pile $StState$, pour mémoriser les états de la séquence courante, et une pile $StTrans$, pour mémoriser les ensembles de transitions restant à exécuter à partir de chacun des états de cette séquence.

- Enfin, les états déjà visités seront mémorisés dans un ensemble noté $Visited$.

On obtient alors l'algorithme suivant :

Algorithme 5-2

Soit l une liste de couples d'états ($l \subseteq Q_1 \times Q_2$).
début
  $StState := \emptyset$; $StTrans := \emptyset$;
  $l := succ\_produit_\Lambda\,(q_{01}, q_{02})$;
  si $l \neq \{fail\}$ **alors**
    empiler $((q_{01}, q_{02}), StState)$;
    empiler $(succ\_produit_\Lambda\,(q_{01}, q_{02}), StTrans)$;
  sinon
    (* $fail$ est accessible depuis l'état initial : les systèmes ne sont pas équivalents *)
    afficher $(S_1 \not\sim_\Lambda S_2)$;
    retourner (**Faux**)
  fsi
  tantque $StState \neq \emptyset$ **faire** **(1)**
    $(q_1, q_2) :=$ sommet $(StState)$;
    si sommet $(StTrans) \neq \emptyset$ **alors**
      (* il reste des séquences successeurs à visiter *)
      choisir $(\lambda, (q_1', q_2'))$ **dans** sommet $(StTrans)$;
      sommet $(StTrans) :=$ sommet $(StTrans) - \{(\lambda, (q_1', q_2'))\}$;
      $l := succ\_produit_\Lambda\,(q_1', q_2')$;
      si $l \neq \{fail\}$ **alors**
        si $(q_1', q_2') \notin Visited$ **alors**
          $Visited := Visited \cup \{(q_1', q_2')\}$; **(2)**
          empiler $((q_1', q_2'), StState)$; **(3)**
          empiler $(l, StTrans)$
        fsi
      sinon
        (* $fail$ est accessible depuis l'état initial : les systèmes ne sont pas équivalents *)
        afficher $(S_1 \not\sim_\Lambda S_2)$;
        retourner (**Faux**)
      fsi
    sinon
      (* toutes les séquences successeurs ont été visitées *)
      dépiler $(StState)$;
      dépiler $(StTrans)$
    fsi
  ftantque
  afficher $(S_1 \sim_\Lambda S_2)$
  retourner (**Vrai**)
fin.

■

En notant n et m les nombre d'états et de transitions du produit $S_1 \otimes_\Lambda S_2$, les coûts en temps et en mémoire de l'algorithme 5-2 sont de $O(m)$. En effet :

- Du fait que d'une part seuls des états n'appartenant pas à $Visited$ soient empilés (en **(3)**) et que, d'autre part, tout nouvel état empilé soit inséré dans $Visited$ (en **(2)**), il est clair que

l'itération **(1)** sera exécutée au plus $|Q|$ fois, si Q dénote l'ensemble des états de $S_1 \otimes_\Lambda S_2$.

De plus, en supposant que les opérations "empiler" et "dépiler", ainsi que l'insertion et le test d'appartenance à l'ensemble $Visited$ puissent être réalisées en $O(1)$, la seule opération non constante de l'itération **(1)** est le calcul de $succ\_produit_\Lambda(q'_1, q'_2)$, qui peut être réalisée en $O(|T_\Lambda[(q'_1, q'_2)]|)$.

Par conséquent, le coût total en temps de l'algorithme 5-2 est de $\Sigma_{q \in Q}\ O(|T_\Lambda[q]|) = O(m)$.

- Les tailles des structure de données les plus coûteuses utilisées dans l'algorithme 5-2 sont de $O(n)$ pour l'ensemble $Visited$, et de $O(c.d)$ pour la pile $StTrans$, avec c et d dénotant respectivement la longueur de la plus longue séquence élémentaire et le facteur de branchement de $S_1 \otimes_\Lambda S_2$. Dans le cas le plus défavorable (i.e., lorsque $d = n$), le produit $c.d$ peut être égal au nombre total de transitions de $S_1 \otimes_\Lambda S_2$. Par suite, le coût en mémoire de cet algorithme est de $O(m)$.

Ces complexités seront étudiées plus en détail au chapitre 6 en fonction des implémentations possibles pour les différentes structures de données.

5.4 Le cas général

Nous présentons tout d'abord de manière intuitive la solution que nous avons retenue pour améliorer la complexité en temps de l'algorithme 5-1 dans le cas général. Nous décrivons alors le nouvel algorithme ainsi obtenu, puis nous le justifions formellement.

5.4.1 Un critère d'équivalence "conditionnel"

Le coût exponentiel de l'algorithme 5-1 est dû au fait que, lorsque le produit $S_1 \otimes_\Lambda S_2$ contient des cycles, un certain nombre de ses états peuvent appartenir à plusieurs séquences d'exécution élémentaires, ce qui explique que le nombre total d'états visités lors de l'énumération de chacune de ces séquences puisse largement dépasser le nombre d'états de ce système. Par conséquent, pour réduire la complexité de cet algorithme, une solution attrayante consisterait à utiliser un parcours similaire à celui mis en œuvre dans le cas déterministe, et qui permettrait de ne visiter qu'une fois et une seule chacun des états de $S_1 \otimes_\Lambda S_2$.

Toutefois, d'une manière générale, pour que cette technique soit applicable, il est nécessaire de savoir associer à chaque état visité lors du parcours d'une séquence d'exécution donnée, une information qui puisse être mémorisée et exploitée par la suite si cet état est de nouveau atteint lors de l'examen d'une séquence suivante. Or, dans le cas de l'algorithme 5-1, la seule information dont on dispose sur les états visités est celle fournie par son invariant (*cf.* section 5.2.2) qui permet, lorsqu'un état (q_1, q_2) est dépilé, de décider si la séquence d'exécution σ contenue dans la pile, appartient ou non à l'ensemble E^e :

$$(Equiv\_List(q_1, q_2) = T_{1_\Lambda}[q_1] \cup T_{2_\Lambda}[q_2]) \ - \ (\sigma, \lambda, (q_1, q_2)) \in E^e$$

Par conséquent, cette information, qui dépend de la séquence courante σ, ne sera pas directement exploitable si (q_1, q_2) est re-visité lors du parcours d'une nouvelle séquence σ'.

La solution que nous proposons repose alors sur les deux arguments suivants, qui sont détaillés dans la suite :

- On montre en premier lieu que, en supposant *à priori* équivalents certains états de $S_1 \otimes_\Lambda S_2$, il est possible de renforcer l'invariant de l'algorithme 5-2 et de le rendre indépendant de la séquence courante en décidant directement, pour chaque couple (q_1, q_2) dépilé, si les états q_1

et q_2 se bisimulent ou non. En notant $\mathcal{H}$ les hypothèses effectuées, l'invariant de l'algorithme devient alors :

$$\mathcal{H} \Rightarrow ((Equiv\_List(q_1, q_2) = T_{1_\Lambda}[q_1] \cup T_{2_\Lambda}[q_2]) \ - \ q_1 \sim_\Lambda q_2)$$

Par suite, lorsqu'un état (q'_1, q'_2), déjà visité lors de l'examen d'une séquence d'exécution précédente, est de nouveau atteint à partir d'un état (q_1, q_2), il n'est plus nécessaire de reparcourir l'ensemble des successeurs de (q'_1, q'_2) pour décider si q_1 et q_2 se bisimulent : l'information $q'_1 \sim_\Lambda q'_2$ suffit.

Sous la condition $\mathcal{H}$, il devient donc possible de vérifier que les états initiaux de S_1 et S_2 se bisimulent en ne visitant qu'une fois et une seule chacun des états de $S_1 \otimes_\Lambda S_2$, soit en $O(|T_\Lambda|)$.

- On montre alors, que, en effectuant au plus $|Q|$ itérations préalables de cet algorithme, il est toujours possible de se ramener à une situation où la condition $\mathcal{H}$ est vérifiée. Par conséquent, le coût total en temps de l'algorithme finalement obtenu est de $O(|Q|.|T_\Lambda|)$

Plus formellement, nous procédons de la manière suivante :

- On définit tout d'abord une relation $R_\Lambda^{\Sigma^e}$ sur $Q_1 \times Q_2$ similaire à celle introduite pour la fonction Σ dans la définition 5.1-2 : l'état (p, q) de $S_1 \otimes_\Lambda S_2$ appartient à $R_\Lambda^{\Sigma^e}$ si et seulement si il termine une séquence d'exécution de E^e.

- On montre alors sur un exemple que la relation $R_\Lambda^{\Sigma^e}$ n'est pas une équivalence de bisimulation dans le cas général, puis on donne une condition suffisante $\mathcal{H}$ sur l'ensemble des séquences d'exécutions de $S_1 \otimes_\Lambda S_2$ pour laquelle cette relation est toujours une bisimulation.

- Enfin, on termine en montrant que la condition $\mathcal{H}$ peut être vérifiée "après coup" de façon indépendante, et, lorsqu'elle n'est pas valide, transformée en une condition $\mathcal{H}'$ telle que :

  - $\mathcal{H}'$ soit plus strictement plus faible que $\mathcal{H}$ (vis-à-vis de l'implication),
  - $\mathcal{H}'$ soit une condition suffisante pour assurer que $R_\Lambda^{\Sigma^e}$ est une bisimulation.

  Par suite, après un nombre fini d'itérations de cette procédure, il est toujours possible d'obtenir une condition suffisante qui soit valide.

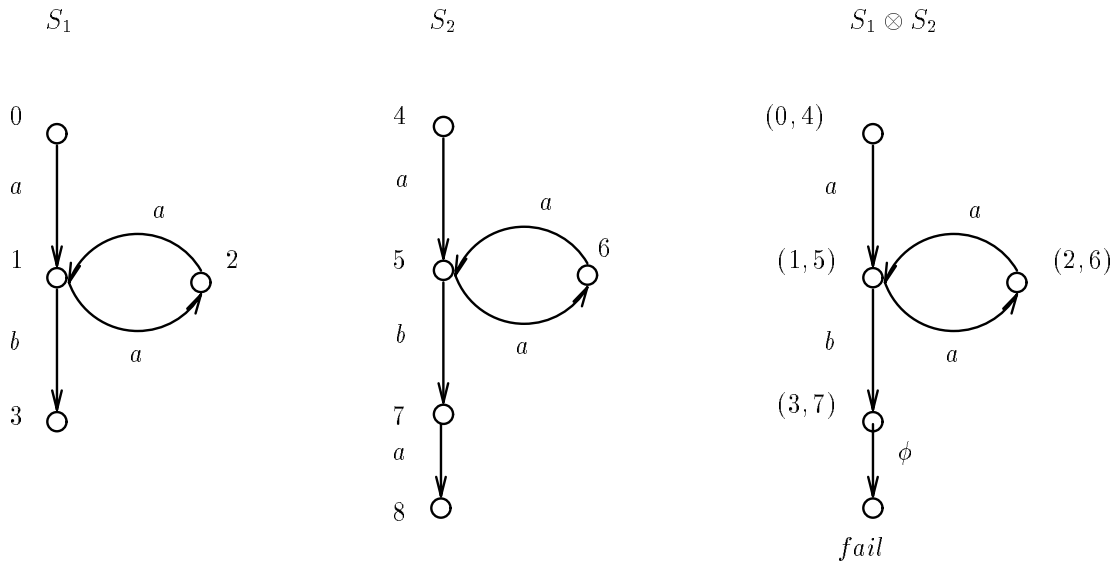
Définition 5.4-1
Soient S_1 et S_2 deux systèmes de transitions étiquetées, soit S le produit synchrone $S_1 \otimes_\Lambda S_2$, avec $S = (Q, A, T, (p_0, q_0))$, et soit E^e le plus grand sous-ensemble de $Ex_\Lambda(S)$ satisfaisant $E^e \subseteq \Sigma_\Lambda^e(E^e)$.
La relation $R_\Lambda^{\Sigma^e} \subseteq Q_1 \times Q_2$ est défini par :

$$R_\Lambda^{\Sigma^e} = \{(p, q) \in Q \mid \exists \sigma \ . \ last(\sigma) = (p, q) \ \wedge \ \sigma \in E^e\}$$

■

L'exemple suivant permet de montrer que la relation $R_\Lambda^{\Sigma^e}$ n'est pas une équivalence de bisimulation dans le cas général :

Exemple 5-1



On note σ la séquence d'exécution $((0,4),a,(1,5),a,(2,6))$ de $S_1 \otimes_\Lambda S_2$. En considérant comme observable chacune des actions de S_1 et S_2 ($\{\{a\} \mid a \in \mathcal{A}\}$), on a alors :

$(2,6) \in R_\Lambda^{\Sigma^e}$ et $2 \not\sim_\Lambda 6$

En effet :

- D'une part la séquence σ appartient à E^e, puisque $T[(2,6)] = \{(a,(1,5))\}$, et $(1,5) \in \sigma$.
- D'autre part, les états 2 et 6 ne se bisimulent pas, comme l'indique la séquence $((2,6),a,(1,5),b,(3,7))$ avec $3 \not\sim^1 7$ qui est une *séquence diagnostique*.

■

Compte-tenu de la proposition 5.1-1, obtenir une condition suffisante pour que $R_\Lambda^{\Sigma^e}$ soit une équivalence de bisimulation revient à obtenir une condition suffisante pour que les relations $R_\Lambda^{\Sigma^e}$ et R_Λ^{Σ} coïncident, ou encore pour que les ensemble E^e et E construit à partir des fonctions Σ^e et Σ coïncident.

La proposition suivante établit une condition nécessaire pour laquelle cette propriété est vérifiée :

Proposition 5.4-1
Soient S_1 et S_2 deux systèmes de transitions étiquetées, soit S le produit synchrone $S_1 \otimes_\Lambda S_2$, avec $S = (Q, A, T, (p_0, q_0))$, et soit X un ensemble de séquences d'exécutions de S ($X \subseteq Ex_\Lambda(S)$).

On note respectivement $H(\sigma, X)$ et $H(S, X)$ les prédicats binaires suivants :

$$H(\sigma,X) \equiv \forall\lambda\,.\,\forall(p',q')\,.\,((\sigma,\lambda,(p',q')) \in X \wedge (p',q') \in \sigma \Rightarrow (\sigma,\lambda,(p',q')) \in E)$$
$$H(S,X) \equiv \forall\sigma\,.\,(\sigma \in Ex_\Lambda(S) \Rightarrow H(\sigma,X))$$

on a alors :

$$(E \subseteq X \wedge H(S,X)) \Rightarrow (E^e = E)$$

■

Preuve : On note H l'ensemble des séquences σ qui vérifient $H(\sigma, X)$. Il suffit alors de montrer :

$$\forall\sigma\,.\,\sigma \in H \Rightarrow (\sigma \in E^e - \sigma \in E)$$

sens $\Rightarrow$: Il est facile de voir, par définition des fonctions Σ_Λ et Σ_Λ^e, que, lorsque $E \subseteq X$ et

$H(\sigma, X)$ est vérifiée,

$$\sigma \in \Sigma^e_\Lambda(E^e) \Rightarrow \sigma \in \Sigma_\Lambda(E^e).$$

On en déduit :

$$\begin{aligned} E^e \subseteq \Sigma^e_\Lambda(E^e) &\Rightarrow (H \cap E^e) \subseteq \Sigma_\Lambda(H \cap E^e), \\ &\Rightarrow (H \cap E^e) \subseteq E. \end{aligned}$$

Par suite,

$$H(\sigma, X) \wedge \sigma \in E^e \Rightarrow \sigma \in E.$$

sens $\Leftarrow$: C'est une conséquence directe du lemme 5.1-2.

□.

Remarque 5-2
Même lorsque la condition $H(S, X)$ est fausse, on a toujours $E \subseteq E^e$ (lemme 5.1-2).

■

On vérifie alors, que, lorsque $E \subseteq X$ et sous l'hypothèse $H(S_1 \otimes_\Lambda S_2, X)$, il est possible de déterminer si une séquence d'exécution donnée σ de $S_1 \otimes_\Lambda S_2$ appartient ou non à E^e en visitant *au plus* une fois chacun des états de ce système. Plus précisément, la proposition suivante montre que, lorsque le dernier état de σ a déjà été dépilé au cours de l'examen d'une séquence précédente σ', il est possible de décider en fonction du résultat obtenu si σ appartient ou non à E^e :

Proposition 5.4-2
Soit S le produit synchrone $S_1 \otimes_\Lambda S_2$ de deux systèmes de transitions étiquetées. En supposant que le prédicat $H(S, X)$ est vérifié pour un X donné avec $E \subseteq X$, on a alors :

(i) $\forall \sigma \in Ex_\Lambda(S), (\exists \sigma' \,.\, (last(\sigma) = last(\sigma') \wedge \sigma' \in E^e)) \Rightarrow \sigma \in E^e$

(ii) $\forall \sigma \in Ex_\Lambda(S), (\exists \sigma' \,.\, (last(\sigma) = last(\sigma') \wedge \sigma' \notin E^e)) \Rightarrow \sigma \notin E^e$

■

Preuve : En notant E le plus grand sous-ensemble de $Ex_\Lambda(S)$ satisfaisant $E \subseteq \Sigma_\Lambda(E)$, on a :

$$H(S, X) \Rightarrow (E^e = E) \quad \text{(proposition 5.4-1)}.$$

On pose $last(\sigma) = (p, q)$. On a alors respectivement :

$$\begin{aligned} last(\sigma') = (p, q) \wedge \sigma' \in E^e &\Rightarrow \sigma' \in E \\ &\Rightarrow (p, q) \in R^\Sigma_\Lambda \\ &\Rightarrow \sigma \in E \quad \text{(corollaire 5.1-1)} \\ &\Rightarrow \sigma \in E^e, \end{aligned}$$

et,

$$\begin{aligned} last(\sigma') = (p, q) \wedge \sigma' \notin E^e &\Rightarrow \sigma' \notin E \\ &\Rightarrow (p, q) \notin R^\Sigma_\Lambda \quad \text{(corollaire 5.1-1)} \\ &\Rightarrow \sigma \notin E \quad \text{(corollaire 5.1-1)} \\ &\Rightarrow \sigma \notin E^e. \end{aligned}$$

□.

On termine alors en montrant qu'il est toujours possible de s'assurer que l'hypothèse $H(S_1 \otimes_\Lambda S_2, X)$ effectuée dans la proposition 5.4-1 sur les séquences d'exécutions de $S_1 \otimes_\Lambda S_2$ est valide, ou, le cas échéant, de la transformer en une hypothèse H' qui soit *strictement* plus faible :

Intuitivement, le prédicat $H(\sigma, X)$ est vérifié si et seulement si chacune des séquences d'exécution σ' obtenues en ajoutant à σ un couple (p', q'), successeur de $last\,(\sigma)$ et inclus dans σ, appartient à l'ensemble E^e. Par suite, les séquences σ' ne sont pas élémentaires, et le fait qu'elles appartiennent ou non à E^e ne pourra donc pas être directement vérifié lors du parcours de $S_1 \otimes_\Lambda S_2$.

Toutefois, à chacune des ces séquences σ', il est toujours possible d'associer une séquence d'exécution *élémentaire* σ" de $S_1 \otimes_\Lambda S_2$ qui d'une part soit un *préfixe propre* de σ', et d'autre part soit telle que si σ" appartient à E^e alors σ' appartient également à cet ensemble.

Plus précisément, si

$$\sigma = ((p_1, q_1), \lambda_1, (p_2, q_2), \ldots, \lambda_{k-1}, (p', q'), \ldots, \lambda_{n-1}, (p, q))$$

et

$$\sigma' = ((p_1, q_1), \lambda_1, (p_2, q_2), \ldots, \lambda_{k-1}, (p', q'), \ldots, \lambda_{n-1}, (p, q), \lambda_n, (p', q')),$$

alors, en posant

$$\sigma" = ((p_1, q_1), \lambda_1, (p_2, q_2), \ldots, \lambda_{k-1}, (p', q')),$$

on a :

$$\sigma" \in E^e \;\Rightarrow\; \sigma' \in E^e \quad \textbf{(1)}.$$

Par conséquent :

- Si σ appartient à l'ensemble des séquences d'exécution énumérées lors du parcours en profondeur de $S_1 \otimes_\Lambda S_2$, alors σ" appartient également à cet ensemble (puisque σ" est un préfixe de σ).
- Vérifier que $H(\sigma, X)$ est valide revient à vérifier que chaque séquence σ" appartient à l'ensemble E^e (d'après **(1)**).

On en déduit alors que chaque hypothèse $H(\sigma, X)$ effectuée lors du parcours de $S_1 \otimes_\Lambda S_2$ pourra être vérifiée par la suite lorsque l'état p', q' sera dépilé à son tour.

La proposition suivante permet de justifier l'implication **(1)** en la généralisant à tout couple de séquences (σ_1, σ_2) de $Ex_\Lambda(S_1 \otimes_\Lambda S_2)$, où σ_1 est un préfixe de σ_2 et σ_1 et σ_2 ont même état terminal (la relation préfixe est notée $<_p$, *cf.* définition 4.4-1, chapitre 4).

Proposition 5.4-3
Soient S_1 et S_2 deux systèmes de transitions étiquetées, et soit S le produit synchrone $S_1 \otimes_\Lambda S_2$. On a alors :

$$\forall \sigma_1, \sigma_2 \in Ex_\Lambda(S)\,.\,(\sigma_1 <_p \sigma_2 \;\wedge\; last\,(\sigma_1) = last\,(\sigma_2)) \;\Rightarrow\; (\sigma_1 \in E^e \;\Rightarrow\; \sigma_2 \in E^e)$$

■

Preuve : On pose $last\,(\sigma_1) = last\,(\sigma_2) = (p, q)$. On montre alors :

$$\forall X,\, X \subseteq Ex_\Lambda(S)\,.\,\sigma_1 \in \Sigma^e_\Lambda\,(X) \;\Rightarrow\; \sigma_2 \in \Sigma^e_\Lambda\,(X)$$

Par définition de Σ^e_Λ,

$$\begin{aligned}
&\sigma_1 \in \Sigma^e_\Lambda\,(X) \Rightarrow \\
&\quad \forall \lambda\,.\,\forall p'\,.\,p \xrightarrow{\lambda}_{T_1} p' \;\Rightarrow\; \exists q'\,.\,q \xrightarrow{\lambda}_{T_2} q' \;\wedge\; ((p', q') \in \sigma_1 \;\vee\; (\sigma_1, \lambda, (p', q'))) \in X \\
&\quad \wedge \\
&\quad \forall \lambda\,.\,\forall q'\,.\,q \xrightarrow{\lambda}_{T_2} q' \;\Rightarrow\; \exists p'\,.\,p \xrightarrow{\lambda}_{T_1} p' \;\wedge\; ((p', q') \in \sigma_1 \;\vee\; (\sigma_1, \lambda, (p', q'))) \in X
\end{aligned}$$

Or,

$$\sigma_1 <_p \sigma_2 \;\Rightarrow\; ((p', q') \in \sigma_1 \;\Rightarrow\; (p', q') \in \sigma_2)$$

d'où, $\sigma_2 \in \Sigma^e_\Lambda(X)$. □.

En utilisant les notations introduites dans cette section, l'algorithme de vérification est alors basé sur le principe suivant :

1. On effectue un parcours du produit synchrone S, sous l'hypothèse $\mathcal{H}_0 = H(S, Ex_\Lambda(S))$, en construisant un ensemble E^e_0 obtenu en visitant *au plus* une fois chacun des états de ce système, et tel que $E \subseteq E^e_0$ (proposition 5.4-2). Différents cas peuvent alors se présenter :

   - Si la séquence d'exécution composée du seul état initial $((q_{01}, q_{02}))$ n'appartient pas à l'ensemble E^e_0 calculé, alors les systèmes S_1 et S_2 ne sont pas équivalents (*cf.* remarque 5-2) et l'algorithme est terminé.
   - Si $((q_{01}, q_{02}))$ appartient à E^e_0 et si l'hypothèse $\mathcal{H}_0$ s'avère valide, alors les systèmes S_1 et S_2 sont équivalents (*cf.* proposition 5.4-1) et l'algorithme est également terminé.
   - Enfin, si $((q_{01}, q_{02}))$ appartient à E^e_0, mais s'il existe une séquence σ, visité lors du parcours de $S_1 \otimes_\Lambda S_2$, telle que $H(\sigma, Ex_\Lambda(S))$ ne soit pas valide, alors il n'est pas possible à ce stade de conclure sur l'existence d'une bisimulation entre S_1 et S_2, puisque l'ensemble E^e calculé est différent de E, et que par conséquent des résultats faux ont pu être utilisés lors de ce calcul. On exécute alors le pas 2.

2. Puisque $H(\sigma, Ex_\Lambda(S))$ n'est pas valide, il existe un préfixe propre σ" de σ tel que σ" n'appartient pas à E^e_0, c'est à dire tel que $last$ (σ") n'appartient pas à R^Σ_Λ (puisque $E \subseteq E^e_0$). On ré-itère alors le pas 1 en utilisant cette information sur $last(\sigma")$, ce qui permet de ne plus avoir à considérer les séquences σ" et σ (puisque l'on sait qu'elles n'appartiennent pas à E). Plus généralement, seules les séquences de E^e_0 nécessiteront d'être visitées lors de ce nouveau parcours, ce qui permet de considérer une hypothèse $\mathcal{H}_1$ plus faible que $\mathcal{H}_0$ (au sens de l'implication) :

   $$\mathcal{H}_1 = H(S, E^e_0)$$

   d'où

   $$\mathcal{H}_0 \Rightarrow \mathcal{H}'_1.$$

Enfin, du fait que la suite $(\mathcal{H}^i)$ des hypothèses que l'on effectue soit strictement croissante (vis-à-vis de l'inclusion), l'algorithme termine en un nombre fini d'itérations.

5.4.2 Algorithme

On propose tout d'abord des structures de données qui permettent d'implémenter l'algorithme informel décrit dans la section 5.4.1. On donne alors l'algorithme complet en justifiant ses complexités en temps et en mémoire.

Le parcours "sous hypothèses" de $S_1 \otimes_\Lambda S_2$ sera implémenté à l'aide d'une fonction *parcours_produit*, d'un principe similaire à l'algorithme 5-2 utilisé dans le cas "déterministe". Cette fonction pourra alors retourner les valeurs **Vrai** (si les systèmes se bisimulent), **Faux** (s'ils ne se bisimulent pas) ou **Inconnu** (lorsqu'une des hypothèses s'est avérée fausse). Par suite, l'algorithme complet consistera en une suite d'appels à cette fonction jusqu'à obtenir un résultat dans l'ensemble {**Vrai**, **Faux**} en modifiant à chaque appel les hypothèses qui seront prises en compte.

On utilisera les structures de données suivantes :

- Le parcours en profondeur sera géré classiquement à l'aide des piles notées *StState*, et *StTrans* qui permettent de mémoriser respectivement la séquence courante et les ensembles de transitions restant à exécuter à partir de chacun des états de cette séquence, ainsi qu'un ensemble *Equiv_List* associé à *StState* (*cf.* proposition 5.2-3).

De plus les états déjà visités (et dépilés) à un instant donné seront mémorisés à l'aide d'un ensemble noté $Visited$.

- D'autre part, nous avons vu qu'il est également nécessaire, pour chaque état (p,q) dépilé, de mémoriser le résultat de son évaluation (i.e., si la séquence d'exécution qu'il termine appartient ou non à E^e, ou encore si (p,q) appartient à la relation $R_\Lambda^{\Sigma^e}$).

  On utilise pour ce faire un ensemble noté Non_Equiv_States, construit de la manière suivante :

  $$(p,q) \notin R_\Lambda^{\Sigma^e} \;\Rightarrow\; (p,q) \in Non\_Equiv\_States$$

  Notons que, l'ensemble Non_Equiv_States ne contient bien que des états non équivalents modulo $\sim_\Lambda$, quelque soit la validité de l'hypothèse $\mathcal{H}$, et qu'il pourra par conséquent être réutilisé lors du prochain appel à *parcours_produit*.

- Enfin, on utilisera également un troisième ensemble d'états, noté $Roots$, qui permet de vérifier si l'hypothèse $\mathcal{H}$ utilisé *à priori* lors du parcours s'avère ou non valide :

  Plus précisément, si σ est la séquence courante, chaque état (p',q'), successeur de $last(\sigma)$ et inclus dans σ (donc appartenant à la pile $StState$) sera inséré dans l'ensemble $Roots$. Une fois l'état (p',q') dépilé, on aura, en notant X l'ensemble des séquences d'exécutions qui ont été énumérées :

  $$H(\sigma, X) \;-\; (p',q') \notin Non\_Equiv\_States,$$

  et, par conséquent, lorsque la fonction *parcours_produit* se termine :

  $$H(S_1 \otimes_\Lambda S_2, Ex_\Lambda(S_1 \otimes_\Lambda S_2)) \;-\; Non\_Equiv\_States \cap Roots = \emptyset$$

On obtient alors l'algorithme de vérification suivant, qui permet de décider si les systèmes de transitions étiquetées S_1 et S_2 sont équivalents modulo une bisimulation $\sim_\Lambda$:

Algorithme 5-3

début
  $Non\_Equiv\_States := \emptyset$;
  répéter
    $parcours\_result := parcours\_produit()$;
  jusqua $parcours\_result \in \{$**Vrai**, **Faux**$\}$;
  si $parcours\_result =$ **Vrai alors**
    afficher $(S_1 \sim_\Lambda S_2)$
  sinon
    afficher $(S_1 \not\sim_\Lambda S_2)$
  fsi
fin.

■

Algorithme 5-4

fonction *parcours_produit* ()
Soit l une liste de couple d'états ($l \subseteq Q_1 \times Q_2$).
début
  $Visited := \emptyset$; $Roots := \emptyset$; $result :=$ **Inconnu** ;
  $StState := \emptyset$; $StTrans := \emptyset$;
  $Equiv\_List(q_{01}, q_{02}) := \emptyset$;
  $l := succ\_produit_\Lambda(q_{01}, q_{02})$;
  si $l \neq \{fail\}$ **alors**
    empiler $((q_{01}, q_{02}), StState)$;

```
        empiler (l, StTrans) ;
    fsi
    tantque StState ≠ ∅ faire   (1)
        result := Vrai ;
        (q1, q2) := sommet (StState) ;
        si sommet (StTrans) ≠ ∅ alors
            (* il reste des séquences successeurs à visiter *)
            choisir (λ, (q'1, q'2)) dans sommet (StTrans) ;
            sommet (StTrans) := sommet (StTrans) −{(λ, (q'1, q'2))} ;
            si (q'1, q'2) ∉ (Visited ∪ Non_Equiv_States) alors   (2)
                si (q'1, q'2) ∉ StState alors
                    l := succ_produit_Λ (q'1, q'2) ;
                    si l ≠ {fail} alors
                        Equiv_List (q'1, q'2) := ∅ ;
                        empiler ((q'1, q'2), StState) ;
                        empiler (l, StTrans)
                    fsi
                sinon
                    (* (q'1, q'2) appartient à la séquence courante *)
                    Roots := Roots ∪ {(q'1, q'2)} ;   (3)
                    Equiv_List (q1, q2) := Equiv_List (q1, q2) ∪ {(λ, q'1), (λ, q'2)}
                fsi
            sinon
                (* (q'1, q'2) a déjà été visité *)
                si (q'1, q'2) ∉ Non_Equiv_States alors
                    Equiv_List (q1, q2) := Equiv_List (q1, q2) ∪ {(λ, q'1), (λ, q'2)}
                fsi
            fsi
        sinon
            (* toutes les séquences successeurs ont été visitées *)
            dépiler (StState) ;
            dépiler (StTrans) ;
            Visited := Visited ∪ {(q1, q2)} ;   (4)
            si Equiv_List (q1, q2) = T1Λ[q1] ∪ T2Λ[q2] alors
                (* la séquence courante appartient à E^e *)
                Equiv_List (sommet(StState)) := Equiv_List (sommet(StState)) ∪ {(λ, q1), (λ, q2)}
            sinon
                Non_Equiv_States := Non_Equiv_States ∪ {(q1, q2)} ;
                si (q1, q2) ∈ Roots alors
                    result := Inconnu ;   (5)
                fsi
            fsi
        fsi
    ftantque
    si Equiv_List(q01, q02) = T1Λ[q01] ∪ T2Λ[q02] alors
        retourner (result)
    sinon
        retourner (Faux)
    fsi
```

fin.

■

La preuve de la correction partielle de l'algorithme 5-3 a été donnée dans la section 5.4.1. La proposition suivante justifie alors la terminaison de cet algorithme, et, en supposant que les opérations sur les structures *StState*, *StTrans*, *Visited*, *Non_Equiv_States* et *Roots* peuvent être réalisées en temps constant, d'obtenir une borne supérieure de sa complexité en temps.

Proposition 5.4-4

L'algorithme 5-3 termine en $O(|T|.|Q|)$ itérations.

■

Preuve : On note $Visited_i$, $Non\_Equiv\_States_i$ et $Roots_i$ les ensembles *Visited*, *Non_Equiv_States* et *Roots* obtenus à la fin du i^{eme} appel de la fonction *parcours_produit*.

On a alors :

(i) Chaque appel à la fonction *parcours_produit* est réalisé en $O(|T|)$:

Cette fonction effectue en effet un parcours similaire à celui mis en œuvre dans l'algorithme 5-2 : seuls les états n'appartenant pas à $Visited \cup Non\_Equiv\_States$ sont empilés (en **(2)**), et tout état empilé est inséré dans *Visited* (en **(4)**). Le nombre d'itérations de la boucle **(1)** est donc majoré par $|Q|$, ce qui implique que chaque transition de $S_1 \otimes_\Lambda S_2$ sera exécutée au plus une fois.

(ii) L'algorithme 5-3 effectue au plus $|Q|$ appels à *parcours_produit* :

A la fin du i^{eme} appel à la fonction *parcours_produit*, on a (en **(5)**) :

$$parcours\_result = \textbf{Inconnu} \; - \; Roots_i \cap Non\_Equiv\_States_i \neq \emptyset$$

Par suite, un $(i+1)^{eme}$ appel sera effectué si et seulement si $Roots_i \cap Non\_Equiv\_States_i \neq \emptyset$.

De plus, puisque d'une part les états de *Non_Equiv_States* ne sont jamais empilés (en **(2)**), et que d'autre part les éléments de *Roots* appartiennent nécessairement à *StState* (en **(3)**), on a :

$$R_i \cap Non\_Equiv\_States_{i-1} = \emptyset.$$

Comme d'autre part $Non\_Equiv\_States_{i-1} \subseteq Non\_Equiv\_States_i$, puisque les seules opérations effectuées sur *Non_Equiv_States* sont des insertions, on en déduit finalement :

$$Non\_Equiv\_States_{i-1} \subset Non\_Equiv\_States_i$$

L'ensemble *Non_Equiv_States* croît donc strictement à chaque appel de *parcours_produit*, et le nombre total d'appels à cette fonction sera par conséquent toujours majoré par n.

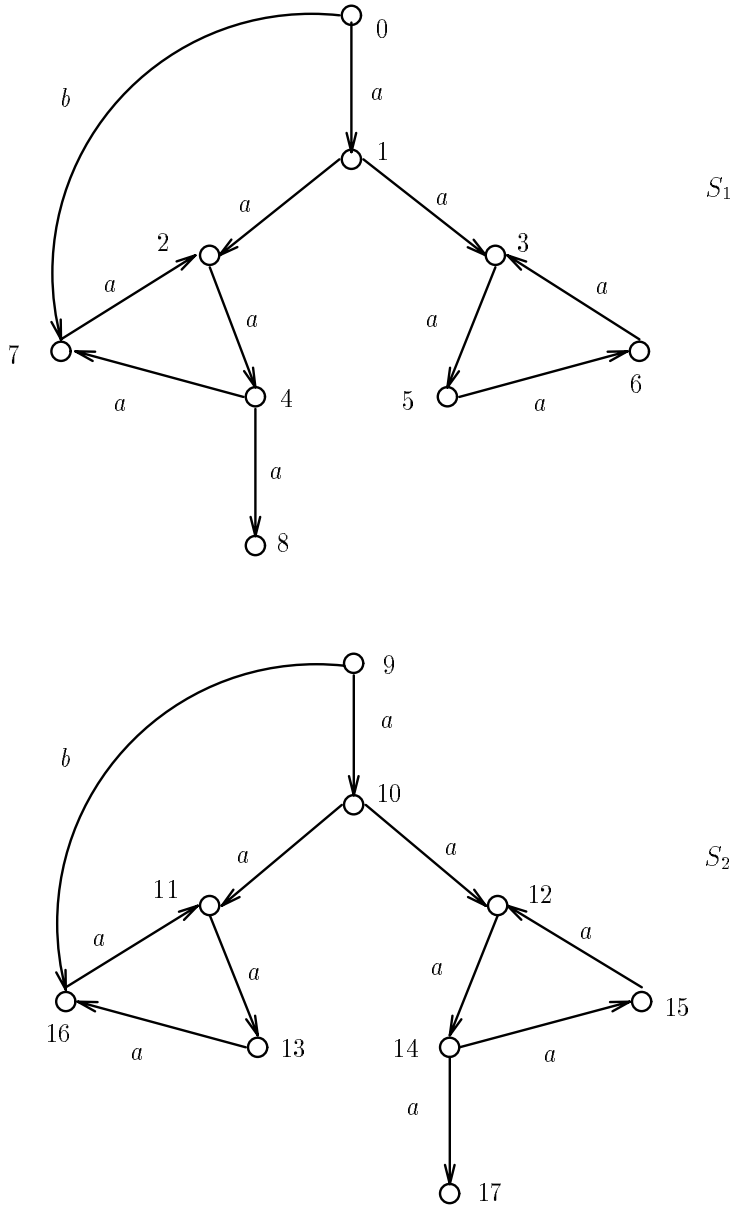
La proposition est alors la conjonction de (i) et (ii). □.

Par conséquent, en notant n et m le nombre d'états et de transitions du système de transitions étiquetées $S_1 \otimes_\Lambda S_2$, les coûts en temps et en mémoire de l'algorithme 5-3 sont respectivement de $O(m.n)$ et $O(m)$:

- Le coût en temps est une conséquence de la proposition 5.4-4,
- Le coût en mémoire est directement obtenu en faisant l'inventaire des structures de données utilisées.

Exemple 5-2

Lorsqu'on applique l'algorithme 5-3 aux systèmes de transitions étiquetées (non déterministes) S_1 et S_2 de la figure suivante, deux exécutions de la procédure *parcours_produit* () peuvent être nécessaires.



Ces deux systèmes sont non équivalents modulo la bisimulation forte. On a en effet la séquence diagnostique suivante :

$$\sigma = ((0,9) \xrightarrow{b} (7,16) \xrightarrow{a} (2,11) \xrightarrow{a} (4,13) \xrightarrow{a} (8,16)) \text{ et } 8 \not\sim^1 16.$$

Néanmoins, si l'on ne considère que les relations de transitions étiquetées par l'action a, alors ces deux systèmes se bisimulent.

En conséquence, si l'on suppose que lors du parcours en profondeur du produit synchrone $S_1 \otimes S_2$ les états atteignables par une action a sont visités en priorité, alors la première exécution de *parcours_produit* () ne retourne pas la valeur **Faux** puisque les états 7 et 16 auront été trouvés "équivalents" (i.e., $(7,16)$ n'appartient pas à $Non\_Equiv\_States_1$).

Par contre, ce premier appel retournera bien la valeur **Inconnu**, puisque l'état $(2,11)$ (ou encore

$(3, 12)$) appartient à $Roots_1 \cap Non\_Equiv\_States_1$. Un deuxième appel à *parcours_produit* () sera donc nécessaire, dans lequel on utilisera l'hypothèse que 2 et 11 sont non-équivalents, et qui retournera alors la valeur **Faux**. ■

5.5 Préordres et équivalences de simulation

La plupart des définitions et propositions introduites dans ce chapitre et qui concernent les équivalences de bisimulation reposent essentiellement sur la *structure* de ces relations, et non sur le fait qu'il s'agit de relations *d'équivalence*. Par conséquent, elles peuvent en général être directement adaptés aux préordres de simulation, et conduire à des algorithmes similaires à ceux obtenus dans le cas de la bisimulation.

On donne dans cette section les principaux résultats qui restent valides dans le cas des préordres de simulation, et qui permettent de déduire des algorithmes de vérification "à la volée" pour ce type de relations. En particulier, on utilisera les mêmes notations que dans les sections précédentes, et on ne redonnera pas les preuves des nouvelles propositions obtenues, qui peuvent être déduites de celles des propositions originales.

5.5.1 Critères de simulation sur l'ensemble des séquences d'exécution

On montre tout d'abord que l'existence d'un relation de simulation $\sqsubseteq_\Lambda$ entre deux systèmes de transitions étiquetées peut être exprimée à l'aide d'un critère sur l'ensemble des séquences d'exécutions de leur produit synchrone, semblable à celui défini dans le cas de la bisimulation.

Définition 5.5-1
Soient $S_i = (Q_i, A_i, T_i, q_{0i})_{(i=1,2)}$ deux systèmes de transitions étiquetées, soit Λ un ensemble de langages sur $\mathcal{A}$ et soit S leur produit synchrone $S = S_1 \times_\Lambda S_2$.

La fonction Σ_Λ est définie par :

$$\Sigma_\Lambda \ : \ 2^{Ex_\Lambda(S)} \longrightarrow 2^{Ex_\Lambda(S)}$$

et

$$\Sigma_\Lambda(E) \ = \ \{\sigma \in Ex_\Lambda(S) \mid last(\sigma) = (p,q) \ \Rightarrow \ \forall \lambda \,.\, \forall p' \,.\, p \xrightarrow{\lambda}_{T_1} p' \ \Rightarrow \ \exists q' \,.\, q \xrightarrow{\lambda}_{T_2} q' \ \wedge \ (\sigma, \lambda, (p', q')) \in E\}$$

■

On note alors E, le plus grand sous-ensemble de $Ex_\Lambda(S_1 \times_\Lambda S_2)$ satisfaisant $E \subseteq \Sigma_\Lambda(E)$, et R^Σ_Λ la relation sur $Q_1 \times Q_2$ définie par :

$$R^\Sigma_\Lambda = \{(p,q) \in Q \mid \exists \sigma \,.\, last(\sigma) = (p,q) \ \wedge \ \sigma \in E\}.$$

R^Σ_Λ coïncide alors avec le préordre de simulation $\sqsubseteq_\Lambda$ sur l'ensemble des états de $S_1 \times_\Lambda S_2$:

Proposition 5.5-1
Soient S_1 et S_2 deux systèmes de transitions étiquetées, soit S le produit synchrone $S_1 \times_\Lambda S_2$, avec $S = (Q, A, T, (q_{01}, q_{02}))$. On a alors :

$$\forall (p_0, q_0) \in Q \,.\, p_0 \sqsubseteq_\Lambda q_0 \ - \ (p_0, q_0) \in R^\Sigma_\Lambda$$

■

De même que dans le cas de le bisimulation, il est possible de restreindre ce critère de décision pour la simulation aux séquences d'exécutions *élémentaires* de $S_1 \times_\Lambda S_2$:

Définition 5.5-2

Soient $S_i = (Q_i, A_i, T_i, q_{0i})_{(i=1,2)}$ deux systèmes de transitions étiquetées, soit Λ un ensemble de langages sur $\mathcal{A}$ et soit S leur produit synchrone $S = S_1 \times_\Lambda S_2$.

Soit $\Sigma^e_\Lambda : 2^{Ex_\Lambda(S)} \longrightarrow 2^{Ex_\Lambda(S)}$ définie par :

$$\begin{aligned}\Sigma^e_\Lambda(E^e) \quad = \quad & \{\sigma \in Ex_\Lambda(S) \mid last(\sigma) = (p,q) \Rightarrow \\ & \forall \lambda \,.\, \forall p' \,.\, p \xrightarrow{\lambda}_{T_1} p' \Rightarrow \exists q' \,.\, q \xrightarrow{\lambda}_{T_2} q' \wedge ((p',q') \in \sigma \vee (\sigma, \lambda, (p',q'))) \in E^e\}\end{aligned}$$

■

On alors :

Proposition 5.5-2
Soient S_1 et S_2 deux systèmes de transitions étiquetées d'états initiaux q_{01} et q_{02}. En notant E^e le plus grand ensemble inclus dans $Ex_\Lambda(S_1 \times_\Lambda S_2)$ satisfaisant $E^e \subseteq \Sigma^e_\Lambda(E^e)$ on a :

$$S_1 \sqsubseteq_\Lambda S_2 \;-\; ((q_{01}, q_{02})) \in E^e$$

■

5.5.2 Algorithmes

A partir de la proposition 5.5-2, il est possible de déduire un premier algorithme de vérification qui consiste à énumérer l'ensemble des séquences d'exécutions élémentaires de $S_1 \times_\Lambda S_2$ afin de décider si la séquence $((q_{01}, q_{02}))$ appartient ou non à E^e.

On montre tout d'abord que, comme dans le cas de la bisimulation, l'ensemble des séquences qu'il est nécessaire de prendre en compte pour cette vérification peut être restreint en considérant uniquement les séquences d'exécution élémentaires d'un produit synchrone $S_1 \oslash S_2$.

Définition 5.5-3
Soient $S_i = (Q_i, A_i, T_i, q_{0i})_{(i=1,2)}$ deux systèmes de transitions étiquetées, soit Λ un ensemble de langages sur $\mathcal{A}$. On note $S_1 \oslash_\Lambda S_2$, le système de transitions étiquetées $S = (Q, A, T, (q_{01}, q_{02}))$ défini par :

- $Q \subseteq (Q_1 \times Q_2) \cup \{fail\}$, avec $fail \notin (Q_1 \cup Q_2)$,
- $A \subseteq \Lambda \cup \{\phi\}$, avec $\phi \notin (A_1 \cup A_2)$,
- $T \subseteq Q \times A \times Q$

et T et Q sont les plus petits ensembles obtenus par application des règles $R0$, $R1$, et $R2$ suivante :

$$(q_{01}, q_{02}) \in Q \qquad [R0]$$

$$\frac{(q_1,q_2) \in Q,\ \mathcal{A}\mathrm{ct}_\Lambda(q_1) \subseteq \mathcal{A}\mathrm{ct}_\Lambda(q_2),\ \lambda \in \Lambda,\ q_1 \xrightarrow{\lambda}_{T_1} q_1',\ q_2 \xrightarrow{\lambda}_{T_2} q_2'}{\{(q_1',q_2')\} \in Q, \{(q_1,q_2) \xrightarrow{\lambda}_T (q_1',q_2')\} \in T} \qquad [R1]$$

$$\frac{(q_1,q_2) \in Q,\ \mathcal{A}\mathrm{ct}_\Lambda(q_1) \not\subseteq \mathcal{A}\mathrm{ct}_\Lambda(q_2),}{\{fail\} \in Q, \{(q_1,q_2) \xrightarrow{\phi}_T fail\} \in T} \qquad [R2]$$

■

Remarque 5-3
Contrairement à l'opérateur $\otimes$, l'opérateur $\oslash$ n'est pas commutatif : dans le cas général,

$$S_1 \oslash S_2 \neq S_2 \oslash S_1$$

■

Clairement, la séquence d'exécution de $S_1 \times_\Lambda S_2$ réduite aux seuls états initiaux $((q_{01}, q_{02}))$ appartient toujours à $Ex_\Lambda(S_1 \oslash_\Lambda S_2)$. La proposition suivante permet alors de montrer que, pour vérifier qu'une séquence donnée de $Ex_\Lambda(S_1 \oslash_\Lambda S_2)$ n'appartient pas à E^e, il suffit en fait de ne considérer que des séquences d'exécution de ce système :

Proposition 5.5-3
Soient S_1 et S_2 deux systèmes de transitions étiquetées d'états initiaux q_{01} et q_{02}, et soit E^e le plus grand ensemble inclus dans $Ex_\Lambda(q_{01} \times_\Lambda q_{02})$ satisfaisant $E^e \subseteq \Sigma^e_\Lambda(E^e)$.

En notant $S = (Q, A, T, q_0)$ le produit $S_1 \otimes_\Lambda S_2$, on a :

(i) $\forall (p,q) \in Q \,.\, p \not\sim^1_\Lambda q \;-\; (p,q) \xrightarrow{\phi}_T fail$

(ii) Pour toute séquence $\sigma \in Ex_\Lambda(S_1 \oslash_\Lambda S_2)$ telle que $last(\sigma) = (p,q)$, on a :

$$\sigma \notin E^e \;-\; ((p,q) \xrightarrow{\phi}_T fail) \vee (\exists \lambda \,.\, \exists p' \,.\, \forall q' \,.\, ((p,q) \xrightarrow{\lambda}_T (p',q') \;\Rightarrow\; ((p',q') \notin \sigma \;\wedge\; (\sigma, \lambda, (p',q'))) \notin E^e)).$$

■

D'un point de vue plus algorithmique, la proposition 5.5-3 s'écrit alors :

Proposition 5.5-4
Soient $S_i = (Q_i, A_i, T_i, q_{0i})_{(i=(1,2))}$ deux systèmes de transitions étiquetées, soit S le produit $S_1 \oslash_\Lambda S_2$, avec $S = (Q, A, T, (q_{01}, q_{02}))$.

Pour toute séquence $\sigma \in Ex_\Lambda(S)$, telle $last(\sigma) = (q_1, q_2)$, l'algorithme suivant permet de décider si σ appartient ou non à E^e avec un coût en temps et en mémoire de $O(|T_\lambda[(q_1,q_2)]|)$:

Soit $Equiv\_List \subseteq T_{1_\Lambda}[q_1]$.
début
  $Equiv\_List\,(q_1,q_2) := \emptyset$;
  si $\neg((q_1,q_2) \xrightarrow{\phi}_T fail)$ **alors**
    pour tout λ dans Λ
      pour tout (q_1', q_2') dans $T_\lambda[(q_1,q_2)]$
        si $((q_1', q_2') \in \sigma)$ **ou** $((\sigma, \lambda, (q_1', q_2')) \in E^e)$ **alors**
          $Equiv\_List := Equiv\_List \cup \{(\lambda, q_1')\}$ **(1)**
        fsi
      fpour
    fpour
  fsi
  si $Equiv\_List = T_{1_\Lambda}[q_1]$ **alors** **(2)**
    retourner $(\sigma \in E^e)$
  sinon
    retourner $(\sigma \notin E^e)$
  fsi
fin.

■

Remarque 5-4
Outre le fait que les produits synchrones utilisées ne sont pas les mêmes ($S_1 \oslash_\Lambda S_2$ au lieu de $S_1 \otimes_\Lambda S_2$), la seule différence entre la proposition 5.5-4 et la proposition similaire introduite dans le cas de la bisimulation (proposition 5.2-3) est que seuls les successeurs de q_1 nécessite d'être examinés pour décider si $q_1 \sqsubseteq_\Lambda q_2$ (en **(1)** et **(2)**). ■

La proposition 5.5-4 permet alors de déduire un premier algorithme de vérification qui consiste à énumérer l'ensemble des séquences d'exécutions élémentaires de $S_1 \oslash_\Lambda S_2$ à l'aide d'un parcours en profondeur de ce système. Identique à l'algorithme 5-1, les coûts en temps et en mémoire de ce nouvel algorithme sont alors respectivement de $O(n!.e+1)$ et $O(d)$, où n représente le nombre d'états de $S_1 \oslash_\Lambda S_2$, et d la longueur de la plus longue de ses séquences élémentaires.

De même que pour les équivalences de bisimulation, il est possible d'obtenir un algorithme qui, toujours sur la base d'un parcours en profondeur, permette de vérifier plus efficacement si $S_1 \sqsubseteq_\Lambda S_2$, et dont le coût dépend de la structure du système de transitions étiquetées S_2 : on parlera alors de cas "déterministe" lorsque S_2 est déterministe (par rapport à Λ) et de cas "général" dans le cas contraire.

La proposition suivante permet d'établir un critère de décision plus simple dans le cas "déterministe", en montrant que les deux systèmes ne se simulent pas si et seulement si il existe une séquence d'exécution de $S_1 \oslash S_2$ qui contient l'état $fail$ ([FM91a]) :

Proposition 5.5-5
Soient S_1 et S_2 deux systèmes de transitions étiquetées tels que S_2 soit *déterministe*. En notant S le produit synchrone $S_1 \oslash_\Lambda S_2$, on a :

$$\forall (p,q) \in Q \,.\, p \not\sqsubseteq_\Lambda q \;-\; \exists \sigma \in Ex_\Lambda((p,q)) \,.\, \exists n \,.\, \sigma(n) = fail$$

■

Dans le cas "déterministe", la vérification peut par conséquent être effectuée à l'aide d'un algorithme classique de parcours du système $S_1 \oslash S_2$, et nécessite alors un coût en temps et en mémoire de $O(n)$ (*cf.* algorithme 5-2).

Dans le cas "général", la solution adoptée pour l'équivalence de bisimulation est là encore directement applicable aux relations de simulation : en effet, la proposition 5.4-1, qui montre que, sous l'hypothèse $H(S,X)$, les ensembles E et E^e coïncident, reste valide avec les nouvelles fonctions Σ_Λ et Σ^e_Λ définies dans cette section. Par suite, il est possible d'utiliser un algorithme de vérification similaire à l'algorithme 5-3, soit d'une complexité de $O(m.n)$ en temps et $O(n)$ en mémoire, où n et m représentent les nombres d'états et de transitions du système $S_1 \oslash_\Lambda S_2$.

Enfin, ces procédures de décision peuvent également être étendues aux équivalences de simulation. Plus précisément, pour vérifier que $S_1 \approx_\Lambda S_2$, deux solutions sont envisageables en pratique :

- On peut vérifier successivement que $S_1 \sqsubseteq_\Lambda S_2$ et $S_2 \sqsubseteq_\Lambda S_1$. L'inconvénient de cette solution est que, dans le cas d'une utilisation "à la volée", les systèmes de transitions devront être générés deux fois.

- On peut également "entrelacer" les deux parcours de la solution précédente en effectuant un seul parcours de l'union des produits $S_1 \oslash_\Lambda S_2$ et $S_2 \oslash_\Lambda S_1$. Cette solution présenterait alors l'avantage de ne générer qu'une seule fois les deux systèmes, mais le coût en mémoire est supérieure à celui de la solution précédente (puisque il est nécessaire de mémoriser davantage d'états).

5.6 Diagnostic

Nous montrons dans cette section comment les procédures de décision présentées dans ce chapitre peuvent être facilement modifiées pour générer une *séquence diagnostique* lorsque les systèmes de transitions étiquetées S_1 et S_2 que l'on compare ne sont pas équivalents.

5.6.1 Cas "déterministe"

Dans le cas déterministe, la proposition suivante montre que toute séquence d'exécution du produit synchrone $S_1 \otimes_\Lambda S_2$ qui conduit à un état $fail$ est une *séquence diagnostique* :

Proposition 5.6-1
Soient S_1 et S_2 deux systèmes de transitions étiquetées, d'états initiaux respectifs p_o et q_0, et tel que S_2 est déterministe, soit S le produit synchrone $S_1 \otimes_\Lambda S_2$ (avec $S = (Q, A, T, (p_0, q_0))$) et soit $\sim_\Lambda$ une équivalence de bisimulation.

On a alors :

$$\forall \sigma \,.\, \sigma \in Ex_\Lambda(S) \;\wedge\; \sigma = ((p_0,q_0), \lambda_0, \ldots, \lambda_{k-1}, (p_k,q_k)) \;\wedge\; (p_k,q_k) \xrightarrow{\phi}_T fail \;\Rightarrow\; \sigma \in Diag_\Lambda(S_1, S_2)$$

■

Preuve : D'après la proposition 5.3-1,

$$\forall (p_i,q_i) \,.\, (p_i,q_i) \in \sigma \;\Rightarrow\; p_i \not\sim_\Lambda q_i$$

De plus, par définition des ensembles Fail^k_λ, on a, si S_2 est déterministe :

$$\forall ((p,q),(p',q')) \in Q^2 \,.\, (p,q) \xrightarrow{\lambda}_T (p',q') \;\wedge\; p \not\sim_\Lambda q \;\wedge\; p' \not\sim_\Lambda q' \;\Rightarrow\; \exists k \,.\, (p',q') \in \text{Fail}^k_\lambda(p,q).$$

Par suite, σ est bien une séquence diagnostique. □.

En pratique, une telle séquence peut directement être obtenue à partir de l'algorithme 5-2 sans modifier sa complexité : lorsque l'état $fail$ est atteint, la pile $StState$ contient une séquence diagnostique.

5.6.2 Cas général

Dans le cas général, lorsque S_1 et S_2 sont non-déterministes, la proposition précédente nécessite d'être renforcée. On reprend les notations de l'algorithme 5-3, et, plus précisément, l'ensemble Non_Equiv_States, et les ensembles $Equiv\_List$ associés aux éléments de ls séquence courante. On a alors la proposition suivante :

Proposition 5.6-2
Soient S_1 et S_2 deux systèmes de transitions étiquetées, d'états initiaux respectifs p_o et q_0, soit S le produit synchrone $S_1 \otimes_\Lambda S_2$ (avec $S = (Q, A, T, (p_0, q_0))$) et soit $\sim_\Lambda$ une équivalence de bisimulation.

Soit σ une séquence d'exécution de S telle que :

$$\sigma = ((p_0,q_0), \lambda_0, \ldots, \lambda_{k-1}, (p_k,q_k)) \;\wedge\; (p_k,q_k) \xrightarrow{\phi}_T fail$$

On a alors :

$$\forall i,\, 0 \le i < k,\, \{p_{i+1}, q_{i+1}\} \not\subseteq Equiv\_List(p_i,q_i) \;\wedge\; (p_i,q_i) \in Non\_Equiv\_States \;\Rightarrow\; \sigma \in Diag_\Lambda(S_1,S_2)$$

■

Preuve : Par définition de Non_Equiv_States, on a :

$$\forall (p,q) \in Q \,.\, (p,q) \in Non\_Equiv\_States \;\Rightarrow\; p \not\sim_\Lambda q$$

D'autre part, par définition des ensembles $Equiv\_List$ et Fail^k_λ, on a :

$$\forall ((p,q),(p',q')) \in Q^2 \,.\, (p,q) \xrightarrow{\lambda}_T (p',q') \;\wedge\; \{p',q'\} \not\subseteq Equiv\_List(p,q) \;\Rightarrow\; \exists k \,.\, (p',q') \in \text{Fail}^k_\lambda(p,q).$$

Par suite, σ est bien une séquence diagnostique. □.

En pratique, une séquence diagnostique peut être obtenue à partir de l'algorithme 5-3 : il suffit alors de mémoriser lors de la vérification un "chaînage" entre des éléments de Non_Equiv_States qui

appartiennent à une séquence satisfaisant les conditions indiquées dans la proposition 5.6-2.

Remarque 5-5

1. Les propositions données dans cette section s'appliquent également aux préordres de simulation.
2. Dans tous les cas, les séquences diagnostiques obtenues à partir des algorithmes "à la volée" sont minimales pour la relation $<_p$.
3. Contrairement au algorithmes classiques présentées au chapitre 3, les procédures de décision "à la volée" ne permettent de construire qu'*une seule* séquence diagnostique. Ceci s'explique par le fait que ces dernières terminent dès le premier "état erreur" rencontré dans le produit synchrone.

■

Chapitre 6

Mise en œuvre

Ce chapitre est consacré à la mise en œuvre des procédures de décision obtenues au chapitre précédent pour les relations de simulation et de bisimulation. Plus précisément, notre objectif est d'une part de montrer la faisabilité d'une vérification "à la volée" à partir de ces algorithmes, et d'autre part de comparer les coûts en pratique avec ceux des procédures de décision classiques.

Nous montrons donc tout d'abord comment les algorithmes généraux de comparaison "à la volée" peuvent être adaptés efficacement à un certain nombre de bisimulations et simulations faibles. Nous présentons alors les structures de données que nous avons retenues pour implémenter ces algorithmes, en justifiant les choix qui les ont motivé, mais en discutant également de la mise en œuvre possible d'algorithmes plus efficaces en mémoire pour ce qui est du parcours en profondeur du produit synchrone. Enfin, nous décrivons plus précisément les fonctionnalités que nous avons ajouté au logiciel ALDÉBARAN. Nous terminons alors en discutant des coûts relatifs des différentes procédures de décision "à la volée" qui ont été implémentées et en les comparant avec ceux des procédures de décision "classiques" qui étaient déjà présentes dans ce logiciel.

6.1 La fonction "successeur"

Les algorithmes de comparaison "à la volée" présentés au chapitre précédent ont été établis dans le cas d'une relation de simulation ou de bisimulation quelconque : seule la fonction "successeur" $succ\_produit_\Lambda$, qui représente la relation de transition du produit synchrone, dépend de l'ensemble de langages Λ sur lequel est construite la relation que l'on considère.

Par suite, pour adapter ces algorithmes à une relation R_Λ donnée, il suffit en pratique d'implémenter la fonction $succ\_produit_\Lambda$ qui lui est associée, et qui, pour tout état (q_1, q_2) du produit synchrone $S_1 \otimes_\Lambda S_2$, est définie de la manière suivante :

Si R_Λ est une bisimulation,

$$\begin{array}{lcl} \mathcal{A}\mathrm{ct}_\Lambda(q_1) = \mathcal{A}\mathrm{ct}_\Lambda(q_2) & \Rightarrow & succ\_produit_\Lambda(q_1,q_2) = \{(\lambda,(q_1',q_2')) \mid (q_1 \xrightarrow{\lambda}_{T_1} q_1' \ \wedge \ q_2 \xrightarrow{\lambda}_{T_2} q_2')\} \\ \mathcal{A}\mathrm{ct}_\Lambda(q_1) \neq \mathcal{A}\mathrm{ct}_\Lambda(q_2) & \Rightarrow & succ\_produit_\Lambda(q_1,q_2) = \{(\phi, fail)\} \end{array}$$

Si R_Λ est une simulation,

$$\begin{array}{lcl} \mathcal{A}\mathrm{ct}_\Lambda(q_1) \subseteq \mathcal{A}\mathrm{ct}_\Lambda(q_2) & \Rightarrow & succ\_produit_\Lambda(q_1,q_2) = \{(\lambda,(q_1',q_2')) \mid (q_1 \xrightarrow{\lambda}_{T_1} q_1' \ \wedge \ q_2 \xrightarrow{\lambda}_{T_2} q_2')\} \\ \mathcal{A}\mathrm{ct}_\Lambda(q_1) \not\subseteq \mathcal{A}\mathrm{ct}_\Lambda(q_2) & \Rightarrow & succ\_produit_\Lambda(q_1,q_2) = \{(\phi, fail)\}. \end{array}$$

Néanmoins, l'efficacité de la procédure de décision ainsi obtenue pour R_Λ dépend largement en pratique de la complexité de l'algorithme utilisé pour implémenter la fonction $succ\_produit_\Lambda$. En particulier, pour que la vérification puisse réellement être mise en œuvre "à la volée", le calcul des successeurs d'un état donné du produit synchrone ne devra pas à lui seul nécessiter plusieurs parcours des deux systèmes de transitions étiquetées.

On propose dans les sections suivantes un certain nombre d'algorithmes pour implémenter les fonctions "successeurs" dans le cas des relations présentées au chapitre 2. Plus précisément, pour chaque relation R, on procédera de la manière suivante :

- On associe en premier lieu à R un produit synchrone, noté $\otimes_R$ (ou $\oslash_R$ dans le cas d'une simulation), dont la relation de transition puisse être calculée efficacement "à la volée". Cette étape pourra donc nécessiter de modifier la définition originale de R, et d'imposer éventuellement des contraintes sur la structure des systèmes de transitions étiquetées que l'on considère.
- On décrit alors l'algorithme de la fonction $succ\_produit_R$, qui calcule l'ensemble des successeurs d'un état donné de $S_1 \otimes_R S_2$.
- On termine en précisant les coûts en temps et en mémoire de la procédure de décision ainsi obtenue pour la relation R.

On suppose dans la suite que les systèmes de transitions S_1 et S_2 à comparer sont représentés de manière "dynamique" à l'aide des fonctions *act* et *succ*, qui, pour tout état p de S_i (avec $i = (1,2)$), calculent respectivement les ensembles de ses actions et transitions "successeurs" :

$$\begin{aligned} act\,(p) &= \mathcal{A}\mathrm{ct}(p), \\ succ\,(p) &= T_i[p], \end{aligned}$$

On suppose également que les fonctions *act* et *succ*, qui sont indépendantes des algorithmes de vérification, peuvent être calculées avec des coûts en temps et en mémoire constants.

Enfin, on utilisera dans ce chapitre les notations suivantes :

- n_1 (*resp.* n_2) représente le nombre d'états de S_1 (*resp.* de S_2) avec $n_1 \geq n_2$,
- m_1 (*resp.* m_2) représente le nombre de transitions de S_1 (*resp.* de S_2), avec $m_1 \geq m_2$,
- n_{1_a} (*resp.* n_{2_a}) représente le nombre d'états de S_1 (*resp.* de S_2) qui ont un prédécesseur par une action *visible* (i.e., différente de τ),
- m_{1_a} (*resp.* m_{2_a}) représente le nombre de transitions de S_i étiquetées par une action *visible*.

On a donc, pour $i = (1,2)$:

$$\begin{aligned} m_{i_a} &= \Sigma^{|}_{a \in \mathcal{A}}\, T_{i_a}| \\ n_{i_a} &= |\{q \in Q_i \mid \exists a \in \mathcal{A}\,.\, T_{i_a}^{-1}[q] \neq \emptyset\}| \end{aligned}$$

Remarque 6-1
m_{i_a} et n_{i_a} correspondent aux nombres d'états et de transitions de la *forme pré-normale* de S_i pour la $\tau^* a$-bisimulation (*cf.* chapitre 3, section 3.3.3). ■

6.2 Bisimulation forte

Cette relation est définie comme l'équivalence de bisimulation obtenue en considérant comme observable chacune des actions des deux systèmes S_1 et S_2, ce qui correspond à l'ensemble de langages Λ_f, avec :

$$\Lambda_f = \{\{a\} \mid a \in \mathcal{A} \cup \{\tau\}\}.$$

Par conséquent, les règles qui définissent les ensembles d'états et de transitions du produit synchrone $S = S_1 \otimes_\sim S_2$ se déduisent immédiatement de la définition générale de l'opérateur $\otimes_\Lambda$ (définition 5.2-1) en l'instanciant avec Λ_f :

$$\frac{(q_1,q_2) \in Q,\ \mathcal{A}\mathrm{ct}(q_1) = \mathcal{A}\mathrm{ct}(q_2),\ a \in \mathcal{A} \cup \{\tau\},\ q_1 \xrightarrow{a}_{T_1} q_1',\ q_2 \xrightarrow{a}_{T_2} q_2'}{\{(q_1',q_2')\} \in Q, \{(q_1,q_2) \xrightarrow{a}_T (q_1',q_2')\} \in T} \quad [R1]$$

$$\frac{(q_1,q_2) \in Q,\ \mathcal{A}\mathrm{ct}(q_1) \neq \mathcal{A}\mathrm{ct}(q_2)}{\{fail\} \in Q, \{(q_1,q_2) \xrightarrow{\phi}_T fail\} \in T} \quad [R2]$$

L'algorithme 6-1 décrit alors le calcul de la fonction $succ\_produit_\sim$ associée au système $S_1 \otimes_\sim S_2$:

Algorithme 6-1

fonction $succ\_produit_\sim\,(p,q)$
Soit $l \subseteq A \times Q$.
début
  $l := \emptyset$;
  si $act\,(p) \neq act\,(q)$ **alors** **(1)**
    $l := \{\phi, fail\}$
  sinon
    pour tout (a,p') de $succ\,(p)$ **faire**
      pour tout (a,q') de $succ\,(q)$ **faire**
        $l := l \cup (a,(p',q'))$
      fpour
    fpour
  fsi
  retourner (l)
fin.

■

Il reste à préciser les complexités en temps et en mémoire (respectivement notées $\mathcal{C}_t$ et $\mathcal{C}_m$) de la procédure de décision ainsi obtenue dans le cas de la bisimulation forte :

- Les nombres d'états et de transitions de $S_1 \otimes_\sim S_2$ sont respectivement de $n_1.n_2$ et $m_1.m_2$.
- De plus, pour tout état (p,q), le coût en temps de la fonction $succ\_produit_\sim\,(p,q)$ est majoré par $O(|T_1[p]|.|T_2[q]|)$.

Par conséquent, on a alors :

- Dans le cas déterministe, $\mathcal{C}_t = O(m_1.m_2)$ et $\mathcal{C}_m = O(n_1.n_2)$.
- Dans le cas général, $\mathcal{C}_t = O((n_1.n_2).(m_1.m_2))$ et $\mathcal{C}_m = O(n_1.n_2)$.

6.3 τ^*a-bisimulation et préordre de sûreté

Ces deux relations sont définies respectivement comme les bisimulation et simulation obtenues à partir du même ensemble de langages d'actions observables Λ_{τ^*a}, avec :

$$\Lambda_{\tau^*a} = \{\tau^*a \mid a \in \mathcal{A}\}.$$

6.3.1 τ^*a-bisimulation

De même que dans le cas de la bisimulation forte, les règles qui définissent les ensembles d'états et de transitions du produit synchrone $S_1 \otimes_{\tau^*a} S_2$ se déduisent directement de la définition de l'opérateur $\otimes_\Lambda$ en l'instanciant avec le langage Λ_{τ^*a} :

$$\frac{(q_1,q_2) \in Q,\ \mathcal{A}\mathrm{ct}_{\Lambda_{\tau^*a}}(q_1) = \mathcal{A}\mathrm{ct}_{\Lambda_{\tau^*a}}(q_2),\ a \in \mathcal{A},\ q_1 \xrightarrow{\tau^*a}_{T_1} q_1',\ q_2 \xrightarrow{\tau^*a}_{T_2} q_2'}{\{(q_1',q_2')\} \in Q, \{(q_1,q_2) \xrightarrow{a}_T (q_1',q_2')\} \in T} \quad [R1]$$

$$\frac{(q_1,q_2) \in Q,\ \mathcal{A}\mathrm{ct}_{\Lambda_{\tau^*a}}(q_1) \neq \mathcal{A}\mathrm{ct}_{\Lambda_{\tau^*a}}(q_2)}{\{fail\} \in Q, \{(q_1,q_2) \xrightarrow{\phi}_T fail\} \in T} \quad [R2]$$

Par suite, une solution immédiate pour calculer la fonction $succ\_produit_{\tau^*a}(q_1,q_2)$ consiste à employer une méthode similaire à celle utilisée dans le cas de la bisimulation forte :

1. On détermine tout d'abord les fonction $succ_{\tau^*a}$, et act_{τ^*a}, qui, pour tout état p de S_1 ou S_2, étendent respectivement les fonctions $succ(p)$ et $act(p)$ à l'ensemble de langages Λ_{τ^*a}.
2. On calcule alors, pour tout état (q_1,q_2) de Q, la fonction $succ\_produit_{\tau^*a}(q_1,q_2)$ en utilisant un algorithme analogue à l'algorithme 6-1, mais dans lequel les fonctions $succ$ et act sont respectivement remplacées par $succ_{\tau^*a}$ et act_{τ^*a}.

Il reste à expliciter le calcul des fonctions $succ_{\tau^*a}$ et act_{τ^*a} à partir des fonctions $succ$ et act. L'algorithme que nous proposons repose sur les deux phases suivantes :

1. On calcule tout d'abord l'ensemble $succ_{\tau^*}(p)$ des états accessibles à partir de p par une séquence quelconque de transitions étiquetées par τ :

$$succ_{\tau^*}(p) = \{p' \mid p \xrightarrow{\tau^*} p'\}$$

2. Les résultats des fonctions $succ_{\tau^*a}(p)$ et $act_{\tau^*a}(p)$ peuvent alors être obtenues au moyen d'un simple parcours de l'ensemble $succ_{\tau^*}(p)$:

$$\begin{aligned} succ_{\tau^*a}(p) &= \{(a,p") \in succ(p') \mid p' \in succ_{\tau^*}(p) \ \wedge\ a \neq \tau\} \\ act_{\tau^*a}(p) &= \{act(p') \mid p' \in succ_{\tau^*}(p)\} \end{aligned}$$

On donne alors l'algorithme de la fonction $tau\_explore(p)$, qui calcule l'ensemble $succ_{\tau^*}(p)$ pour tout état p de S_1 (*resp.* de S_2) en effectuant récursivement un parcours en profondeur des successeurs de p accessibles par la relation de transition T_{1_τ} ($resp.T_{2_\tau}$). De plus, afin de ne visiter qu'une seule fois chaque successeur de p, cette fonction maintient également l'ensemble Tau_Set des états déjà visités, et qui sera donc ré-initialisé à chaque nouvel appel de la fonction $succ_{\tau^*a}$.

Algorithme 6-2

fonction $tau\_explore\,(p)$
Soit l une liste d'états.
début
  $l := \emptyset$;
  si $p \notin Tau\_Set$ **alors**
    $Tau\_Set := Tau\_Set \cup \{p\}$;
    pour tout (τ, q) de $succ\,(p)$ **faire**
      $l := l \cup tau\_explore\,(q)$ (* appel récursif *)
    fpour
    $l := l \cup \{p\}$;
  fsi
  retourner (l)
fin.

■

On termine en précisant les complexités obtenues pour les algorithmes de vérification "à la volée" dans le cas de la τ^*a-bisimulation :

- Dans le cas le plus défavorable, pour tout état p de S_i, le calcul de la fonction $succ_{\tau^*a}\,(p)$ nécessite de parcourir l'ensemble des transitions de S_i. Par conséquent, les coûts en temps et en mémoire du calcul de la fonction $succ_{\tau^*a}$ sont respectivement de $O(m_i)$ et $O(n_i - n_{i_a})$.
- Par ailleurs, les nombres d'états et de transitions du produit synchrone $S_1 \otimes_{\tau^*a} S_2$ sont majorées par $m_{1_a}.m_{2_a}$ et $n_{1_a}.n_{2_a}$.

On en déduit alors :

- Dans le cas déterministe, $\mathcal{C}_t = O((m_1.m_{1_a}.m_2.m_{2_a}))$ et $\mathcal{C}_m = O(n_1.n_2)$.
- Dans le cas général, $\mathcal{C}_t = O((n_{1_a}.n_{2_a}).(m_1.m_{1_a}.m_2.m_{2_a}))$ et $\mathcal{C}_m = O(n_1.n_2)$.

Remarque 6-2

- Les coûts en temps plus élevés que dans le cas de la bisimulation forte sont dus au fait que de nombreux états de S_1 et S_2 sont générés plusieurs fois lors du parcours, puisque seuls les états qui appartiennent à $S_1 \otimes_{\tau^*a} S_2$ pourront être mémorisés dans l'ensemble $Visited$ (et donc visités une fois et une seule).
- En contrepartie, la taille mémoire nécessaire à la phase de vérification proprement dite est bien plus faible que dans le cas de la bisimulation forte, puisque les structures de données utilisées sont majorées par le nombre d'états du produit $S_1 \otimes_{\tau^*a} S_2$ (soit n_{1_a} dans le cas déterministe et n_{2_a} dans le cas général). L'essentiel du coût en mémoire de l'algorithme provient donc du calcul de la fonction $succ_{\tau^*a}$.

■

6.3.2 Préordre de sûreté

Une première solution pour adapter l'algorithme de comparaison "à la volée" au préordre de sûreté consiste à employer une approche similaire à celle décrite dans la section précédente pour la τ^*a-bisimulation.

Plus précisément, en notant S le produit synchrone $S_1 \oslash_\Lambda S_2$ obtenu pour l'ensemble de langages $\Lambda_{\tau^* a}$, l'algorithme de calcul de la fonction "successeur" $succ\_produit_{\tau^* a}$ associée à S se déduit directement de l'algorithme 6-1 en effectuant les deux modifications suivantes :

- Par définition de l'opérateur $\oslash_\Lambda$, l'instruction conditionnelle **(1)** est remplacée par :

  si $act\,(p) \not\subseteq act\,(q)$ **alors**

- Les fonctions *succ* et *act* sont respectivement remplacées par les fonctions $succ_{\tau^* a}$ et $act_{\tau^* a}$, identiques à celles définies pour la $\tau^* a$-bisimulation.

Par suite, dans le cas le plus défavorable, les coûts en temps et en mémoire de cette procédure de décision pour le préordre de sûreté sont identiques à ceux obtenus pour la $\tau^* a$-bisimulation, et, en particulier, un certain nombre d'états de S_1 ou S_2 qui n'apparaissent pas dans le produit synchrone pourront être générés plusieurs fois lors des appels successifs à la fonction $succ_{\tau^* a}$ (*cf.* remarque 6-2).

Toutefois, il existe une seconde approche dans le cas particulier où l'un des deux systèmes de transitions est τ-*free* (i.e., dépourvue de τ-transitions). En effet, dans ce cas de figure, le préordre de sûreté $\sqsubseteq_s$ coïncide avec la *relation d'implantation sûre* $\leq$, définie dans [Rod88] :

Définition 6.3-1
Soient S_1 et S_2 deux systèmes de transitions étiquetées, avec $S_i = (Q_i, A_i, T_i, q_{0i})_(i = 1,2)$. On note $\leq$ la plus grande relation incluse dans $Q_1 \times Q_2$ satisfaisant :

$p \leq q$ si et seulement si

(i) $\forall a \in \mathcal{A}\ .\ \forall p'\ .\ p \xrightarrow{a}_{T_1} p' \ \Rightarrow\ \exists q'\ .\ q \xrightarrow{a}_{T2} q' \ \wedge\ p' \leq q'$

(ii) $\forall p'\ .\ p \xrightarrow{\tau}_{T_1} p' \ \Rightarrow\ p' \leq q$

■

La relation d'implantation sûre $\leq$ peut donc être vue comme le préordre de simulation obtenu pour l'ensemble de langages Λ_s défini par :

$$\Lambda_s = \{\{a\} \mid a \in \mathcal{A}\} \cup \{\tau, \varepsilon\},$$

où ε dénote le langage vide.

La proposition suivante est issue de [Rod88] :

Proposition 6.3-1
Pour tous systèmes de transitions étiquetées S_1 et S_2 tels que S_2 soit τ-*free*, on a :

$$S_1 \leq S_2 \ -\ S_1 \sqsubseteq_s S_2$$

■

Il reste alors à décrire comment les algorithmes de comparaison "à la volée" peuvent être mis en œuvre dans le cas de la relation d'implantation sûre. On note $S = S_1 \oslash_\leq S_2$ le produit synchrone obtenu en instanciant l'opérateur $\oslash_\Lambda$ avec l'ensemble de langages Λ_s. En supposant en outre que S_2 est τ-*free*, les ensembles d'états et de transitions de S sont définis par les règles suivantes :

$$\frac{(q_1,q_2) \in Q,\ \mathcal{A}\mathrm{ct}(q_1) \subseteq \mathcal{A}\mathrm{ct}(q_2) \cup \{\tau\},\ a \in \mathcal{A},\ q_1 \xrightarrow{a}_{T_1} q_1',\ q_2 \xrightarrow{a}_{T_2} q_2'}{\{(q_1',q_2')\} \in Q, \{(q_1,q_2) \xrightarrow{a}_T (q_1',q_2')\} \in T} \qquad [R1]$$

$$\frac{(q_1,q_2) \in Q,\ \mathcal{A}\mathrm{ct}(q_1) \subseteq \mathcal{A}\mathrm{ct}(q_2) \cup \{\tau\},\ q_1 \xrightarrow{\tau}_{T_1} q_1'}{\{(q_1',q_2)\} \in Q, \{(q_1,q_2) \xrightarrow{\tau}_T (q_1',q_2)\} \in T} \qquad [R2]$$

$$\frac{(q_1,q_2)\in Q,\ \mathcal{A}\mathrm{ct}(q_1)\not\subseteq \mathcal{A}\mathrm{ct}(q_2)\cup\{\tau\}}{\{fail\}\in Q, \{(q_1,q_2)\xrightarrow{\phi}_T fail\}\in T} \quad [R3]$$

A partir de cette définition, on déduit immédiatement l'algorithme de calcul de la fonction "successeur" $succ\_produit_{\leq}$ associée à S :

Algorithme 6-3

fonction $succ\_produit_{\leq}$,(p,q)
Soit $l \subseteq A \times Q$.
début
  $l := \emptyset$;
  si $act\,(p)\ \not\subseteq\ (act\,(q)\cup\{\tau\})$ **alors**
    $l := \{\phi, fail\}$
  sinon
    pour tout (a,p') de $succ\,(p)$ **faire**
      pour tout (a,q') de $succ\,(q)$ **faire**
        $l := l \cup (a,(p',q'))$
      fpour
    fpour
    pour tout (τ,p') de $succ\,(p)$ **faire**
      $l := l \cup (\tau,(p',q))$
    fpour
  fsi
  retourner (l)
fin.

■

Il est alors facile de voir que, dans le cas le plus défavorable :

- D'une part, les nombres d'états et de transitions du produit synchrone $S_1 \oslash_{\leq} S_2$ sont identiques à ceux du produit $S_1 \otimes_{\sim} S_2$ obtenu dans le cas de la bisimulation forte.
- D'autre part le coût en temps de la fonction $succ\_produit_{\leq}\,(p,q)$ est de $O(|T_1[p]|)$ si S_2 est déterministe et $O(|T_1[p]|.|T_2[q]|)$ dans le cas contraire.

Par suite, toujours dans le cas le plus défavorable, les complexités en temps et en mémoire de cette procédure de décision sont identiques à celles obtenues pour la bisimulation forte.

Remarque 6-3
Par rapport à la relation initiale $\sqsubseteq_s$, l'intérêt de considérer la relation $\leq$ est dû au fait que sa définition prend en compte davantage de couples de $Q_1 \times Q_2$. Par conséquent, le nombre d'états du produit synchrone qui pourront être mémorisés sera plus élevé, et le nombre d'états générés plusieurs fois sera donc plus faible.

Cette solution n'est pas envisageable dans le cas de τ^*a-bisimulation, et en particulier l'équivalence de bisimulation obtenue à partir de l'ensemble de langages Λ_s ne coïncident pas avec cette dernière même lorsque l'un des deux systèmes est τ-*free*. ■

6.4 Delay bisimulation

La delay bisimulation est l'équivalence de bisimulation obtenue à partir de l'ensemble de langages Λ_d, avec

$$\Lambda_d = \{\tau^*\} \cup \Lambda_{\tau^* a}$$

Par conséquent, le produit synchrone $S = S_1 \otimes_d S2$ se déduit là encore directement de la définition de l'opérateur $\otimes_\Lambda$, et ses ensembles d'états et de transitions sont définis par les règles suivantes :

$$\frac{(q_1,q_2) \in Q,\ \mathcal{A}\mathrm{ct}_{\Lambda_{\tau^* a}}(q_1) = \mathcal{A}\mathrm{ct}_{\Lambda_{\tau^* a}}(q_2),\ a \in \mathcal{A},\ q_1 \xrightarrow{\tau^* a}_{T_1} q_1',\ q_2 \xrightarrow{\tau^* a}_{T_2} q_2'}{\{(q_1',q_2')\} \in Q, \{(q_1,q_2) \xrightarrow{a}_T (q_1',q_2')\} \in T} \quad [R1]$$

$$\frac{(q_1,q_2) \in Q,\ \mathcal{A}\mathrm{ct}_{\Lambda_{\tau^* a}}(q_1) = \mathcal{A}\mathrm{ct}_{\Lambda_{\tau^* a}}(q_2),\ q_1 \xrightarrow{\tau}_{T_1} q_1',\ q_2 \xrightarrow{\tau^*}_{T_2} q_2'}{\{(q_1',q_2')\} \in Q, \{(q_1,q_2) \xrightarrow{\tau}_T (q_1',q_2')\} \in T} \quad [R2]$$

$$\frac{(q_1,q_2) \in Q,\ \mathcal{A}\mathrm{ct}_{\Lambda_{\tau^* a}}(q_1) = \mathcal{A}\mathrm{ct}_{\Lambda_{\tau^* a}}(q_2)\ \ q_1 \xrightarrow{\tau^*}_{T_1} q_1',\ q_2 \xrightarrow{\tau}_{T_2} q_2'}{\{(q_1',q_2')\} \in Q, \{(q_1,q_2) \xrightarrow{\tau}_T (q_1',q_2')\} \in T} \quad [R3]$$

$$\frac{(q_1,q_2) \in Q,\ \mathcal{A}\mathrm{ct}_{\Lambda_{\tau^* a}}(q_1) \neq \mathcal{A}\mathrm{ct}_{\Lambda_{\tau^* a}}(q_2)}{\{fail\} \in Q, \{(q_1,q_2) \xrightarrow{\phi}_T fail\} \in T} \quad [R4]$$

Pour tout état (q_1,q_2) de S, l'algorithme de la fonction $succ\_produit_d\,(q_1,q_2)$ est obtenu à partir de celui de la bisimulation forte (algorithme 6-1) en remplaçant les fonctions *succ* et *act* par les fonctions $succ_d$ et act_d définies par :

$$\begin{aligned} act_d\,(p) &= act_{\tau^* a}\,(p) \\ succ_d\,(p) &= succ_{\tau^* a}\,(p) \cup succ_{\tau^*}\,(p) \end{aligned}$$

Par conséquent, les coûts en temps et en mémoire de la procédure de décision obtenue pour cette relation sont similaires à ceux de la $\tau^* a$-bisimulation.

6.5 Bisimulation de branchement

La bisimulation de branchement, notée $\approx_b$, a été définie comme le plus grand point-fixe de l'opérateur B^{br} (*cf.* définition 2.4-2, chapitre 2). Par suite, il ne s'agit pas à proprement parler d'une *équivalence de bisimulation* dans le sens où il n'existe pas d'ensemble de langages Λ tel que les relations $\sim_\Lambda$ et $\approx_b$ coïncident. Par conséquent, les algorithmes de comparaison "à la volée" tels qu'ils ont été décrits dans le précédent chapitre ne peuvent pas s'appliquer directement à cette relation.

Toutefois, dans le cas où l'un des deux systèmes de transitions étiquetées est τ-*free*, nous avons vu que la bisimulation de branchement coïncide avec un certain nombre d'équivalences de bisimulation, comme l'équivalence observationnelle ou la delay bisimulation (*cf.* proposition 2.6-9, chapitre 2). Il devient alors possible dans ce cas particulier de mettre en œuvre une procédure de décision "à la volée" commune à ces trois relations.

On donne tout d'abord une caractérisation plus simple de la bisimulation de branchement, obtenue à partir de sa définition générale en supposant que le système de transitions étiquetées S_2 est τ-*free* :

Proposition 6.5-1
Soient S_1 et S_2 deux systèmes de transitions étiquetées tels que S_2 soit τ-*free*. Pour tout (p,q) de $Q_1 \times Q_2$, $p \approx_b q$ si et seulement si les trois implications suivantes sont vérifiées :

$(i)\ \forall a \in A\ .\ \forall p'\ .\ p \xrightarrow{a}_{T_1} p'\ \Rightarrow\ \exists q'\ .\ q \xrightarrow{a}_{T_2} q'\ \wedge\ p' \approx_b q'$

$(ii)\ \forall p'\ .\ p \xrightarrow{\tau}_{T_1} p'\ \Rightarrow\ p' \approx_b q$

$(iii)\ \forall a \in A\ .\ \forall q'\ .\ q \xrightarrow{a}_{T_2} q'\ \Rightarrow\ \exists p'\ .\ p \xrightarrow{\tau^* a}_{T_1} p'\ \wedge\ p' \approx_b q'$

■

D'après la proposition 6.5-1, lorsque l'un des deux systèmes de transitions étiquetées est τ-*free*, la bisimulation de branchement peut être vue comme l'équivalence de bisimulation définie sur l'ensemble de langages Λ_b suivant :

$$\Lambda_b = \{\{\tau^* a\} \mid a \in \mathcal{A}\} \cup \{\tau, \varepsilon\}.$$

Par suite, une première définition pour le produit synchrone $S = S_1 \otimes_b S_2$ consisterait à instancier l'opérateur $\otimes_\Lambda$ avec l'ensemble de langages Λ_b. Toutefois, il est également possible au prix d'une caractérisation plus fine de la bisimulation de branchement de diminuer le nombre de transitions de ce produit synchrone, et par suite le coût de la procédure décision qui lui sera associée.

On définit tout d'abord pour tout système $S = (Q, A, T, q_0)$ la *relation de connexité forte* vis-à-vis de la relation de transition de S étiquetée par τ :

$$=_{sc}= \{(p, q) \in Q \times Q \mid p \xrightarrow{\tau}_T q\ \wedge\ q \xrightarrow{\tau}_T p\}$$

Il est alors facile de voir que $=_{sc}$ est une relation d'équivalence, et qu'elle est plus forte que la bisimulation de branchement :

Proposition 6.5-2

$$=_{sc} \subseteq \approx_b$$

■

On définit alors dans le lemme suivant le prédicat τ_root qui caractérise l'ensemble des états de S qui n'ont pas de τ-successeurs ou qui appartiennent à une composante fortement connexe non triviale pour la relation T_τ (i.e., une classe d'équivalence de $=_{sc}$ non réduite à un singleton) :

Lemme 6.5-1
Pour tout système de transitions étiquetées $S = (Q, A, T, q_0)$, on note τ_root le prédicat binaire défini sur Q par :

$$\tau\_root\,(p)\ \equiv\ ((T_\tau[p] = \emptyset)\ \vee\ (\exists p'\ .\ p \neq p'\ \wedge\ p =_{sc} p'))$$

On a alors :

$$\forall p \in Q, \exists p'\ .\ p \xrightarrow{\tau^*}_T p'\ \wedge\ \tau\_root\,(p')$$

■

Preuve : Clairement, pour tout état p_0 de Q on a :

- Soit $T_\tau[p_0] = \emptyset$, et par conséquent $\tau\_root\,(p_0)$ est vérifié.
- Si $T_\tau[p_0] \neq \emptyset$, on note σ le plus long chemin élémentaire de S satisfaisant :

  $$\sigma = (p_0 \xrightarrow{\tau}_T p_1 \cdots \xrightarrow{\tau}_T p_n).$$

  Par suite, Q étant fini, soit $T_\tau[p_n] = \emptyset$, soit il existe un p_i tel que $p_n \xrightarrow{\tau}_T p_i$, avec $0 \leq i \leq n$, et par conséquent p_n appartient à une composante fortement connexe non triviale pour la relation T_τ. Par conséquent dans les deux cas $\tau\_root\,(p_n)$ est vérifié.

□.

On montre alors que, lorsque la condition (iii) de la proposition 6.5-1 est vérifiée pour l'ensemble des états de Q_1 qui satisfont τ_root, elle est également vérifiée pour n'importe quel état de Q_1 :

Proposition 6.5-3
Soient S_1 et S_2 deux systèmes de transitions étiquetées tels que S_2 soit τ-*free*. Pour tout (p, q) de $Q_1 \times Q_2$, $p \approx_b q$ si et seulement si les trois implications suivantes sont vérifiées :

(i) $\forall a \in A \,.\, \forall p' \,.\, p \xrightarrow{a}_{T_1} p' \;\Rightarrow\; \exists q' \,.\, q \xrightarrow{a}_{T_2} q' \;\wedge\; p' \approx_b q'$

(ii) $\forall p' \,.\, p \xrightarrow{\tau}_{T_1} p' \;\Rightarrow\; p' \approx_b q$

$(iii)'$ $\forall (p, q) \in Q \,.\, (\tau\_root\,(p) \;\Rightarrow\; (\forall a \in A \,.\, \forall q' \,.\, (q \xrightarrow{a}_{T_2} q' \;\Rightarrow\; \exists p' \,.\, p \xrightarrow{\tau^* a}_{T_1} p' \;\wedge\; p' \approx_b q')))$

■

Preuve : En reprenant les notations introduites dans la proposition 6.5-1, il suffit de montrer :

$(iii) \;-\; (iii)'$.

Le sens '$\Rightarrow$' étant immédiat, on justifie donc uniquement le sens '$\Leftarrow$' :

Soit $(p, q) \in Q$. D'après le lemme 6.5-1,

$$\exists p_1 \,.\, p \xrightarrow{\tau^*}_{T_1} p_1 \;\wedge\; \tau\_root\,(p_1)$$

Par conséquent,

$$(iii)' \;\Rightarrow\; (\forall a \in A \,.\, \forall q' \,.\, (q \xrightarrow{a}_{T_2} q' \;\Rightarrow\; \exists p' \,.\, p_1 \xrightarrow{\tau^* a}_{T_1} p' \;\wedge\; p' \approx_b q'))$$

d'où

$$\forall a \in A \,.\, \forall q' \,.\, (q \xrightarrow{a}_{T_2} q' \;\Rightarrow\; \exists p' \,.\, p \xrightarrow{\tau^* a}_{T_1} p' \;\wedge\; p' \approx_b q')$$

et par suite l'implication (iii) est vérifiée. □.

Remarque 6-4
Compte-tenu de la proposition 6.5-2, pour tout couple (p, q) de $Q_1 \times Q_2$, pour vérifier que p et q sont équivalents vis-à-vis de la relation $\approx_b$, il suffit en pratique de vérifier qu'il existe un état p' tel que $p' =_{sc} q$ et $p' \approx_b q$:

$$\forall (p, q) \in Q_1 \times Q_2 \,.\, p \approx_b q \;-\; \exists p' \,.\, p' =_{sc} q \;\wedge\; p' \approx_b q$$

■

De la proposition 6.5-3, on déduit alors les règles qui définissent les ensembles d'états et de transitions du produit synchrone $S = S_1 \otimes_b S_2$:

$$\frac{(q_1, q_2) \in Q,\; \mathcal{A}\mathrm{ct}_{\Lambda_{\tau^* a}}(q_1) = \mathcal{A}\mathrm{ct}(q_2),\; q_1 \xrightarrow{\tau}_{T_1} q_1'}{\{(q_1', q_2)\} \in Q, \{(q_1, q_2) \xrightarrow{\tau}_T (q_1', q_2)\} \in T} \qquad [R1]$$

$$\frac{(q_1, q_2) \in Q,\; \mathcal{A}\mathrm{ct}_{\Lambda_{\tau^* a}}(q_1) = \mathcal{A}\mathrm{ct}(q_2),\; \neg\tau\_root\,(q_1),\; a \in \mathcal{A},\; q_1 \xrightarrow{a}_{T_1} q_1',\; q_2 \xrightarrow{a}_{T_2} q_2'}{\{(q_1', q_2')\} \in Q, \{(q_1, q_2) \xrightarrow{a}_T (q_1', q_2')\} \in T} \qquad [R2]$$

$$\frac{(q_1, q_2) \in Q,\; \mathcal{A}\mathrm{ct}_{\Lambda_{\tau^* a}}(q_1) = \mathcal{A}\mathrm{ct}(q_2),\; \tau\_root\,(q_1),\; a \in \mathcal{A},\; q_1 \xrightarrow{\tau^* a}_{T_1} q_1',\; q_2 \xrightarrow{a}_{T_2} q_2'}{\{(q_1', q_2')\} \in Q, \{(q_1, q_2) \xrightarrow{a}_T (q_1', q_2')\} \in T} \qquad [R3]$$

$$\frac{(q_1, q_2) \in Q,\; \mathcal{A}\mathrm{ct}_{\Lambda_{\tau^* a}}(q_1) \neq \mathcal{A}\mathrm{ct}(q_2)}{\{fail\} \in Q, \{(q_1, q_2) \xrightarrow{\phi}_T fail\} \in T} \qquad [R4]$$

A partir de cette définition, l'implémentation de la fonction "successeur" $succ\_produit_b$ associée à S est immédiate si l'on dispose des fonctions $succ_{\tau^* a}$ et $act_{\tau^* a}$ décrites dans la section 6.3.1. Néanmoins, l'inconvénient majeur de cette solution est que l'algorithme ainsi obtenu pour $succ\_produit_b$ hérite du coût en temps élevé nécessaire au calcul de ces deux fonctions. On propose donc dans la suite une seconde approche, qui ne nécessite pas de calculer explicitement la fonction $succ_{\tau^* a}$, et qui repose sur les arguments suivants :

- Lorsque q_1 n'a pas de τ-successeur, la règle $R1$ est sans objet et les ensembles $succ_{\tau^* a}(q_1)$ et $\mathcal{A}\mathrm{ct}_{\tau^* a}(q_1)$ sont directement obtenus à l'aide des fonctions $succ$ et act. Par suite, les règles $R2$ et $R3$ peuvent être implémentées de manière similaire au cas de la bisimulation forte.

- Lorsque q_1 possède des τ-successeurs, on procède de la manière suivante :

  1. On applique tout d'abord la règle $R'1$, obtenue à partir de $R1$ en remplaçant dans son prémisse la condition

     $$\mathcal{A}\mathrm{ct}_{\Lambda_{\tau^* a}}(q_1) = \mathcal{A}\mathrm{ct}(q_2)$$

     par la condition plus faible

     $$(\mathcal{A}\mathrm{ct}(q_1) - \{\tau\}) \subseteq \mathcal{A}\mathrm{ct}(q_2).$$

     La règle $R'1$ peut alors être implémentée directement à l'aide des fonctions $succ$ et act.

     De plus, on profite également des applications successives de $R'1$ au τ^*-successeurs de (q_1, q_2) pour synthétiser d'une part le résultat du prédicat $\tau\_root(q_1)$, et, d'autre part, les ensembles $S_{\tau^* a}(q_1, q_2)$ et $A_{\tau^* a}(q_1, q_2)$ définis par :

     $$\begin{aligned} S_{\tau^* a}(q_1, q_2) &= \{(a, q_1') \mid \exists q_2' \,.\, (q_1, q_2) \xrightarrow{\tau^* a}_T (q_1', q_2')\} \\ A_{\tau^* a}(q_1, q_2) &= \{a \in \mathcal{A} \mid \exists q_2' \,.\, (q_1, q_2) \xrightarrow{\tau^* a}_T (q_1', q_2')\} \end{aligned}$$

  2. On termine alors en appliquant les règles $R2$, $R3$ et $R4$ à l'aide des ensembles $S_{\tau^* a}$ et $A_{\tau^* a}$ obtenus au pas 1.

Plus formellement, le produit $S = S_1 \otimes_b S_2$ est défini par les règles suivantes, ou $R'1$ est toujours appliquée en priorité :

$$\frac{(q_1, q_2) \in Q,\ (\mathcal{A}\mathrm{ct}(q_1) - \{\tau\}) \subseteq \mathcal{A}\mathrm{ct}(q_2)\ q_1 \xrightarrow{\tau}_{T_1} q_1'}{\{(q_1', q_2)\} \in Q, \{(q_1, q_2) \xrightarrow{\tau}_T (q_1', q_2)\} \in T} \qquad [R'1]$$

$$\frac{(q_1, q_2) \in Q,\ A_{\tau^* a}(q_1) = \mathcal{A}\mathrm{ct}(q_2),\ a \in \mathcal{A},\ \neg \tau\_root(q_1),\ q_1 \xrightarrow{a}_{T_1} q_1',\ q_2 \xrightarrow{a}_{T_2} q_2'}{\{(q_1', q_2')\} \in Q, \{(q_1, q_2) \xrightarrow{a}_T (q_1', q_2')\} \in T} \qquad [R'2]$$

$$\frac{(q_1, q_2) \in Q,\ A_{\tau^* a}(q_1) = \mathcal{A}\mathrm{ct}(q_2),\ \tau\_root(q_1),\ (a, q_1') \in S_{\tau^* a}(q_1),\ q_2 \xrightarrow{a}_{T_2} q_2'}{\{(q_1', q_2')\} \in Q, \{(q_1, q_2) \xrightarrow{a}_T (q_1', q_2')\} \in T} \qquad [R'3]$$

$$\frac{(q_1, q_2) \in Q,\ A_{\tau^* a}(q_1) \neq \mathcal{A}\mathrm{ct}(q_2)}{\{fail\} \in Q, \{(q_1, q_2) \xrightarrow{\phi}_T fail\} \in T} \qquad [R'4]$$

Remarque 6-5

- La règle $R'3$ n'ajoute pas de nouveaux états au produit synchrone S mais uniquement des transitions, puisque l'ensemble des états de $S_{\tau^* a}$ ont déjà été générés lors des applications préalables des règles $R'1$ et $R'2$.
- De plus, compte-tenu de la remarque 6-4, on appliquera $R'3$ à un couple (q_1, q_2) donné uniquement lorsqu'il n'existe pas de q'_1 tel que d'une part $q_1 =_{sc} q'_1$ et que d'autre part le couple (q'_1, q_2) appartienne à la pile $StState$ (puisqu'il suffira alors de vérifier si q'_1 et q_2 sont équivalents).

■

On termine alors en donnant quelques précisions sur les complexités en temps et en mémoire de la procédure de décision ainsi obtenue pour la bisimulation de branchement :

- Le nombre d'états du produit $S_1 \otimes_b S_2$ est identique à celui du produit $S_1 \otimes_\sim S_2$ obtenu dans le cas de la bisimulation forte, soit $n_1.n_2$.

  Le nombre de ses transitions dépend du nombre d'applications de la règle $R'3$, qui est de l'ordre du nombre de composantes fortement connexes non triviales de S_1 pour la relation de transition étiquetées. Toutefois, il est toujours compris entre $m_1.m_2$ et $(m_1 + n_1.n_{1_a}).m_2$ dans le cas général.
- La complexité la fonction $succ\_produit_b\,(q_1, q_2)$ dépend également du fait que la règle $R'3$ soit ou non appliquée. Dans le cas le plus défavorable, tous les τ^a-successeurs de q_1 seront visités, ce qui implique un coût en temps de $O(n_{1_a})$.

6.6 Mise en œuvre du parcours en profondeur

Nous décrivons dans cette section les solutions qui ont été retenues pour mettre en œuvre les différents parcours en profondeur du produit synchrone utilisés par les algorithmes de vérification "à la volée". On précise donc les choix que nous avons effectués pour implémenter les structures de données nécessaires à ces algorithmes, mais on discute également des améliorations envisageables qui permettraient de réduire leur coût en mémoire.

6.6.1 Implémentation des structures de données

On rappelle tout d'abord la liste des structures de données mises en œuvre dans les algorithmes de vérification "à la volée" :

- Les piles $StState$ et $StTrans$, qui contiennent respectivement l'ensemble des états de la séquence courante et l'ensemble de leurs successeurs non encore explorés.
- L'ensemble $Visited$, qui représente l'ensemble des états visités mais n'appartenant pas à la séquence courante.

En outre, il convient également d'ajouter dans le cas "non déterministe" des structures de données plus spécifiques à la vérification, et qui sont respectivement :

- Les ensembles $Equiv\_List$, associés aux états de la séquence courante.
- Les ensembles $Roots$ et Non_Equiv_States, qui sont des sous-ensembles de l'ensemble des états déjà visités.

Le choix d'une représentation adéquate pour ces différentes structures est alors conditionné par les arguments suivants :

- La plupart des opérations effectuées sur $StState$, $Visited$, $Roots$ et Non_Equiv_States sont identiques : il s'agit essentiellement de *recherches* et d'*insertions*.
- Les états de $StState$ peuvent également être *dépilés*, mais tout état *dépilé* appartient (ou est immédiatement *inséré*) dans $Visited$.
- Les ensembles $Roots$ et Non_Equiv_States sont inclus dans $StState \cup Visited$.

Par conséquent, nous avons opté pour une représentation de l'ensemble de tous les états (q_1, q_2) du produit synchrone au sein d'une structure de donnée unique, notée $States$. Outre la valeur de (q_1, q_2), on associe alors à chaque élément de $States$ les informations suivantes :

- Le, ou les, ensembles auxquels (q_1, q_2) appartient parmi $StState$, $Visited$, $Roots$ et Non_Equiv_States. Cette information notée, $status\,(q_1, q_2)$, peut donc être codée sur un octet.
- Pour les éléments de la séquence courante (i.e., tels que $status\,(q_1, q_2) = StState$), la liste de ses successeurs non encore visités, notée $trans\,(q_1, q_2)$, avec :

  $$trans\,(q_1, q_2) \subseteq T[(q_1, q_2)].$$

  On implémente ainsi la pile $StTrans$.
- Enfin, toujours pour les éléments de la séquence courante et dans le cas "non déterministe" uniquement, la liste $Equiv\_List\,(q_1, q_2)$, avec :

  $$Equiv\_List\,(q_1, q_2) \subseteq T_1[q_1] \cup T_2[q_2].$$

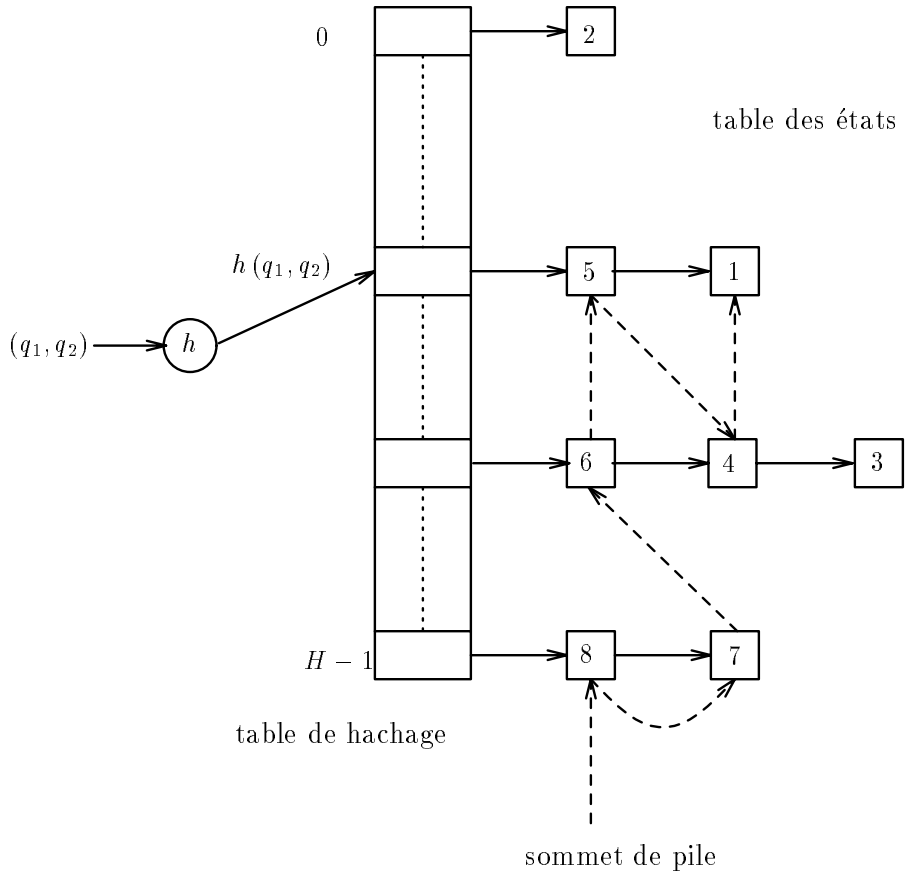
  Cet ensemble peut être représenté de manière compacte sous la forme d'un tableau de bits en codant chaque successeur de q_1 (*resp.* de q_2) par sa position dans la liste $T_1[q_1]$ (*resp.* $T_2[q_2]$). La taille de $Equiv\_List\,(q_1, q_2)$ dans le cas le plus défavorable est donc de $(|T_1[q_1]| + T_2[q_2]|)$ bits. Notons également que la mémoire utilisée par ce tableau de bits peut être libérée dès lors que l'état (q_1, q_2) est dépilé.

Il reste alors à décrire comment la structure $States$ est implémentée. Du fait que les deux systèmes de transitions étiquetées sont générés pendant le parcours en profondeur, il n'est pas possible de déterminer à l'avance un majorant pour le nombre d'états de leur produit synchrone. Par conséquent la structure $States$ devra être gérée de manière dynamique. Nous avons donc retenu la représentation suivante :

- Les éléments de $States$ sont alloués au fur et à mesure de leur génération dans une *table des états*.
- L'accès à la *table des états* se fait par la méthode dite du "hachage ouvert" (*open hashing*, [Knu73]) : à tout état (q_1, q_2) du produit synchrone, est associée une entrée dans une *table de hachage* de taille H (avec H de préférence premier), calculée au moyen d'une fonction $h\,(q_1, q_2)$ qui retourne un entier compris entre 0 et $(H - 1)$.
- Chaque entrée de la *table de hachage* pointe sur un *ensemble de collisions*, géré sous forme de liste, qui contient tous les éléments de $States$ représentant des états du produit synchrone ayant même image par h.
- Enfin, la structure de pile de $StState$ est conservée en chaînant entre-eux par ordre inverse de leur insertion l'ensemble des éléments de $States$ associés aux états de la séquence courante. Le sommet de la pile est alors représenté par un pointeur sur le dernier élément inséré.

Exemple 6-1
Sur la figure suivante, les éléments de $States$ sont numérotés de 1 à 8 par ordre d'insertion (1 est le premier état qui a été inséré, 8 le dernier). Le chaînage des éléments de la pile $StState$ est représenté par les flèches en pointillés :

- (1, 4, 5, 6, 7, 8) sont les éléments de la séquence courante,
- 2 et 3 sont des éléments de $Visited$.



■

Le coût en mémoire de la structure *States* est linéaire par rapport au nombre d'états du produit synchrone.

Les coûts en temps des opérations *insérer*, *dépiler* et *rechercher* sur cette structure sont les suivants :

- En choisissant d'*insérer* systématiquement en tête d'une liste de collisions, le coût en temps d'une insertion dépend uniquement du calcul de $h\,(q_1, q_2)$, qui est généralement fonction de la taille du codage choisi pour q_1 et q_2.
- L'opération *dépiler* s'effectue en temps constant, puisque l'élément de *States* correspondant à l'état que l'on dépile est accédé de manière directe via le pointeur de sommet de pile. Il suffit alors de modifier son *status* et de mettre à jour le chaînage de la pile.
- Enfin, *rechercher* un élément nécessite d'une part de calculer son entrée dans la table de hachage et d'autre part de parcourir la liste de collisions. En supposant que la fonction h est équitable, le coût en temps de ce parcours séquentiel est alors de $O(N/H)$, où N est le nombre d'éléments déjà insérés dans *States*.

Remarque 6-6
Théoriquement, il est possible d'obtenir une réduction logarithmique du coût de la fonction *rechercher* en représentant les ensembles de collisions non plus par des listes mais par des arbres binaires. Cette solution est décrite dans [JÓ1] pour une structure de donnée similaire à la notre. ■

6.6.2 Des algorithmes de parcours plus efficaces en mémoire

Si les solutions décrites dans la section précédente permettent bien de respecter les contraintes que nous nous étions fixées du point de vue des complexités, les procédures de décision obtenues restent cependant réservées uniquement à des couples de systèmes de transitions étiquetées dont l'ensemble des états du produit synchrone puisse être conservés en mémoire. Plus précisément, si K est la taille mémoire nécessaire pour stocker un état du produit (ainsi que l'ensemble des informations qui lui sont associées), et si M est la quantité de mémoire disponible sur la machine cible, seuls des systèmes de transitions étiquetées dont la taille du produit synchrone n'excède pas (M/K) états pourront être comparés.

Cette limitation, qui est commune à toutes les méthodes de vérification basées sur des algorithmes de parcours de systèmes de transitions étiquetées, a fait l'objet d'un certain nombre de recherches ([Hol85, Wes86, Hol89, JJ89]). A l'heure actuelle, deux solutions semblent émerger et ont été appliquées avec succès à la vérification de propriétés de logiques temporelles. Nous présentons brièvement ces deux approches en discutant de leur applicabilité au problème de la vérification à la volée de relations d'équivalence comportementales.

Réduire la taille d'un état :

La première approche a été proposée par G. Holzmann [Hol89]. Elle consiste à minimiser au mieux la mémoire nécessaire pour stocker un état du système de transitions étiquetées S que l'on parcourt (i.e., la valeur du paramètre K) en utilisant une méthode de hachage dépourvue de gestion des collisions.

En effet, si la taille de la table de hachage est suffisamment élevée (i.e., de l'ordre du nombre d'états de S), il est possible de construire une fonction de hachage h telle que le nombre de collisions obtenues soit arbitrairement petit. Par suite, en supposant que, pour tout état s de S, il n'existe pas d'états s' tel que $h(s) = h(s')$, mémoriser que l'état s a été visité revient simplement à mettre le bit $h(s)$ à 1, ce qui peut être réalisé en temps constant. En utilisant alors comme table de hachage l'ensemble de la mémoire disponible, l'algorithme de parcours obtenu peut être appliqué à des systèmes de transitions étiquetées comportant jusqu'à M états.

Cette méthode, qui a été utilisée pour la vérification de formules de logiques temporelles dont la *négation* est spécifiée par un automate de Büchi ([CVWY90]), peut également être adaptée sous certaines conditions au algorithmes de comparaison "à la volée" présentés au précédent chapitre :

- Dans le cas "déterministe", son applicabilité est immédiate puisque la comparaison repose sur un simple parcours du produit synchrone des deux systèmes de transitions étiquetées.
- Dans le cas général, quelques modifications sont nécessaires du fait qu'un certain nombre d'informations supplémentaires doivent être associées au états que l'on mémorise. Néanmoins, pour tout état (q_1, q_2) du produit synchrone :
  - L'information $status(q_1, q_2)$, qui est accédée de manière aléatoire, peut être codée sur 2 bits.
  - L'ensemble $Equiv\_List(q_1, q_2)$, qui est toujours accédé de manière séquentielle, peut être mémorisée en mémoire secondaire.

Par conséquent, toujours dans l'hypothèse où la taille de la mémoire principale disponible est de M bits, cette méthode est théoriquement applicable à des couples de systèmes de transitions étiquetées dont le produit synchrone comporte $(M/2)$ états.

Néanmoins, du point de vue de la vérification, la validité de cette approche reste fortement dépendante de la fonction de hachage choisie. En effet, même si le nombre de collisions s'avère faible, le risque qu'il existe deux états ayant même image par h ne peut pas être écarté, et il n'est donc jamais certain que *tous* les états du produit synchrone aient été visités.

Par conséquent,

- dans le cas "déterministe", cette méthode peut être vue comme une méthode de vérification *partielle* : si l'état $fail$ a été atteint pendant le parcours, alors les deux systèmes ne sont pas équivalents ; dans le cas contraire, on ne peut pas conclure.
- dans le cas général, en l'absence de garanties sur la fonction de hachage utilisée, cette méthode ne permet pas de décider (même partiellement) si deux états sont équivalents puisqu'il est indispensable que l'ensemble de leurs successeurs aient été examinés.

Réduire le nombre des états mémorisés :

La seconde approche, évoquée par Holzmann dans [Hol87], puis reprise et approfondie par C. Jard et T. Jéron [JJ89, JJ91], nous parait plus adaptée au problème de la vérification "à la volée". Elle consiste à majorer le nombre d'états mémorisés lors du parcours en profondeur d'un système de transitions étiquetées en fonction de la quantité de mémoire disponible.

En effet, nous avons vu que dans le cas d'une exploration en profondeur, l'ensemble des états mémorisés se partageait en deux classes :

- Les états de la séquence courante, qui correspondent à la pile $StState$,
- Les états déjà visités mais n'appartenant plus à la séquence courante, qui correspondent à l'ensemble $Visited$.

Le point important est que seuls les éléments de $StState$ sont nécessaires pour assurer que le parcours se termine, l'ensemble $Visited$ servant uniquement de réduire son coût en temps en visitant au plus une fois chaque état du système de transitions étiquetées. Il n'est par conséquent pas essentiel de mémoriser cet ensemble dans son intégralité.

Les algorithmes proposés dans [Hol87] et [JJ89] consistent alors à gérer au mieux la quantité de mémoire disponible en mémorisant d'une part la pile $StState$, et, d'autre part, le plus grand sous-ensemble *possible* de $Visited$. Plus précisément, lorsqu'un nouvel état doit être ajouté à $StState$, on procède de la manière suivante :

- Si un nouvel élément peut être alloué dans la table des états, alors l'insertion s'effectue de manière usuelle.
- S'il n'est pas possible d'allouer un nouvel élément faute de mémoire suffisante, alors on *remplace* un élément de $Visited$ choisi de manière *aléatoire* par l'état à insérer.

Les expériences effectuées par les auteurs de ces algorithmes montrent que le surcoût en temps dus au états *remplacés* de $Visited$ (et donc susceptibles d'être revisités) reste raisonnable en pratique, y compris lorsque l'on considère des systèmes de transitions étiquetées dont le nombre d'états peut être plusieurs fois supérieur au facteur (M/K).

Cette méthode, qui a été appliquée à la vérification de formules de logiques temporelles spécifiées au moyen d'automates de Büchi ([J91]), peut également être adaptée au algorithmes de comparaison "à

la volée" :

- Dans le cas "déterministe", là encore son applicabilité est immédiate.
- Dans le cas général, certaines précautions sont nécessaires du point de vue du remplacement des états du fait que la terminaison des parcours successifs est basée sur la croissance strict de l'ensemble Non_Equiv_States. Par conséquent, seuls des états de l'ensemble $(Visited - Non\_Equiv\_States)$ pourront être remplacés.

Enfin, notons également que cette technique peut être mise en œuvre à partir de structures de données similaires à celles que nous avons décrites dans la section 6.6.1. Seule la table des états nécessiterait alors une gestion particulière afin de permettre le remplacement (*cf.* [J91]).

6.7 Applications au sein de l'outil ALDÉBARAN

Les procédures de décision décrites dans ce chapitre ont été implémentées au sein de l'outil ALDÉBARAN. Plus précisément, nous avons envisagé deux types d'applications pour ces algorithmes :

- d'une part – comme les algorithmes "classiques" – ils peuvent être utilisés pour comparer des systèmes de transitions étiquetées déjà construits.
- d'autre part – et c'est là leur originalité – ils permettent également de vérifier un programme "à la volée", sans générer au préalable le système de transitions étiquetées qui le représente.

Nous rappelons d'abord brièvement les fonctionnalités d'ALDÉBARAN, puis nous présentons plus en détails ces deux applications.

6.7.1 ALDÉBARAN

ALDÉBARAN [Fer88, Fer89] est un outil de vérification qui permet de *réduire* et *comparer* des systèmes de transitions étiquetées modulo un certain nombre de relations d'équivalence : sont notamment implémentées la bisimulation forte, l'équivalence observationnelle, la $\tau^* a$-bisimulation, l'équivalence de sûreté et l'équivalence par modèle d'acceptation.

Les systèmes de transitions étiquetées traités par ALDÉBARAN doivent être représentés "en extension" par l'intermédiaire de fichiers ASCII qui contiennent la liste de leurs états et de leur transitions (selon le *format aldébaran* [Fer88]). En pratique, ces systèmes peuvent être générés à partir de programmes LOTOS par le compilateur CÆSAR.

Les algorithmes de minimisation et de comparaison mis en œuvre dans ce logiciel sont les algorithmes "classiques" qui correspondent à ces relations, et qui ont été décrits au chapitre 3. En particulier les calculs de raffinements de partitions sont effectués en utilisant l'algorithme de Paige & Tarjan.

Enfin, ALDÉBARAN fournit également un diagnostic lorsque les deux systèmes de transitions étiquetées que l'on compare ne sont pas identifiés par la relation d'équivalence considérée. Ce diagnostic se présente alors sous la forme de couples de séquences d'exécutions de ces deux systèmes, minimales pour la relation d'ordre $<_\cap$, selon le formalisme proposé au chapitre 4.

6.7.2 De nouvelles procédures de décision

Les algorithmes décrits dans ce chapitre ont tout d'abord été mis en œuvre dans le contexte classique de la comparaison de systèmes de transitions étiquetées déjà générés. L'objectif de cette première

application était essentiellement d'expérimenter ces algorithmes sur des systèmes de grandes tailles – qui pouvaient être produits par le compilateur CÆSAR – et de confronter ces résultats avec ceux obtenus dans des conditions similaires en utilisant les algorithmes "classiques".

Dans ce but, nous avons donc implémenté les procédures de décision correspondant respectivement à la bisimulation forte, la τ^*a-bisimulation, le préordre et l'équivalence de sûreté, la delay bisimulation, et, dans le cas où l'un des deux systèmes est τ-*free*, à la bisimulation de branchement (ou, ce qui revient au même dans ce cas de figure, à l'équivalence observationnelle).

Les résultats obtenus à partir d'un certain nombre d'exemples ont montré l'intérêt de cette première application, qui a notamment permis d'enrichir ALDÉBARAN sur deux principaux points :

- En premier lieu, elle offre une solution alternative aux procédures de décision qui étaient déjà implémentées dans ce logiciel. Il est en effet apparu que, dans le cas d'un certain nombre de relations comme la τ^*a-bisimulation, l'équivalence de sûreté et l'équivalence observationnelle (*cf.* section 6.8), ce type d'algorithmes se comportait mieux en pratique que les algorithmes classiques du point de vue de l'efficacité en temps et en mémoire.

- D'autre part, elle permet également de traiter de nouvelles relations comme la delay bisimulation et surtout le préordre de sûreté, pour lequel les algorithmes classiques sont beaucoup plus coûteux.

Enfin, du point de vue de la mise en œuvre, on peut noter un certain nombre de points qui sont spécifiques à cette application :

- Du fait que les relations de transition des deux systèmes de transitions étiquetées sont représentées en mémoire, l'implémentation des fonctions *succ* et *act* est immédiate.

- De plus, puisque le nombre des états des deux systèmes est connu à l'avance, il est possible de coder chacun de ces états par un *entier* unique. Par suite la structure de donnée générale présentée dans la section 6.6 pour représenter les états du produit synchrone peut être simplifiée, ainsi que la fonction de hachage.

6.7.3 Vérification "à la volée"

Compte-tenu des résultats obtenus à partir de cette première application, qui ont montré que ce type d'approche était tout à fait réaliste en pratique, nous avons pu envisagé d'implémenter réellement ces procédures de décision dans le contexte d'une vérification "à la volée".

Description des programmes

Pour des raisons conjoncturelles, il n'était pas possible de modifier le compilateur CÆSAR et d'effectuer la vérification directement à partir de spécifications LOTOS. Nous avons donc adopté un formalisme intermédiaire pour décrire les programmes, basé sur les *processus communicants* : intuitivement, un programme P est constitué d'un ensemble fini de processus séquentiels élémentaires P_i qui sont exécutés en parallèle et qui se synchronisent par *rendez-vous* sur certaines de leurs actions. La manière dont ont lieu ces synchronisations est alors défini à l'aide d'une *expression de composition* qui indique quels sont les processus qui communiquent entre-eux, et par l'intermédiaire de quelles actions.

Ce type de formalisme est très fréquemment utilisé pour décrire des programmes parallèles, et en général, seul le langage qui permet de définir l'expression de composition diffère. Dans ce travail, nous avons retenu un langage restreint à deux opérateurs, un opérateur de *composition parallèle* et un opérateur d'*abstraction*, qui ont intuitivement les sémantiques suivantes :

opérateur de composition :
si P_1 et P_2 sont deux processus et si C est un sous-ensemble des actions que peuvent effectuer ces deux processus, alors "$P_1|[C]|P_2$" est le processus résultant de la mise en parallèle de P_1 et P_2 tel que :

- Les actions de C sont effectuées de manière *synchrone* par P_1 et P_2 : C représente l'*ensemble de communication* entre ces deux processus.
- Les actions qui n'appartiennent pas à C sont effectuées librement, de manière *asynchrone*, par chacun de ces deux processus.

Cet opérateur correspond en fait à l'opérateur de composition parallèle |[ ]| du langage LOTOS.

opérateur d'abstraction :
si P_1 est un processus, et si L est un sous-ensemble des actions qu'il peut effectuer, alors "*hide* L *in* P" est le processus obtenu à partir de P en renommant l'ensemble de ses actions qui appartiennent à L en l'*action interne* τ.

Il correspond donc à l'opérateur **hide** de LOTOS.

Enfin, nous supposons que chaque processus séquentiel élémentaire P_i est représenté par un système de transitions étiquetées qui modélise son comportement.

Plus formellement, la syntaxe d'une expression de composition est donnée par la grammaire BNF suivante, où les terminaux *idf_ste* et *liste_action* dénotent respectivement un identificateur de système de transitions étiquetées et une liste d'actions (éventuellement vide) :

expression	::=	expression |[*liste-actions*]| terme | terme
terme	::=	*hide liste_actions in* terme | facteur
facteur	::=	*idf_ste* | (expression)

La sémantique des opérateurs d'abstraction et de composition parallèle est alors définie par les règles suivantes :

Soient $S_i = (Q_i, A_i, T_i, q_{0i})_{i=1,2}$ deux systèmes de transitions étiquetées, avec $Q_1 \cap Q_2 = \emptyset$, et soient L et C des parties de $\mathcal{A}$.

- *hide* S_1 *in* L est le système de transitions étiquetées $S = (Q_1, A, T, q_{01})$ avec T et A définis par :

$$\frac{p \xrightarrow{a}_{T_1} p',\ a \in (A_1 \cap L)}{\tau \in A,\ p \xrightarrow{\tau}_{T} p'} \quad [A1]$$

$$\frac{p \xrightarrow{a}_{T_1} p',\ a \in (A_1 - L)}{a \in A,\ p \xrightarrow{a}_{T} p',} \quad [A2]$$

- $S_1|[C]|S_2$ est le ste $S = (Q, A_1 \cup A_2, T, (q_{01}, q_{02}))$ avec T et Q définis par :

$$\frac{(q_1,q_2) \in Q,\ q_1 \xrightarrow{a}_{T_1} q_1',\ q_2 \xrightarrow{a}_{T_2} q_2',\ a \in C}{(q_1',q_2') \in Q,\ (q_1,q_2) \xrightarrow{a}_{T} (q_1',q_2')} \quad [C1]$$

$$\frac{(q_1,q_2) \in Q,\ q_1 \xrightarrow{a}_{T_1} q_1',\ a \notin C}{(q_1',q_2) \in Q,\ (q_1,q_2) \xrightarrow{a}_{T} (q_1',q_2)} \quad [C2]$$

$$\frac{(q_1,q_2) \in Q,\ q_2 \xrightarrow{a}_{T_2} q_2',\ a \notin C}{(q_1,q_2') \in Q,\ (q_1,q_2) \xrightarrow{a}_{T} (q_1,q_2')} \quad [C3]$$

Traduction des programmes LOTOS

Outre sa simplicité, la principale raison qui a motivé le choix de ce langage de description est qu'il permet de représenter facilement des programmes écrits en LOTOS. L'intérêt est alors de pouvoir confronter les procédures de décision "à la volée" à de vrais exemples décrits en LOTOS puis traduits dans ce langage.

Nous ne rappelons pas ici la définition du langage LOTOS (qui peut être trouvée dans [BB88]), et nous ne présentons pas non plus de manière formelle la traduction d'un programme LOTOS vers un programme décrit par un ensemble de processus communicants, mais nous donnons une idée intuitive de la transformation que nous avons utilisée en pratique, et qui est illustrée par l'exemple fournis en annexe (*cf.* annexe B).

- La première étape consiste à transformer le programme LOTOS original en un programme LOTOS "équivalent" qui respecte la grammaire que nous avons donnée pour une expression de composition, à la seule différence que le terminal *idf_ste* représente non plus un identificateur de systèmes de transitions étiquetées mais une définition de *processus* (ou *process*), au sens LOTOS du terme.

  Théoriquement, tout programme LOTOS peut être réécrit sous une forme qui respecte cette grammaire, quitte à ce que le programme entier soit lui-même contenu dans une définition de *processus*. En pratique, il existe en général différentes décompositions possibles, selon le niveau de granularité choisi pour définir les *processus* LOTOS que l'on considère.

- La seconde étape consiste alors à générer un système de transitions étiquetées correspondant à chacun des *processus* définis dans l'étape 1, ce qui a été réalisé en utilisant le compilateur CÆSAR. On obtient ainsi une représentation du programme LOTOS initial sous forme de processus communicants qui est conforme à la grammaire que nous avons donnée.

Remarque 6-7

1. La transformation purement *syntaxique* décrite dans la première étape pourrait être automatisée. Toutefois, il apparaît en pratique que le choix de la décomposition en *processus* élémentaires a beaucoup d'influence sur la taille des systèmes de transitions étiquetées obtenus. En général, une bonne connaissance de la sémantique du programme permet d'obtenir "à la main" une décomposition efficace. Une solution possible consisterait peut-être à adopter une approche interactive, dans laquelle l'utilisateur pourrait procéder par "essais-erreurs".

2. Si cette transformation est toujours possible en théorie, elle peut toutefois très bien s'avérer irréalisable en pratique en raison de la taille des systèmes de transitions étiquetées engendrés pour chaque *processus* élémentaire, qui peut même éventuellement dépasser celle du système de transitions étiquetées correspondant au programme complet. Néanmoins, pour les exemples que nous avons traités, il a toujours été possible d'obtenir une décomposition pour laquelle ce problème ne se produisait pas.

■

Enfin, il s'est également avéré intéressant en pratique de *minimiser* les systèmes de transitions obtenus à partir des processus élémentaires en les remplaçant par leurs systèmes quotients modulo une relation d'équivalence R_{min}. La relation R_{min} choisie doit alors satisfaire deux contraintes :

- R_{min} doit être une *congruence* pour les opérateurs de restriction et de composition parallèle. Plus précisément, si S_1, S_2 et S sont des systèmes de transitions étiquetées, on doit avoir :

  (i) $\forall L \,.\, S_1 \; R_{min} \; S_2 \;\Rightarrow\; (\text{hide } L \text{ in } S_1) \; R_{min} \; (\text{hide } L \text{ in } S_2)$
  (ii) $\forall C \,.\, S_1 \; R_{min} \; S_2 \;\Rightarrow\; (S_1 |[C]| S) \; R_{min} \; (S_1 |[C]| S)$

- R_{min} doit être *plus forte* que la relation R_{verif} utilisée lors de la vérification pour comparer le programme à sa spécification comportementale :

$$R_{min} \subseteq R_{verif}$$

Mise en œuvre

On termine en donnant quelques indications sur la manière dont nous nous avons mis en œuvre cette application, et plus précisément, sur l'implémentation des fonctions *succ* et *act* à partir d'une description du programme sous forme de processus communicants.

Le système de transitions étiquetées $S = (Q, A, T, q_0)$ qui modélise le comportement d'un programme est défini par une expression de la forme

$$S = \mathcal{F}(S_1, S_2, \ldots, S_N)$$

telle que :

- Les S_i $(1 \leq i \leq N)$ sont des systèmes de transitions étiquetées donnés, avec $S_i = (Q_i, A_i, T_i, q_{0i})$.
- $\mathcal{F}$ est une *expression de composition*, construite à l'aide des opérateurs de composition parallèle et d'abstraction dont la syntaxe et la sémantique ont été précisées ci-dessus.

Dans la suite, on pose $I = [1, .., N]$.

On décrit tout d'abord comment sont représentés les états de S. Par définition de Q (règles $C1$, $C2$, $C3$), on a :

$$Q \subseteq Q_1 \times Q_2 \times \ldots \times Q_N.$$

Par conséquent, chaque élément $(q_1, q_2, \ldots, q_n)$ de Q sera codé par la chaîne de bits obtenue en concaténant les valeurs des q_i qui le constituent (codage *par champs*). Le coût en mémoire de cette représentation sera donc de

$$O(|Q|.(\log_2 |Q_1| + \log_2 |Q_2| + \ldots + \log_2 |Q_N|)).$$

Il reste alors à expliciter le calcul de la relation de transition T en fonction des relations T_i de chacun des S_i :

On associe à toute expression $\mathcal{F}$ un *arbre abstrait* $T(\mathcal{F})$, qui comporte quatre types de nœuds :

- Des nœuds feuilles qui correspondent au identificateur de système de transitions étiquetées *idf_ste* (i.e., à des éléments de I).
- Des nœuds feuilles qui correspondent au liste d'actions *liste_actions* (i.e., à des parties de $\mathcal{A}$).
- Des nœuds binaires n qui correspondent à l'opérateur *hide* L *in* $\mathcal{F}_1$. On pose alors :

$$\begin{aligned} n.liste &= L \\ n.fils &= T(\mathcal{F}_1) \\ n.op &= abstraction \end{aligned}$$

- Des nœud ternaires n qui correspondent à l'opérateur $\mathcal{F}_1|[C]|\mathcal{F}_2$. On pose alors :

$$\begin{aligned} n.liste &= C \\ n.fils\_d &= T(\mathcal{F}_1) \\ n.fils\_g &= T(\mathcal{F}_2) \\ n.op &= composition \end{aligned}$$

Pour tout nœud n, on note $ste(n)$ l'ensemble de ses feuilles qui représentent des identificateurs de système de transitions étiquetées.

On définit alors pour chaque action a de A (avec $A = A_1 \cup A_2 \ldots \cup A_N$), l'ensemble $Synchro\,(a)$ qui représente les ensembles d'identificateurs de systèmes de transitions étiquetées pour lesquels a est une action *synchrone* (*cf.* exemple 6-2). Plus précisément $Synchro\,(a)$ correspond à l'ensemble des parties de I définies par :

$$Synchro\,(a) = \{B_k \mid B_k \subseteq I \ \wedge \ B_k \neq \emptyset\}$$

avec :

- Deux éléments d'une même partie B_k appartiennent aux feuilles gauches et droites d'un nœud "composition parallèle" qui les synchronisent pour l'action a :

  $(i,j) \in B_k \ \Rightarrow$
  $(\exists n \,.\, n \in T\,(\mathcal{F}) \ \wedge \ n.op = composition \ \wedge \ a \in n.liste \ \wedge \ i \in ste\,(n.fils\_g) \ \wedge \ j \in ste\,(n.fils\_d))$

- Les éléments de $Synchro\,(a)$ sont *maximaux* vis-à-vis de la relation d'inclusion $\subseteq$:

  $$\forall (B_k, B_l) \,.\, \{B_k, B_l\} \subseteq Synchro\,(a) \ \Rightarrow \ (B_k \not\subseteq B_l \ \wedge \ B_l \not\subseteq B_k)$$

- Tout élément i de I appartient à un élément de $Synchro\,(a)$, éventuellement réduit au singleton $\{i\}$ (i.e., $Synchro\,(a)$ définit un *recouvrement* de l'ensemble I) :

  $$\forall i \,.\, i \in I \ \Rightarrow \ (\exists B_k \,.\, B_k \in Synchro\,(a) \ \wedge \ i \in B_k)$$

En pratique, les ensembles $Synchro\,(a)$ associés à chaque action a peuvent être synthétisés lors d'un parcours de l'arbre abstrait.

Enfin, on définit également l'ensemble V qui représente les actions de A différentes de τ et non renommées en τ par l'opérateur d'abstraction (*cf.* exemple 6-2) :

$$V = \{a \in A \mid a \neq \tau \wedge \ \nexists n \in T\,(\mathcal{F}) \,.\, (n.op = abstraction \ \wedge \ a \in n.liste)\}$$

Là encore, cet ensemble peut être synthétisé lors d'un parcours de l'arbre abstrait.

La relation de transition T de S est alors définie à l'aide de ces deux ensembles en fonction des relations T_i, de la manière suivante :

Pour tout $(q_1, \ldots, q_N)$ de Q,

(i) $(q_1, \ldots, q_N) \xrightarrow{a}_T (q'_1, \ldots, q'_N)$ $-$
$(a \in V \ \wedge \ \exists B_k \,.\, (B_k \in Synchro\,(a) \ \wedge \ (\forall i \in B_k \,.\, q_i \xrightarrow{a}_{T_i} q'_i) \ \wedge \ (\forall i \in (I - B_k) \,.\, q_i = q'_i)))$

(ii) $(q_1, \ldots, q_N) \xrightarrow{\tau}_T (q'_1, \ldots, q'_N)$ $-$
$(\exists a, B_k \,.\, (a \in (A - V) \ \wedge \ B_k \in Synchro\,(a) \ \wedge \ (\forall i \in B_k \,.\, q_i \xrightarrow{a}_{T_i} q'_i) \ \wedge \ (\forall i \in (I - B_k) \,.\, q_i = q'_i)))$

Les fonctions *succ* et *act* sont alors calculées à partir de cette définition de T.

Exemple 6-2
Soient S_1, S_2 et S_3, des systèmes de transitions étiquetées avec $S_i = (Q_i, A_i, T_i, q_{0i})_{(i \in \{1,2,3\})}$. et tels que $A_1 = \{a, b, c\}$, $A_2 = \{a, b, d\}$, et $A_3 = \{a, b\}$.

On considère l'expression de composition $\mathcal{F}$, définie par :

$$\mathcal{F} = hide \ \ b \ \ in \ \ ((S_1 \ |[\,b\,]| \ S_2) \ |[\,a, b\,]| \ S_3)$$

On a alors :

- L'ensemble des identificateurs de systèmes de transitions étiquetées est l'ensemble

  $$I = \{1, 2, 3\}$$

- L'ensemble des actions visibles est l'ensemble

  $$V = \{a, c, d\}$$

- Pour chaque action, les ensembles *Synchro* sont définis par :

$$\begin{aligned} Synchro\,(a) &= \{\{1,3\},\{2,3\}\} \\ Synchro\,(b) &= \{\{1,2,3\}\} \\ Synchro\,(c) &= \{\{1\}\} \\ Synchro\,(d) &= \{\{2\}\} \end{aligned}$$

On note $S = (Q, A, T, q_0)$ le système de transitions étiquetées associé à l'expression $\mathcal{F}$, avec $Q \subseteq Q_1 \times Q_2 \times Q_3$.

A partir de ces ensembles, la relation de transition T est alors défini de la manière suivante :

$$\begin{aligned} (q_1,q_2,q_3) \xrightarrow{a}_T (q'_1,q'_2,q'_3) \quad - \quad & (q_1 \xrightarrow{a}_{T_1} q'_1 \ \wedge\ q_3 \xrightarrow{a}_{T_1} q'_3 \ \wedge\ q_2 = q'_2) \\ & \vee \\ & (q_2 \xrightarrow{a}_{T_2} q'_2 \ \wedge\ q_3 \xrightarrow{a}_{T_1} q'_3 \ \wedge\ q_1 = q'_1) \end{aligned}$$

$$(q_1,q_2,q_3) \xrightarrow{c}_T (q'_1,q'_2,q'_3) \quad - \quad q_1 \xrightarrow{c}_{T_1} q'_1 \ \wedge\ q_2 = q'_2 \ \wedge\ q_3 = q'_3$$

$$(q_1,q_2,q_3) \xrightarrow{d}_T (q'_1,q'_2,q'_3) \quad - \quad q_2 \xrightarrow{d}_{T_2} q'_2 \ \wedge\ q_1 = q'_1 \ \wedge\ q_3 = q'_3$$

$$(q_1,q_2,q_3) \xrightarrow{\tau}_T (q'_1,q'_2,q'_3) \quad - \quad q_1 \xrightarrow{b}_{T_1} q'_1 \ \wedge\ q_2 \xrightarrow{b}_{T_2} q'_2 \ \wedge\ q_3 \xrightarrow{b}_{T_3} q'_3$$

■

6.8 Comparaison avec les méthodes classiques

L'ensemble des procédures de décision implémentées dans ALDÉBARAN ont pu être expérimentées sur un nombre important d'exemples, qui ont permis de comparer assez objectivement les comportements des deux approches, l'approche classique, et l'approche "à la volée". On résume dans cette section les résultats de cette analyse, en essayant de les justifier.

Plus précisément, on rappelle tout d'abord, pour chaque relation, les complexités en temps et en mémoire des deux types d'algorithmes lorsque l'on se place dans des conditions qui correspondent à celles que l'on rencontre réellement en pratique. On donne alors les valeurs des temps de comparaison relevés sur un exemple particulier, le scheduler de Milner, qui représentent toutefois assez bien les résultats obtenus pour l'ensemble des applications que nous avons considérées.

6.8.1 Comparaison des complexités

Dans la pratique, les systèmes de transitions étiquetées que l'on est amené à comparer lorsque l'on vérifie une spécification comportementale sont rarement quelconques. En effet, si l'on note S_1 le système qui représente le programme et S_2 celui qui représente sa spécification, on a d'une manière générale les propriétés suivantes :

- Les nombres d'états et de transitions de S_2 sont *négligeables* devant ceux de S_1 :

$$n_1 >> n_2 \quad \text{et} \quad m_1 >> m_2$$

- Le nombre m_{1_a} de transitions de S_1 qui sont étiquetées par une action visible est *faible* devant le nombre total de transitions de S_1, et par suite, le nombre n_{1_a} d'états de S_1 qui ont un

prédécesseur par une transition étiquetées par une action visible est *faible* devant le nombre total d'états de ce système.

Sur les exemples que nous avons traités avec ALDÉBARAN, on relève en moyenne :

$$m_1 \simeq 10.m_{1_a} \quad \text{et} \quad n_1 \simeq 10.n_{1_a}$$

- Enfin, sur tous les exemples que nous avons rencontrés, les systèmes de transitions étiquetées S_2 étaient *τ-free* et comportaient très peu de *non-déterminisme* (i.e, le nombre des états qui avaient plusieurs successeurs par une même action était faible devant le nombre total d'états).

Compte-tenu de ces hypothèses, il est possible de simplifier les valeurs des complexités que nous avons données pour les différentes procédures de décision "à la volée" et, par suite, de les comparer plus facilement avec celles des procédures classiques. Plus précisément, nous effectuons les approximations suivantes :

- Concernant les algorithmes classiques, les nombres n et m d'états et de transitions de l'*union* de S_1 et S_2 sont peu différents des nombres d'états et de transitions de S_1 :

$$n = (n_1 + n_2) \simeq n_1 \quad \text{et} \quad m = (m_1 + m_2) \simeq m_1$$

- Concernant les algorithmes "à la volée", les nombres n' et m' d'états et de transitions du *produit synchrone* $S_1 \otimes_\Lambda S_2$ vérifient :

$$n_1 \leq n' \leq n_1.n_2 \quad \text{et} \quad m_1 \leq m' \leq m_1.m_2$$

On approchera donc ces deux inégalités à l'aide d'une constante k telle que :

$$n' = k.n \quad \text{et} \quad m' = k.m$$

On a donc dans tous les cas, $k \leq m_2$.

Les complexités en temps et en mémoire obtenues pour ces deux algorithmes sont alors résumées dans les tableaux suivants :

Relation	Algorithmes "à la volée"	Algorithmes classiques
Bisimulation Forte	$k^2.m.n$	$m.\log n$
Préordre de Sûreté	$k^2.m.n$	$m.n_a^2 + m^2$
Equivalence Observationnelle	$k^2.(m + n.n_a).n_a$	$(n - n_a)^3 + n^2.\log n$
Bisimulation de Branchement	$k^2.(m + n.n_a).n_a$	$m.n$
$\tau^* a$-bisimulation	$k^2.m.m_a$	$m.n_a^2 + n_a^2.\log n_a$

Comparaison des complexités en temps

Relation	Algorithmes "à la volée"	Algorithmes classiques
Bisimulation Forte	$k.(m + n)$	$m + n$
Préordre de Sûreté	$k.(m + n)$	$m + n_a^2 + n$
Equivalence Observationnelle	$k.(m + n.n_a)$	$m + n^2 + n$
Bisimulation de Branchement	$k.(m + n.n_a)$	$m + n$
$\tau^* a$-bisimulation	$k.(m + n)$	$m + n_a^2 + n$

Comparaison des complexités en mémoire

Remarque 6-8

1. Toutes ces valeurs sont données *dans le cas le plus défavorable*, en tenant compte des approximations mentionnées au début de cette section. En particulier, pour les algorithmes "à la volée", les complexités en temps concernent l'algorithme général (et non le cas "déterministe"), et les complexités en mémoire supposent que la plus longue séquence élémentaire du produit synchrone de S_1 et S_2 contient l'ensemble des états de ce système.

2. Pour l'équivalence observationnelle et la bisimulation de branchement, nous avons considéré le même algorithme "à la volée" (décrit dans la section 6.5), car S_2 est supposé τ-*free*.

3. Pour le préordre de sûreté, les complexités qui sont données pour l'approche classique font référence à l'algorithme proposé par Cleaveland et Steffen [CS91] (*cf.* chapitre 3, section 3.5).

■

6.8.2 Un exemple : "le scheduler de Milner"

Le *scheduler de Milner* [Mil80] a été utilisé comme *benchmark* pour de nombreux algorithmes de comparaison ou de minimisation des systèmes de transitions étiquetées. ([Fer88, GV90, FM90, EFT91, Qin91]). Bien qu'il ne s'agisse pas à proprement parler d'un "vrai" exemple, il présente néanmoins l'intérêt de pouvoir obtenir des systèmes de tailles variables, ce qui permet d'avoir une idée de la variation du temps de comparaison lorsque la taille des systèmes augmente.

Nous rappelons tout d'abord en quoi consiste cet exemple, puis nous donnons les temps de vérification obtenus en considérant successivement les procédures de décision classiques puis les procédures de décision "à la volée". Dans les deux cas, cette vérification a été menée à partir des systèmes de transitions *déjà construits*. En effet, l'objectif étant de comparer ces deux types d'algorithmes dans des conditions similaires, il était nécessaire de les appliquer aux mêmes systèmes de transitions étiquetées, et avec les mêmes contraintes d'accès à leurs relations de transition.

Description du problème

On considère N processus p_i $(1 \leq i \leq N)$, organisés en anneau, qui doivent accomplir une tâche donnée de façon répétitive. Informellement, les contraintes qui doivent être respectées par le système résultant de la mise en parallèle de ces N processus sont les suivantes :

1. Les N tâches associées à chacun des p_i doivent être exécutées de manière cyclique.

2. Aucun p_i ne peut recommencer une exécution de la tâche qui lui est attribuée avant d'en avoir terminé avec l'exécution précédente.

Le problème est alors de décrire un contrôleur qui garantisse le respect de ces contraintes.

La solution proposée par Milner consiste à faire circuler un jeton sur l'anneau pour indiquer aux processus p_i s'ils ont ou non l'autorisation de commencer leur tâche. Plus précisément, un p_i accepte le jeton lorsqu'il est disponible, et il commence alors l'exécution de la tâche qui lui est attribuée avant d'offrir le jeton à son successeur sur l'anneau (l'exécution de cette tâche pouvant ou non être terminée).

En CCS, cette solution est décrite par la mise en parallèle de N termes, traditionnellement appelés $Cycler$, qui représentent le comportement de chacun des p_i, et d'un terme $Starter$ qui permet de lancer la première exécution de la tâche associé à p_1 :

$$\begin{aligned} Starter &= g_1.Nil \\ Cycler_1 &= g_1.a_1.(b_1.\overline{g_2}.Cycler_1 + \overline{g_2}.b_1.Cycler_1) \end{aligned}$$

$$
\begin{aligned}
Cycler_2 &= g_2.a_2.(b_2.\overline{g_3}.Cycler_2 + \overline{g_3}.b_2.Cycler_2) \\
&\vdots \\
Cycler_N &= g_N.a_N.(b_N.\overline{g_1}.Cycler_N + \overline{g_1}.b_N.Cycler_N)
\end{aligned}
$$

Intuitivement, les actions $(g_i, \overline{g_i})$ correspondent à une transmission du jeton entre les processus p_{i-1} et p_i, et les actions a_i (*resp.* b_i) dénotent le début (*resp.* la fin) de l'exécution par p_i de la tâche qui lui est attribuée.

L'expression complète du Scheduler en CCS est alors :

$$Scheduler = (Starter \,|\, Cycler_1 \,|\, Cycler_2 \,|\, \ldots \,|\, Cycler_N) \backslash \{g_1, \ldots, g_N\}$$

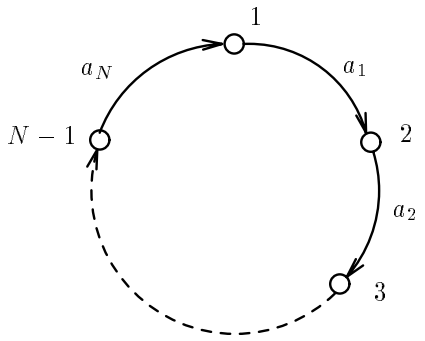
Le tableau suivant donne pour différentes valeurs de N les tailles des systèmes de transitions étiquetées S_1 générés par CÆSAR à partir d'une traduction en LOTOS de cette expression CCS.

N	nombre d'états	nombre de transitions
8	3073	13825
9	6913	34561
10	15361	84481
11	33793	202753
12	73729	479233

Tailles des systèmes de transitions étiquetées

Vérification

On s'intéresse à la vérification de la première contrainte de la spécification informelle, c'est à dire au fait que toutes les tâches associées aux processus p_i doivent être exécutées de manière cyclique. Plus formellement, cette spécification signifie que, lorsque seules les actions a_i sont visibles, le comportement observable du Scheduler est représenté par le système de transitions étiquetées S_2 de la figure suivante, qui comporte N états et N transitions :



Remarque 6-9
Le système de transitions étiquetées S_2 est bien τ-*free* et déterministe. ■

Les deux tableaux suivants donnent les temps de comparaison de S_1 et S_2 obtenus en fonction de N avec les procédures de décision classiques dans le cas de l'équivalence observationnelle et dans le cas de la $\tau^* a$-bisimulation. Nous avons détaillé les coûts des deux phases de ces algorithmes qui sont respectivement :

- Le calcul des formes pré-normales de S_1 et S_2 (colonne "Fermeture Transitive"), qui dépend donc de la relation d'équivalence que l'on considère.
- La comparaison de ces formes pré-normales modulo la bisimulation forte (colonne "Paige & Tarjan"), mise en œuvre avec le même algorithme de raffinement de partition pour les deux relations.

La troisième colonne représente alors le temps total qui a été nécessaire à la comparaison. Enfin, le symbole "—" signifie qu'il y a eu saturation de l'espace mémoire lors du calcul de le forme pré-normale.

N	Fermeture Transitive	Paige & Tarjan	Coût Total
8	134.200	4.417	138.617
9	918.917	15.333	934.250
10	—	—	—

Equivalence Observationnelle (algorithme classique)

N	Fermeture Transitive	Paige & Tarjan	Coût Total
8	4.367	0.533	4.900
9	20.350	1.650	22.000
10	111.517	5.333	116.850
11	581.017	7.600	588.617
12	—	—	—

$\tau^* a$-bisimulation (algorithme classique)

Le tableau suivant donne alors les temps obtenus lorsqu'on effectue la comparaison à l'aide des procédures de décision "à la volée", obtenus dans des conditions similaires au résultats précédents (i.e, à partir des systèmes de transitions étiquetées déjà construits).

N	$\tau^* a$-bisimulation	Bisimulation de Branchement	Préordre de sûreté
8	1.650	0.950	0.483
9	5.900	2.850	1.233
10	21.800	7.300	2.933
11	83.950	15.333	6.883
12	341.283	39.300	16.383

Algorithmes "à la volée"

Remarque 6-10
Toutes les valeurs présentées dans ces tableaux ont été obtenus sur un SUN 4 (SparcStation) comportant 12 Mo de mémoire vive. Ils correspondent à des *temps utilisateurs* retournés par la fonction standard *times*() du système UNIX. ■

6.8.3 Discussion

Les résultats obtenus dans le cas du *scheduler* nous ont paru assez généraux, et ils reflètent assez bien les comportements de ces algorithmes observés sur d'autres exemples. Il est par conséquent possible de dégager un certain nombre de points qui nous paraissent intéressants :

- Les différences de complexités entre les procédures de décision classiques dédiées respectivement à l'équivalence observationnelle et à la τ^*a-bisimulation apparaissent clairement. En particulier, l'explosion combinatoire du nombre de transitions lors du calcul de la forme pré-normale dans le cas de l'équivalence observationnelle rend cette relation assez peu intéressante en pratique, même pour des exemples de taille moyenne.

- La comparaison entre les procédures de décision classiques et les procédures de décision "à la volée" montre bien tout l'intérêt de cette dernière approche, y compris lorsque l'on effectue la comparaison à partir de systèmes de transitions étiquetées générés au préalable. Pour ce type d'application, le gain en temps et en mémoire est alors essentiellement dû au fait que ces algorithmes ne nécessitent pas de calculer une forme pré-normale.

- Les comportements relatifs de ces trois procédures de décision "à la volée" peuvent différer selon les exemples : si la comparaison vis-à-vis du préordre de sûreté s'est toujours avérée la plus efficace (lorsque S_2 est τ-*free*), il n'est pas possible d'établir un classement en ce qui concerne la τ^*a-bisimulation et la bisimulation de branchement.

Toutefois, il serait injuste de ne pas mentionner également les deux limitations que nous avons relevées à partir de notre implémentation pour ces procédures de décision "à la volée" :

- D'une part, si les hypothèses effectuées sur la structure de S_2 ne sont plus vérifiées (en particulier si S_2 comporte plusieurs milliers d'états, avec un "non-déterminisme" important) alors les procédures de décision classiques — lorsqu'il est possible de les mettre en œuvre — peuvent s'avérer plus efficaces du point de vue du coût en temps que les procédures de décision "à la volée". Nous n'avons toutefois jamais rencontré un tel cas de figure lorsque lors de la vérification de "vrais" exemples.

- Enfin, il est certain que le fait de mémoriser tous les états entraîne nécessairement une limite à la taille des exemples que l'on peut traiter (même si cette limite est bien supérieure à celle obtenue avec les procédures de décision classique). Il semble donc nécessaire de mettre en œuvre des techniques de gestion mémoire encore plus efficaces, et en particulier le parcours en profondeur avec remplacement des états ([JJ89], *cf.* section 6.6.2).

Conclusion

Bilan

Ce travail est centré sur les méthodes pratiques de vérification de spécifications comportementales et plus particulièrement sur l'étude d'algorithmes efficaces pour comparer deux systèmes de transitions étiquetées modulo une relation de simulation ou de bisimulation.

Les procédures de décision classiques, qui reposent sur des algorithmes de raffinement de partitions, nécessitent de *construire* et *mémoriser* dans leur ensemble les relations de transition des deux systèmes. De plus, certaines d'entre elles mettent également en œuvre des algorithmes de fermeture transitive sur ces relations de transition (phase de *saturation*), ce qui peut augmenter leur taille dans des proportions importantes. Dans tous les cas, ces procédures de décision sont donc limitées à la comparaison de systèmes de transitions étiquetées de taille moyenne (de l'ordre d'une centaine de milliers d'états) ce qui les rend insuffisantes en pratique.

En nous inspirant d'une approche qui avait été mise en œuvre pour la vérification de certains types de spécification logiques, nous avons proposé un algorithme original, qui permet d'effectuer la comparaison "à la volée", c'est à dire *au fur et à mesure* de la génération des deux systèmes. Plus précisément, nous avons montré que l'existence d'une relation de bisimulation (ou de simulation) entre deux systèmes de transition étiquetées S_1 et S_2 pouvait être exprimée à l'aide d'un critère sur l'ensemble des séquences d'exécution élémentaires d'un *produit synchrone* S entre S_1 et S_2. Par suite, décider si ces deux systèmes sont en relation revient à décider si ce critère est vérifié ou non. La procédure de décision dépend alors de la *structure* de S_1 et S_2 :

- Si l'un au moins des deux systèmes de transitions étiquetées est *déterministe*, le critère d'équivalence peut être simplifié et la vérification se ramène à une simple analyse d'accessibilité sur le produit synchrone. Le coût de cet algorithme est alors de $O(m)$ en temps et de $O(n)$ en mémoire, où n et m dénotent respectivement les nombres d'états et de transitions du produit synchrone.

- Lorsque les deux systèmes de transitions étiquetées sont non déterministes, le critère d'équivalence ne peut pas toujours être vérifié efficacement par un parcours unique du produit synchrone. La solution que nous avons retenue consiste alors à effectuer un parcours en profondeur *sous hypothèses* (i.e., en supposant certains états *à priori* équivalents), et à vérifier après coup si ces hypothèses sont ou non valides. Dans le cas contraire, un nouveau parcours en profondeur sera nécessaire, mais l'hypothèse initiale pourra être remplacée par une hypothèse *strictement plus faible* (au sens de l'implication logique). Par suite, après un nombre fini d'itérations de cette procédure, les hypothèses effectuées deviennent valides. Le coût total de l'algorithme est alors dans le cas le plus défavorable de $O(m.n)$ en temps et de $O(n)$ en mémoire.

Quelle que soit la relation de bisimulation (ou de simulation) considérée, la vérification repose donc sur un même algorithme de parcours en profondeur du produit synchrone. En pratique, ce parcours peut être mis en œuvre au fur et à mesure de la génération de S_1 et S_2 à partir d'une représentation "dynamique" des relations de transition de ces deux systèmes, sous la forme d'une fonction "successeur". Seule cette dernière dépend alors de la relation de bisimulation (ou de simulation) considérée, et en particulier du langage d'actions observables sur lequel cette relation est définie.

Nous avons donc pu adapter cet algorithme général à un certain nombre de relations intéressantes en pratique, comme la bisimulation et la simulation forte, le préordre et l'équivalence de sûreté, la τ^*a-bisimulation, la delay bisimulation, et, dans le cas où l'un des deux systèmes de transitions étiquetées ne contient pas de τ-transitions, la bisimulation de branchement (qui coïncide alors avec l'équivalence observationnelle).

Les procédures de décision ainsi obtenues ont été implémentées au sein du logiciel de vérification ALDÉBARAN [Fer88, FM91b], en considérant deux types d'applications :

- La comparaison de systèmes de transitions étiquetées déjà construits, ce qui constitue alors une solution alternative aux procédures de décision classiques.

- La vérification "à la volée", sans construction préalable des deux systèmes de transitions étiquetées. La comparaison est alors effectuée à partir d'une représentation intermédiaire du programme, sous forme d'une *expression de composition* entre systèmes de transitions étiquetées élémentaires, qui peut être elle-même dérivée d'un programme LOTOS.

L'expérimentation de ces algorithmes sur un grand nombre d'exemples montre le réalisme de notre approche. Ces bons résultats sont essentiellement dus au fait que les systèmes de transitions étiquetées que l'on est amené à comparer lors de la vérification de spécifications comportementales ne sont pas quelconques. En effet, le système de transitions étiquetées associé à la spécification ne contient généralement pas de τ-transitions et il est "assez déterministe". Par suite, la vérification a toujours pu être réalisée en effectuant un nombre réduit de parcours du produit synchrone.

Plus précisément, les exemples traités ont permis de mettre en valeur les principaux avantages des procédures de décision qui ont été proposées :

- Contrairement à certaines procédures de décision classiques, elles ne comportent pas de phase de saturation. Par suite, mises en œuvre dans des conditions similaires (i.e., à partir des systèmes déjà construits), elles s'avèrent plus efficaces en temps et en mémoire que ces dernières. A titre d'exemple, le plus gros système de transitions étiquetées traité par ALDÉBARAN avec ce type d'approche (vis-à-vis de la τ^*a-bisimulation) avoisine le million d'états, alors que la procédure de décision classique pour cette relation reste limitée à une centaine de milliers d'états.

- Elles ont pu être appliquées à des relations basées sur le préordre de simulation. Dans ce cas, les complexités obtenues sont alors bien meilleures que celles des procédures de décision classiques, et plus particulièrement lorsque l'un des deux systèmes de transitions étiquetées ne comporte pas de τ-transitions.

- De plus, l'implémentation "à la volée" a permis d'effectuer des vérifications sur des programmes LOTOS pour lesquels il n'était pas possible de générer de systèmes de transitions étiquetées, ce qui n'aurait pas pu être obtenu avec une approche classique.

Enfin, nous nous sommes également intéressés à une composante indispensable d'un outil de vérification qui est l'élaboration d'un *diagnostic*, permettant d'expliquer, le cas échéant, pourquoi deux systèmes de transitions étiquetées ne sont pas équivalents. Nous avons choisi un formalisme qui nous a semblé adapté en pratique et qui repose sur les séquences d'exécution des deux systèmes : une *séquence diagnostique* est un couple de séquences d'exécution qui, à partir des états initiaux des deux

systèmes, conduisent par les mêmes actions observables à des états où la non-équivalence apparaît clairement (i.e, il existe une action observable que seul l'un des deux systèmes peut effectuer).

Nous avons alors présenté trois relations d'ordre sur l'ensemble des séquences d'exécutions d'un système de transitions étiquetées, ainsi que les algorithmes qui permettent la construction des *séquences diagnostiques minimales* qui leur sont associées. Ces algorithmes ont été intégrés aux procédures de décision implémentées dans ALDÉBARAN.

Perspectives

Concernant l'algorithme de vérification "à la volée" que nous avons présenté, un certain nombre de points mériterait d'être approfondis :

- En premier lieu, l'algorithme lui-même semble pouvoir être amélioré dans le cas où les deux systèmes de transitions étiquetées sont non-déterministes. En effet, la condition sur laquelle on se base pour décider, à la fin d'un parcours en profondeur, si les hypothèses faites lors de ce parcours sont valides ou non n'est pas une condition nécessaire. Par conséquent, certains des parcours peuvent être effectués inutilement. Il serait donc intéressant de pouvoir affaiblir cette condition, par exemple en ordonnant certains éléments de l'ensemble des états visités.

- Un second prolongement envisageable serait l'extension de cet algorithme à d'autres relations que celles que nous avons présentées, en particulier à des relations qui ne sont ni des simulations ni des bisimulation. L'équivalence de test [NH84], ou les équivalences par modèles d'acceptation ou de refus [BHR84, GS86, BKO88], pour lesquelles les procédures de décision classiques sont coûteuses, pourraient être des candidates intéressantes. Par ailleurs, les relations de bisimulation que nous avons traitées à l'aide de cet algorithme peuvent facilement être étendues en prenant en compte des *prédicats de divergence* lors de la vérification.

- Enfin, une question plus générale concerne la définition des propriétés qu'il est possible de vérifier "à la volée", par un parcours du système de transitions étiquetées associé au programme à vérifier. La caractérisation de cette classe de propriétés permettrait sans doute de trouver de nouvelles applications à ce type d'algorithme.

De façon similaire, deux directions nous semblent prioritaires pour améliorer l'implémentation que nous avons réalisée :

- Tout d'abord, il est souhaitable de pouvoir effectuer directement la vérification sur des programmes décrits en LOTOS. Deux approches sont alors envisageables :

  - L'implémentation, éventuellement sous une forme interactive, de l'algorithme que nous avons utilisé pour traduire un programme LOTOS vers une description sous forme d'une expression de composition entre systèmes de transitions étiquetées.
  - La modification du compilateur CÆSAR [GS90a], afin qu'il fournisse une représentation "dynamique" du système de transitions étiquetées, sous la forme d'une fonction "successeur".

- D'autre part, les exemples traités ont montré qu'il était nécessaire d'adopter des techniques plus performantes du point de vue du coût en mémoire pour mettre en œuvre le parcours en profondeur du produit synchrone. En particulier, nous avons vu que la solution du parcours avec *remplacement des états visités* proposé par C. Jard et T. Jéron [JJ91] peut s'appliquer à notre algorithme de vérification.

Concrètement, ce travail d'implémentation s'inscrit dans le contexte plus général d'un outil de vérification "ouvert" (appelé OPEN-CÆSAR), constitué de trois types de composants (au sens de l'outil VESAR [Gar90]) :

- Le premier module, engendré automatiquement par le compilateur CÆSAR, fournit une représentation "dynamique" du système de transitions étiquetées associé au programme à vérifier.
- Le second module, interchangeable, décrit l'algorithme de vérification "à la volée" à mettre en œuvre sur ce programme (i.e., le type de parcours à effectuer). Outre une spécification comportementale, il est d'ores et déjà possible d'envisager de vérifier d'autres propriétés comme l'absence de blocage, ou une spécification logique donnée sous forme d'automates de Büchi [CVWY90, JJ91] (cette liste n'étant pas exhaustive).
- Le troisième module, également interchangeable, définit alors les structures de données utilisées lors du parcours. En particulier, il est possible de mettre en œuvre le parcours en profondeur *avec remplacement* mentionné ci-dessus, ou la technique de représentation des états proposée par G. Holzmann [Hol89].

Nous terminons en replaçant ce travail dans le cadre du *problème de l'explosion d'états* évoqué en introduction. L'expérience montre en effet que l'ensemble des systèmes de transitions étiquetées qui modélisent le comportement d'un programme parallèle peut être partitionné en trois classes, auxquelles correspondent un certain nombre de méthodes pratiques pour la vérification de spécifications comportementales :

- La première classe contient les systèmes dont la relation de transition peut être construite et mémorisée dans son ensemble. Toutes les procédures de décision peuvent alors s'appliquer à ces systèmes et la vérification "à la volée" constitue une approche plus efficace que les méthodes classiques lorsque celles-ci mettent en œuvre des algorithmes de calculs de fermetures transitives.
- La deuxième classe, assez large, contient les systèmes pour lesquels il est possible d'*énumérer* efficacement les états, mais dont la relation de transition ne peut être mémorisée. Pour tous ces systèmes, et contrairement aux méthodes classiques, les procédures de décision "à la volée" que nous avons proposées permettent encore de vérifier une spécification comportementale modulo une relation de bisimulation.
- Enfin, il semble clair qu'il existe également une troisième classe de systèmes, pour lesquels une énumération exhaustive n'est pas envisageable, y compris avec des techniques de gestion mémoire efficaces. Dans ce cas, seules les méthodes qui permettent de générer directement un *modèle réduit* peuvent s'appliquer. Dans le cas des relations de bisimulation, il existe un algorithme [BFH90a] qui permet de construire directement le *quotient* d'un système de transitions étiquetées à partir d'une représentation symbolique d'un programme, mais son application à des langages comme LOTOS reste encore un problème ouvert.

Annexe A

Preuves des lemmes 4.7.1 et 4.7.2

A.1 Lemme 4.7.1

Lemme 1
Soit $S = (Q, A, T, q_0)$ un système de transitions étiquetées et soit $\sim_\Lambda$ une équivalence de bisimulation. Soit $(\rho_i, W_i)_{(i \in \mathcal{N})}$ la suite définie par :

- $(\rho_0, W_0) = (\{Q\}, \{Q\})$,
- $\forall i \geq 0$, $(\rho_{i+1}, W_{i+1}) = \theta'_\Lambda(\rho_i, W_i)$.

Pour tout $k \geq 0$, toute classe de ρ_k issue de la décomposition d'une classe de ρ_{k-1} servira de partitionneur lors du calcul de ρ_{k+1} :

$$(\forall X \,.\, X \in \rho_k \ \wedge \ X \notin \rho_{k-1}) \ \Rightarrow \ X \in Part\,(\rho_{k+1}).$$

■

Preuve : La preuve du lemme est la conjonction des propriétés suivantes :

(i) Toute classe X de ρ_k issue de la décomposition d'une classe de ρ_{k-1} apparaît dans W_{k+1}, soit en tant que bloc simple, soit contenue dans l'union des feuilles d'un bloc arbre.

(ii) Toute classe X de ρ_k qui apparaît dans W_{k+1}, soit en tant que bloc simple, soit contenue dans l'union des feuilles d'un bloc arbre sert de partitionneur lors du calcul de ρ_{k+1} (i.e, $X \in Part\,(\rho_{k+1})$).

Il reste alors à montrer ces deux propriétés :

(i) On note $(\rho'_i, W'_i, W''_i)_{(i \in \mathcal{N})}$ la suite, de limite $(\rho'_r, \emptyset, W''_r)$, définie par (*cf.* définition de θ') :

- $(\rho'_0, W'_0, W''_0) = (\rho_{k-1}, W_{k-1}, \emptyset)$,
- $\forall i \geq 0$, $(\rho'_{i+1}, W'_{i+1}, W''_{i+1}) = \theta''_\Lambda(\rho'_i, W'_i, W''_i, x)$ avec $x \in W'_i$.

On a alors $\rho_k = \rho'_r$ et $W_k = W''_r$.

Or, pour tout X de ρ_k issue de la décomposition d'une classe de ρ_{k-1} on a :

$$\exists i,\ i \leq r \,.\, X \notin \rho'_i \ \wedge \ X \in \rho'_{i+1}$$

On note alors x le partitionneur de W'_i utilisé pour raffiner ρ'_i ($\rho'_{i+1} = Ref_\Lambda(\rho'_i, x)$ et Y la classe de ρ'_i qui contenait X ($X \subset Y$). Différents cas sont à envisager :

- $X \notin W'_i \ \wedge \ X \notin W''_i$:

  Le bloc arbre représentant la décomposition de Y est alors ajouté à W''_{i+1} (par (3) de la définition de θ'').

- $X \notin W'_i \ \wedge \ X \in W''_i$:

  * si Y est un bloc simple, Y est remplacé dans W''_{i+1} par les classes obtenues lors de son raffinement (et donc en particulier par X), d'après 2) de la définition de θ'').
  * si Y est la feuille d'un bloc arbre n alors n est remplacé dans W''_{i+1} par un arbre n' obtenu à partir de n en remplaçant chaque feuille par un nœud représentant sa décomposition (par (4) de la définition de θ'').

- $X \in W'_i \ \wedge \ X \notin W''_i$:

  Là encore,

  * si Y est un bloc simple, alors les classes obtenues lors de son raffinement (et en particulier X) sont ajoutées à W''_{i+1} (par (2) de la définition de θ'').
  * si Y est la feuille d'un bloc arbre n alors n est remplacé dans W'_{i+1} par un arbre n' obtenu à partir de n en remplaçant chacune des ses feuilles par un nœud représentant sa décomposition (par (4) de la définition de θ''). Par suite, lorsque n' sera le partitionneur courant, d'après (5), les fils gauches et droits n'_1 et n'_2 de n' seront insérés dans W''_{i+p+1} (avec $p > 0$), et X sera bien contenu dans l'union des feuilles de n'_1 ou n'_2.

- $X \in W'_i \ \wedge \ X \in W''_i$:

  Il est facile de voir par induction sur i que cette situation ne peut se produire :

  * $W'_0 = \rho_{k-1} \ \wedge \ W''_0 = \emptyset$,
  * et $\forall i \geq 0, \forall Y \ . \ (Y \notin W'_i \ \vee \ Y \notin W''_i) \ \Rightarrow \ (Y \notin W'_{i+1} \ \vee \ Y \notin W''_{i+1})$.

Par conséquent, toute classe X de ρ_k qui est issue de la décomposition d'une classe de ρ_{k-1} est présente dans W''_r (et donc dans W_k), soit en tant que bloc simple, soit contenue dans l'union des feuilles d'un bloc arbre.

(*ii*) On considère la suite $(\rho'_i, W'_i, W''_i)_{(i \in \mathcal{N})}$, de limite $(\rho'_r, \emptyset, W''_r)$, définie par :

- $(\rho'_0, W'_0, W''_0) = (\rho_k, W_k, \emptyset)$,
- $\forall i \geq 0, (\rho'_{i+1}, W'_{i+1}, W''_{i+1}) = \theta''_\Lambda(\rho'_i, W'_i, W''_i, x)$ avec $x \in W'_i$.

On a $\rho_{k+1} = \rho'_r$ et $W_{k+1} = W''_r$.

Il reste alors à montrer que, pour toute classe X, bloc simple de W'_0 ou contenue dans l'union des feuilles d'un bloc arbre de W'_0, il existe un entier i tel que :

$$(\rho''_{i+1}, W'_{i+1}, W''_{i+1}) = \theta''_\Lambda(\rho''_i, W'_i, W''_i, x) \text{ avec } x \in W'_i \ \wedge \ X \in Classe\,(x).$$

Or, pour tout i, on a, par définition de θ'' :

- Si X est un bloc simple de W'_i alors :
  * si X n'est pas le partitionneur courant, alors, que la classe X soit ou non décomposé, $X \in W'_{i+1}$ (par (1) et (2)).
  * si X est le partitionneur courant x, alors $X \in Classe\,(x)$.
- Si X est contenue dans l'union des feuilles d'un bloc arbre x de W'_i alors :
  * si x n'est pas le partitionneur courant, alors, que X soit ou non décomposée, X sera contenue dans l'union des feuilles du bloc arbre x' qui remplace x dans W'_{i+1} (par (4), avec éventuellement $x = x'$).

$*$ si x est le partitionneur courant, alors $X \in Classe\,(x)$.

Par suite, du fait que $(|W'_i|)$ décroît strictement, tout élément de W'_i devient nécessairement partitionneur, et par conséquent, toute classe X, bloc simple de de W_k ou contenue dans l'union des feuilles d'un bloc arbre de W_k, appartient à $Part\,(\rho_{k+1})$.

□.

A.2 Lemme 4.7.2

Lemme 2
Soit $S = (Q, A, T, q_0)$ un système de transitions étiquetées et soit $\sim_\Lambda$ une équivalence de bisimulation. Soit $(\rho_i, W_i)_{(i \in \mathcal{N})}$ la suite définie par :

- $(\rho_0, W_0) = (\{Q\}, \{Q\})$,
- $\forall i \geq 0$, $(\rho_{i+1}, W_{i+1}) = \theta'_\Lambda(\rho_i, W_i)$.

Pour tout $k \geq 0$, toute classe utilisée pour raffiner la partition ρ_k lors du calcul de ρ_{k+1} est soit une classe de ρ_k, soit une union de classes de ρ_k :

$$X \in Part\,(\rho_{k+1}) \;\Rightarrow\; \exists X_1, \ldots, X_n \in \rho_k \;.\; X = \bigcup_{i=1}^{n} X_i.$$

■

Preuve : Là encore, la preuve du lemme est obtenue par conjonction de deux propriétés :

(i) Les classes utilisées pour raffiner ρ_k lors du calcul de ρ_{k+1} (i.e, les éléments de $Part\,(\rho_{k+1})$) sont soit des blocs simples de W_k soit des blocs contenus dans l'union des feuilles du fils gauche ou droit d'un bloc arbre de W_k.

(ii) Les blocs simples de W_k et les classes contenues dans l'union des feuilles des fils gauche et droit d'un bloc arbre de W_k sont soit des classes de ρ_k, soit des unions de classes de ρ_k.

Il reste alors à justifier (i) et (ii) :

(i) On note $(\rho'_i, W'_i, W''_i)_{(i \in \mathcal{N})}$ la suite, de limite $(\rho'_r, \emptyset, W''_r)$, définie par :

- $(\rho'_0, W'_0, W''_0) = (\rho_k, W_k, \emptyset)$,
- $\forall i \geq 0$, $(\rho'_{i+1}, W'_{i+1}, W''_{i+1}) = \theta''_\Lambda(\rho'_i, W'_i, W''_i, x)$ avec $x \in W'_i$.

On a alors $\rho_k = \rho'_{r+1}$ et $W_k = W''_{r+1}$ (définition de θ').

Soit $X \in Part\,(\rho_{k+1})$. Il existe donc un i, $0 \leq i \leq r$, tel que :

$$(\rho'_{i+1}, W'_{i+1}, W''_{i+1}) = \theta''_\Lambda(\rho''_i, W'_i, W''_i, x) \text{ avec } x \in W'_i \;\wedge\; X \in Classe\,(x).$$

Or, par construction des W'_i (*cf.* définition de θ''), on a :

- Si x est un bloc simple de W'_i, alors x est nécessairement un bloc simple de W'_0.
- Si x est un bloc arbre n, alors, soit n appartient à W'_0, soit n est le fils gauche ou droit d'un bloc arbre de W'_0, soit n a été obtenu à partir d'un bloc arbre n' de W'_0 en remplaçant des feuilles de n' par des nœuds représentant leur décomposition.

Dans tous les cas, X est un bloc simple de W'_0 (et donc de W_k) ou est inclu dans l'union des feuilles du fils gauche ou droit d'un bloc arbre de W'_0 (et donc de W_k).

(ii) On montre cette propriété par induction sur k :

- Pour $k = 0$, on a $W_0 = \rho_0 = \{Q\}$, et par suite, le seul bloc simple contenu dans W_0 est élément de ρ_0.
- On suppose (ii) vraie pour tout $i < k$, avec k fixé ($k \geq 1$).

  On considère alors la suite $(\rho'_i, W'_i, W''_i)_{(i \in \mathcal{N})}$, de limite $(\rho'_r, \emptyset, W''_r)$, définie par :

  * $(\rho'_0, W'_0, W''_0) = (\rho_{k-1}, W_{k-1}, \emptyset)$,
  * $\forall i \geq 0$, $(\rho'_{i+1}, W'_{i+1}, W''_{i+1}) = \theta''_\Lambda(\rho'_i, W'_i, W''_i, x)$ avec $x \in W'_i$,

  et on a $\rho_k = \rho'_r$ et $W_k = W''_r$.

  Il est facile de voir que, par définition de θ'' :

  * Lorsqu'un partitionneur bloc simple est *inséré* ou *remplacé* dans W''_i (par (2)), alors X est issue de la décomposition d'une classe de ρ_{k-1}. Par suite, X est une classe de ρ_k.
  * Lorsqu'un partitionneur bloc arbre n est *inséré* dans W''_i, alors :
    - soit il représente la décomposition d'une classe de ρ_{k-1} (par (3)), et par conséquent l'union des feuilles de ses fils gauche et droit sont des unions de classes de ρ_k.
    - soit il est issu d'un bloc arbre de W_{k-1} (par (5)) et, d'après l'hypothèse d'induction, l'union des feuilles de ses fils gauche et droit sont des unions de classes de ρ_k.
  * Lorsqu'un partitionneur bloc arbre n est *remplacé* dans W''_i (par (4)), alors, l'union des feuilles de ses fils reste inchangée.

  Par conséquent, la propriété (ii) est encore vérifiée au rang k, et elle donc vraie par induction pour tout entier k positif.

□.

Annexe B

Exemple : un protocole de diffusion atomique

Nous décrivons la modélisation et la vérification d'une partie du service d'un *protocole de diffusion atomique*, le protocole *rel/REL* [SE90], qui a été menée au sein des *Laboratoires Helwett-Packard* (*Bristol, Royaume-Uni*) en collaboration avec Simon Bainbridge. Une description plus complète de ce travail peut être trouvée dans [BM91].

On présente tout d'abord de manière informelle le protocole *rel/REL* ainsi que le service qui en est attendu. On propose alors une modélisation en LOTOS de ce protocole, puis on déduit du programme ainsi obtenu une description sous forme de *processus communicants*, exprimée dans le langage présenté au chapitre 6 (i.e., comportant uniquement des opérateurs de composition parallèle et d'abstraction). Enfin, on termine en décrivant la vérification de l'une des propriétés du service qui peut être menée selon l'approche comportementale en représentant le comportement attendu du protocole par un système de transitions étiquetées.

B.1 Le protocole *rel/REL*

On décrit tout d'abord le principe général du protocole, puis on présente de manière informelle chacun des processus qui le compose. Cette description, inspirée de [SE90], servira de support à la modélisation en LOTOS qui est proposée dans la section suivante.

B.1.1 Le principe du protocole

Le protocole *rel/REL* permet d'effectuer des *échange atomique de messages* dans un réseau entre une station émettrice et un ensemble de stations réceptrices, tout en tolérant les pannes potentielles de chacune des stations engagées dans la communication. Plus précisément, le service de ce protocole peut être résumé par les deux propriétés informelles suivantes :

atomicité :
si la station émettrice E envoie un message m à chaque membre d'un groupe de stations réceptrices noté G, alors, soit m est correctement reçu par *toutes* les stations en fonctionnement de G, soit *aucune* d'entre-elles ne le reçoit, et ce malgré un nombre arbitraires de pannes dans l'ensemble $G \cup \{E\}$.

préservation de l'ordre des messages :
lorsque la station émettrice E envoie une séquence de messages vers un groupe de stations réceptrices G, alors les messages ne peuvent être reçues par les éléments de G que dans l'ordre dans lequel ils ont été émis.

De telles facilités de communication sont nécessaires pour de nombreuses applications, et en particulier chaque fois que des contraintes d'intégrité doivent être préservées entre des données partagées sur différents sites. C'est par exemple le cas pour réaliser une transaction dans une base de données répartie, ou encore pour gérer des objets dupliqués dans un système tolérant les pannes.

Pour implémenter la propriété d'atomicité — et garantir que, même en cas de panne de la station émettrice, le message est reçu soit par l'ensemble de ses destinataires en fonctionnement, soit par aucun d'entre-eux — le protocole *rel/REL* repose sur le principe du *verrouillage transactionnel à deux phase* :

- L'émetteur envoie séquentiellement deux copies identiques du message à transmettre à chacun de ses destinataires. Chacune de ces copies est identifiée de manière unique par un numéro (le numéro du message à laquelle elle est associée) ainsi que par une information indiquant s'il s'agit d'une première ou d'une seconde copie du même message. On suppose également que chaque message contient la liste de ses destinataires.

- Chaque récepteur R qui reçoit une première copie d'un message en attend une seconde copie. Si celle-ci n'est pas reçue au bout d'un certain délai, alors R peut supposer que l'émetteur a cessé de fonctionner, et que, par conséquent, certain des destinataires du message n'ont reçu aucune des deux copies.

  Le récepteur R "relaie" alors l'émetteur en diffusant à son tour deux copies de ce même message vers l'ensemble de ses destinataires, et donc en utilisant à nouveau le protocole *rel/REL*. Toutefois, afin de limiter le nombre total de copies émises, cette rediffusion devra être interrompue si la seconde copie attendue est reçue, soit de la part de l'émetteur, soit de la part d'un autre récepteur déjà en train de le relayer.

Pour mettre en œuvre ce protocole, la transmission point-à-point de messages entre les différentes stations du réseau est réalisée à l'aide d'une couche de niveau *transport*, noté *rel*, qui assure une communication fiable entre n'importe quel couple de stations tout en préservant l'ordre des messages échangés.

Enfin, notons que le protocole *rel/REL* n'est correct que si l'on effectue un certain nombre d'hypothèses sur la nature des pannes qui peuvent affecter les stations du réseau :

- Soit une station fonctionne correctement, soit elle cesse définitivement d'émettre et recevoir des messages (*fail-silent behaviour*).

- Même en cas de pannes d'un nombre quelconque de stations, on suppose qu'il n'y a jamais de partition du réseau : en particulier, deux stations qui fonctionnent peuvent toujours échanger des messages.

B.1.2 Description informelle des composants

La station émettrice est représenté par un processus *Transmitter*, dont le rôle consiste à envoyer les deux copies des messages à transmettre (comme décrit ci-dessus) à l'aide d'une procédure notée REL. Ces messages sont fournis par l'intermédiaire d'une file *message_queue*. Le processus *Transmitter* est décrit par le comportement suivant :

Transmitter :
cycle
si $message\_queue \neq \emptyset$ **alors**
extraire le premier message m de la file $message\_queue$;
REL (m) ; /* diffuser m à ses destinataires */
fsi
fcycle

Chaque message contient la liste de ses destinataires ainsi qu'un champ *type* qui permet d'indiquer s'il s'agit d'une première ou d'une seconde copie du message. La procédure *rel* permet d'envoyer le message m à chacun de ses destinataire en utilisant la couche transport sur laquelle repose le protocole. L'algorithme de la procédure REL est alors :

procédure REL (m)
début
$m.type := first$; $rel\,(m)$; /* envoi de la première copie ... */
$m.type := second$; $rel\,(m)$; /* envoi de la seconde copie ... */
fin

Chaque station réceptrice contient un processus *Receiver* qui est chargé de recevoir les messages provenant du réseau (par l'intermédiaire du protocole de transport). Chaque message m reçu par ce processus sera alors géré par un processus local à la station réceptrice, noté $Thread_m$, et dont le rôle sera de relayer l'émetteur si la seconde copie du message n'a pas été reçue après un certain délai. Le processus *Receiver* est décrit par le comportement suivant :

Receiver :
cycle
réception (m) ; /* réception du message m de la part d'une station du réseau */
si $m.type = first$ **alors**
/* il s'agit d'une première copie du message */
si cette copie de m à déjà été reçue **alors**
détruire (m) /* m doit être ignoré ... */
sinon
délivrer (m) à la station réceptrice ; /* un nouveau message vient d'être reçu */
créer un processus $Thread_m$ associé à m ;
transmettre m au processus $Thread_m$
fsi
sinon
/* il s'agit d'une seconde copie du message */
si cette copie de m à déjà été reçue **alors**
détruire (m) /* m doit être ignoré ... */
sinon
transmettre m au processus $Thread_m$
fsi
fsi
fcycle

Il reste alors à décrire le comportement du processus $Thread_m$:

Thread$_m$:
réception (m) ; /* réception du message m de la part du processus *Receiver* */

```
démarrer un timer ;
/* il reste à attendre la seconde copie du message ou l'expiration du timer */
cycle
    si réception (m) alors
        /* réception de la seconde copie du message m de la part du processus Receiver
           le processus Thread_m est détruit */
        stop
    fsi
    si timeout alors
        /* le timer a expiré
           le message m est réémis et le processus Thread_m est détruit */
        REL (m) ; stop
    fsi
fcycle
```

B.2 Modélisation du protocole en LOTOS

On présente dans un premier temps une description du protocole en considérant des stations émettrices et réceptrices *fiables*. On ajoute alors la modélisation des pannes afin d'obtenir le programme LOTOS complet.

B.2.1 Architecture du protocole

La première tâche consiste à modéliser la couche transport *rel* sur laquelle repose le protocole *rel/REL*. Rappelons que le service fourni par cette couche doit offrir les fonctionnalitées suivantes :

- Il doit permettre une transmission fiable de messages, d'une part de la station émettrice vers n'importe quelle station réceptrice du réseau, et d'autre part entre deux stations réceptrices quelconques (au moment du relais).
- L'ordre des messages doit être préservé.

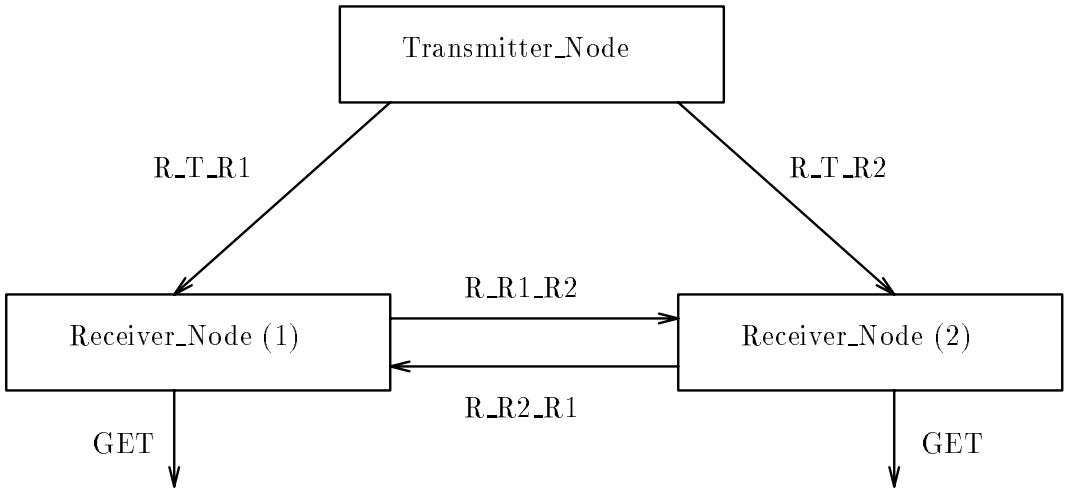
En LOTOS, la solution la plus simple pour implémenter un tel service consiste à utiliser directement le mécanisme du "rendez-vous" : chaque station est représentée par un processus LOTOS et l'échange de messages entre deux stations se fait alors par rendez-vous binaire sur une porte commune à ces deux processus.

On considère dans la suite un réseau composé d'une station émettrice, représentée par le processus LOTOS **Transmitter_Node**, et de deux stations réceptrices, représentées par les processus LOTOS **Receiver_Node (1)** et **Receiver_Node (2)**, mais cette description peut être généralisée à un nombre quelconque de stations réceptrices.

Le tableau suivant donne la nomenclature des portes utilisées pour modéliser la couche *rel* :

Porte	Processus origine	Processus destination
R_T_R1	Transmitter_Node	Receiver_Node (1)
R_T_R2	Transmitter_Node	Receiver_Node (2)
R_R1_R2	Receiver_Node (1)	Receiver_Node (2)
R_R2_R1	Receiver_Node (2)	Receiver_Node (1)

Enfin, on note GET la porte utilisée par les deux stations réceptrices pour transmettre les messages reçus à la couche implantée au-dessus du protocole *rel/REL* (i.e, à son service). La figure suivante décrit alors l'architecture obtenue pour ce réseau :



Remarque B-1

1. Bien que dans le protocole *rel/REL* tous les messages originaires de la station émettrice seront toujours transmis au deux station réceptrices, il n'est pas envisageable de modéliser cette diffusion directement par un rendez-vous n-aire entre ces trois processus, sans quoi la propriété d'atomicité serait trivialement vérifiée.
2. Il aurait été possible d'utiliser une porte unique pour implémenter les communications entre les deux stations réceptrices (le sens de la transmission pouvant alors être codé dans un champ du message). Toutefois, par symétrie avec les messages reçus de la part de la station émettrice, il nous a semblé plus clair d'utiliser deux portes distinctes.

■

Il reste à préciser les types de données nécessaires pour représenter les messages échangées ainsi que les adresses des stations réceptrices :

- Concernant les messages, les seules informations significatives du point de vue du protocole sont le numéro du message et le champ indiquant s'il s'agit d'une première ou d'une seconde copie. En LOTOS, ces données seront respectivement codé par le type MESSAGE, isomorphe au type entier de la librairie standard, et par le type énuméré TYPEMSG, muni d'un opérateur de comparaison :

```
type MESSAGE is NATURAL renamedby
       sortnames MSG for NAT
endtype

type TYPEMSG is BOOLEAN
      sorts
         TYPEMSG
```

```
        opns
           FIRST
           SECOND : -> TYPEMSG
           EQ_TYPE : TYPEMSG, TYPEMSG -> BOOL (* operateur de comparaison *)
        eqns
           forall X, Y : TYPEMSG
           ofsort BOOL
              EQ_TYPE (X, X) = true;
              EQ_TYPE (X, Y) = false; (* en supposant les equations ordonnees *)
  endtype
```

- Les adresses des stations réceptrices seront également codées par le type énuméré ADDRESS :

```
type ADDRESS is
      sorts
         ADR
      opns
            1
            2 : -> ADR
  endtype
```

On obtient alors le fragment de programme LOTOS suivant, pour lequel les processus Transmitter_Node et Receiver_Node sont explicités dans la suite :

```
specification rel_REL_protocol [GET] : noexit behaviour

  hide R_T_R1, R_T_R2 in (

        Transmitter_Node [R_T_R1, R_T_R2]

                |[R_T_R1, R_T_R2]|

        Receiver_Group [R_T_R1, R_T_R2, GET]
  )

where

   process  Receiver_grp [R_T_R1, R_T_R2, GET] : noexit :=

        hide R_R1_R2, R_R2_R1 in (

                Receiver_Node [R_T_R1, R_R1_R2, R_R2_R1, GET] (1 of ADR)

                              |[R_1, R_2]|

                Receiver_Node [R_T_R2, R_R2_R1, R_R1_R2, GET] (2 of ADR)
        )

   endproc
endspec
```

B.2.2 Description de la station émettrice

D'après la description informelle du protocole, la station émettrice se compose d'un processus unique *Transmitter*, chargé de diffuser à l'ensemble des récepteurs deux copies des messages extraits d'une file *message_queue*. Dans la modélisation en LOTOS, cette file peut être remplacée par une constante entière MAX_MSG qui indique le nombre total de messages à émettre : le comportement du processus LOTOS **Transmitter** consiste alors à diffuser séquentiellement deux copies de chaque message numéroté dans l'intervalle [1, .., MAX_MSG].

En représentant la procédure REL par le processus LOTOS **BigRel** on obtient :

```
process Transmitter_Node [R_T_R1, R_T_R2] : noexit :=

   Transmitter [R_T_R1, R_T_R2] (1 of MSG)

where

   process Transmitter [R_T_R1, R_T_R2] (M:MSG) : noexit :=

           [M gt MAX_MSG] -> stop
        []
           [M le MAX_MSG] -> (
                                BigRel [R_T_R1, R_T_R2] (M)
                                    >> Transmitter [R_T_R1, R_T_R2] (M+1)
                             )
   endproc
endproc
```

Enfin, la diffusion d'une copie d'un message consiste à émettre cette copie vers chacune des stations réceptrices, sans préjuger de l'ordre dans laquelle elles doivent la recevoir. En LOTOS, cette diffusion est modélisé par une composition parallèle *asynchrone* (ou *entrelacement*) des rendez-vous sur les portes R_T_R1 et R_T_R2 entre les processus **Transmitter_Node** et **Receiver_Node**.

Le comportement du processus **BigRel** est alors :

```
process BigRel [R_T_R1, R_T_R2] (M:MSG) : exit :=

     rel [R_T_R1, R_T_R2] (M, FIRST) (* diffusion de la premiere copie *)
   >>
     rel [R_T_R1, R_T_R2] (M, SECOND) (* diffusion de la seconde copie *)

where

   process rel [R_T_R1, R_T_R2] (M:MSG, T:TYPEMSG) : exit :=

        ( R_T_R1 !M !T  ; exit ) (* transmission du message au Receiver_Node (1) *)
                |||
        ( R_T_R2 !M !T  ; exit ) (* transmission du message au Receiver_Node (2) *)

   endproc
endproc
```

B.2.3 Description des stations réceptrices

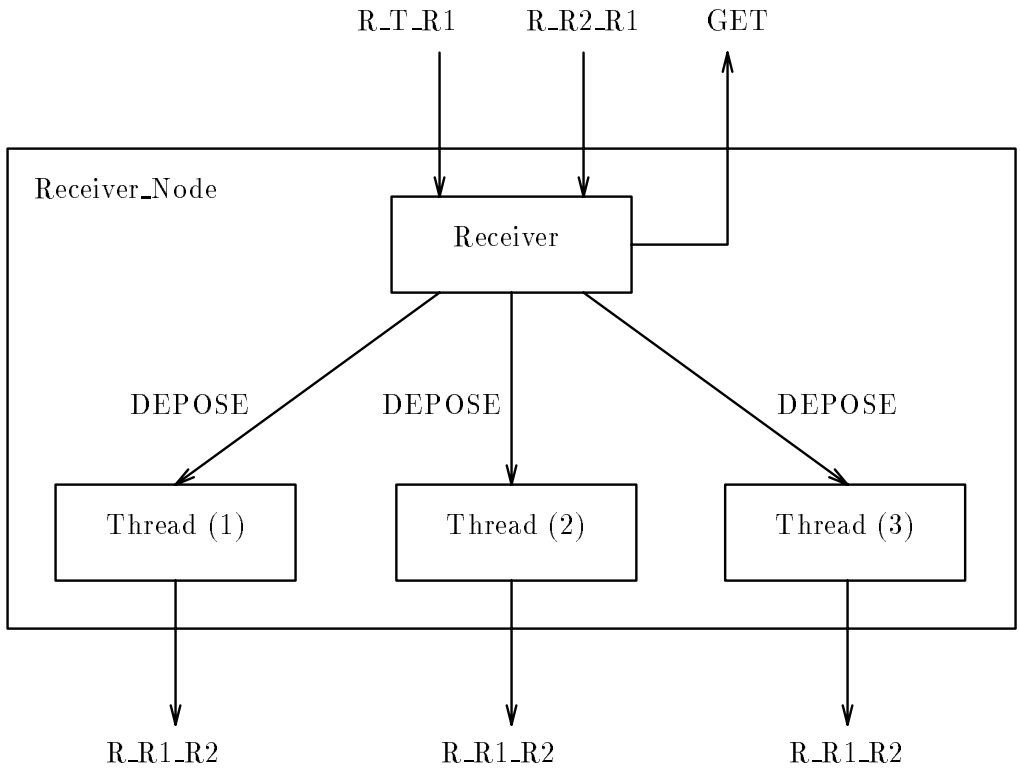
Dans la description informelle, une station réceptrice est constitué d'un processus *Receiver*, qui reçoit les messages du réseau et les délivre le cas échéant à la couche supérieure, et des processus $Thread_m$, qui sont crées en fonction des messages reçus. Cette décomposition sera conservée pour modéliser le processus LOTOS **Receiver_Node**.

Toutefois, la création dynamique de processus n'étant pas possible en LOTOS, il est nécessaire que les processus **Thread** soient mis en place statiquement, en fonction du nombre de messages qui pourront être adressés à la station réceptrice que l'on considère. Dans notre cas, ces messages appartiennent à l'intervalle [1, .., MAX_MSG].

Il reste alors à préciser comment ces différents processus se synchronisent :

- Le processus **Receiver** communique avec chacun des processus **Thread**, par rendez-vous sur une porte notée DEPOSE.
- Les processus **Thread** ne communiquent pas entre-eux, ce qui correspond à une composition parallèle asynchrone.

A titre d'exemple, pour MAX_MSG $= 3$, l'architecture du processus **Receiver_Node (1)** est représentée par la figure suivante :



Plus généralement, cette figure correspond au fragment de programme LOTOS suivant, pour lequel il

reste à expliciter les processus **Receiver** et **Thread** :

```
process Receiver_Node [RT, RE, RR, GET] (A:ADR) : noexit :=

   hide DEPOSE in (

        Receiver [RT, RR, DEPOSE, GET] (A)

                        |[DEPOSE]|
        (
                Thread [DEPOSE, RE] (1 of MSG)
                           |||
                Thread [DEPOSE, RE] (2 of MSG)
                           |||
                            .
                            .
                            .
                           |||
                Thread [DEPOSE, RE] (MAX_MSG)
        )
   )

endproc
```

D'après sa description informelle, le comportement du processus **Receiver** nécessite de savoir détecter si une première ou deuxième copie d'un message donné a déjà été reçue. On associe donc à ce processus deux tables (une pour chaque copie) qui permettront de mémoriser les numéros des messages déjà reçus. Plus formellement, ces structures sont définies par le type TABLE suivant, qui est isomorphe au type "ensemble" usuel :

```
type TABLE is BOOLEAN, MESSAGE
   sorts
        TAB
   opns
        Reset : -> TAB          (* la table vide *)
        Set : TAB, MSG -> TAB   (* ajout de message dans la table *)
        Test : TAB, MSG -> BOOL (* test de la presence d'un message *)
   eqns
      forall T:TAB, M, N:MSG
      ofsort BOOL
         Test (Reset, N) = false;
         Test (Set (T, M), N) = (M eq N) or Test (T, N);
endtype
```

Le processus **Receiver** se déduit alors directement de sa définition informelle :

```
process Receiver [RT, RR, DEPOSE, GET] (A:ADR, TF:TAB, TS:TAB) : noexit :=

 choice REL_R in [RT, RR] []
    (
        REL_R ? M:MSG ? T:TYPEMSG ; (* reception d'un message du reseau *)
        (
          [EQ_TYPE(T, FIRST) and not Test (TF,M)] ->
```

```
            GET !A !M ;      (* delivrer le message a la couche superieure *)
               DEPOSE !M ;   (* transmettre M a sa thread *)
                 Receiver[RT, RR, DEPOSE, GET] (A, SET(TF,M), TS)
        []
          [EQ_TYPE(T, FIRST) and Test (TF,M)]  ->  (* copie deja recue *)
            Receiver[RT, RR, DEPOSE, GET] (A, TF, TS)
        []
          [EQ_TYPE(T, SECOND) and not Test(TS,M)] ->
            DEPOSE !M ;      (* transmettre M a sa thread *)
                 Receiver[RT, RR, DEPOSE, GET] (A, TF, SET(TS,M))
        []
          [EQ_TYPE(T, SECOND) and Test(TS,M)] ->  (* copie deja recue *)
            Receiver[RT, RR, DEPOSE, GET] (A, TF, TS)
        )
   )
endproc
```

Remarque B-2
Dans le processus **Receiver_Node**, les deux tables sont initialisées avec l'ensemble vide :

```
Receiver [RT, RR, DEPOSE, GET] (A, Reset, Reset)
```

■

Il reste alors à modéliser les processus **Thread**, en fonction de leur description informelle :

- L'interruption du relais sur réception de la seconde copie du message est modélisée à l'aide de l'opérateur *disable* "[>" de LOTOS.
- Le timer n'a pas à être représenté explicitement. En effet, puisque les actions de réception des copies du message (sur la porte DEPOSE) et de retransmission (sur les portes R_R1_R2 ou R_R2_R1) sont asynchrone, les trois cas de figure possibles sont bien modélisés :
  - la réception de la seconde copie a lieu *avant* le début du relais,
  - la réception de la seconde copie a lieu *pendant* le relais, entre deux actions atomiques quelconques,
  - la réception de seconde copie a lieu *après* la fin du relais.
- Enfin, le relais lui-même est modélisé par un processus **BigRel_Relay** similaire au processus **BigRel** utilisé dans le processus **Transmitter_Node**, à la différence que lors d'un relais les messages ne sont pas adressés au récepteur qui est à l'origine de ce relais.

On obtient alors le comportement LOTOS suivant :

```
process Thread [DEPOSE, RR] (M : MSG) : noexit :=

   DEPOSE !M ; (* reception de la premiere copie d'un message *)
        (
             ( BigRel_Relay [RR] (M) >> stop ) (* relais *)
           [>
             ( DEPOSE !M ; stop )
                   (* interrompu par la reception de la seconde copie *)
        )

where
```

```
   process BigRel_Relay [R] (M : MSG) : exit :=

          rel_relay [R] (M, FIRST)  (* diffusion de la premiere copie *)
        >>
          rel_relay [R] (M, SECOND) (* diffusion de la seconde copie *)

   where

        process rel_relay [R] (M:MSG, T:TYPEMSG) : exit :=

                ( R !M !T  ; exit )  (* emission vers l'autre recepteur *)

        endproc
   endproc
endproc
```

B.2.4 Modélisation des pannes

A partir du programme LOTOS obtenu dans la section précédente, il reste à modéliser les pannes qui peuvent affecter chacune des stations du réseau afin d'obtenir une description complète du protocole *rel/REL*.

Pour un processus LOTOS quelconque **P**, on s'intéresse donc à la définition d'un processus **Fail_P**, dont le comportement soit semblable à celui de **P**, mais qui puisse également *à tout instant* cesser de fonctionner. Compte-tenu des hypothèses retenues dans la description informelle du protocole sur la nature des pannes, et en particulier le *fail-silent behaviour*, une solution immédiate consiste à procéder de la manière suivante :

- Lorsque **Fail_P** cesse de fonctionner, il effectue une action particulière notée CRASH.
- Pour chaque action atomique du processus **P**, **Fail_P** peut soit choisir d'évoluer normalement en effectuant la même action, soit choisir de cesser de fonctionner en effectuant l'action CRASH et en se bloquant (c'est à dire en n'effectuant plus aucune action).

En LOTOS le processus **Fail_P** se déduit donc directement de la définition de **P** à l'aide de l'opérateur *disable* "[>" :

```
process Fail_P [...] : noexit :=

        (P [> (CRASH ; stop))

endproc
```

Néanmoins, si cette première solution permet bien de modéliser les pannes d'un processus **P** *isolé*, elle ne peut pas être directement appliquée aux processus **Transmitter_Node** et **Receiver_Node** décrits dans la section précédente, qui sont supposés effectuer des rendez-vous avec leur *environnement* (i.e., les autres stations du réseau). Plus précisément, cette solution présente dans ce cas deux inconvénients :

1. Elle ne permet pas de distinguer un processus *en panne* d'un processus *bloqué* (i.e., qui attend indéfiniment un rendez-vous avec un autre processus). En effet, la sémantique de l'opérateur "[>" implique que, si **P** se bloque (et devient ainsi équivalent au processus **stop**), alors **Fail_P**

effectue *nécessairement* l'action CRASH.

Cette situation n'est pas sans conséquence sur la vérification. Par exemple, la propriété d'atomicité du service de ce protocole ne concerne que les stations réceptrices "en fonctionnement". Si, par la suite d'un blocage possible dans le protocole, une station réceptrice cesse "artificiellement" de fonctionner alors elle ne sera pas prise en compte lors de la vérification de cette propriété, et l'anomalie ne sera donc pas détectée.

2. Un station en panne peut empêcher deux stations en fonctionnement de communiquer. Cette situation, contraire aux hypothèses retenues sur la nature des pannes, et due au choix que nous avons fait de modéliser la couche *rel* par du rendez-vous. En effet, si le processus **Transmitter_Node**, lors de la diffusion d'une copie du message, tente de se synchroniser par rendez-vous avec un processus **Receiver_Node** en panne (i.e., qui a effectué une action CRASH), alors ce dernier ne pourra jamais accepter le rendez-vous et le processus **Transmitter_Node** sera donc bloqué à son tour, ce qui interrompt sa diffusion.

En conséquence, il est nécessaire d'affiner cette première définition du processus **Fail_P** afin obtenir une modélisation des pannes qui soit mieux adaptée à la situation que l'on souhaite décrire.

Pour distinguer un processus en panne d'un processus bloqué, la solution que nous avons retenue consiste à exprimer différemment le fait que **Fail_P** puisse cesser de fonctionner *à tout instant*. Dans ce nouveau modèle, pour chaque action atomique A du processus **P**, **Fail_P** peut choisir de manière non déterministe entre :

- effectuer l'action A, puis *nécessairement* effectuer l'action CRASH et cesser de fonctionner (en n'effectuant plus aucune action).
- effectuer l'action A, puis ne pas avoir la possibilité d'effectuer l'action CRASH avant d'avoir effectué une nouvelle action.

En LOTOS, ce choix non déterministe peut être facilement exprimé en utilisant la technique de programmation *par contraintes* (ou *constraint-oriented*), qui consiste à définir **Fail_P** comme la composition parallèle de **P** et d'un processus noté **Crash_P**, et dont le rôle est de modéliser les pannes de **P**. Dans le cas du processus **Transmitter_Node**, on obtient alors la nouvelle définition suivante :

```
hide CRASH_T in (
        Transmitter [R_T_R1, R_T_R2] (1 of MSG)

                |[R_T_R1, R_T_R2]|

        Crash_Transmitter [R_T_R1, R_T_R2]
   )

   where

   process Crash_Transmitter [R_T_R1, R_T_R2, CRASH_T] : noexit :=

      (R_T_R1 ? M:MSG ? T:TYPEMSG ; Crash_Transmitter [R_T_R1, R_T_R2, CRASH_T])
     []
      (R_T_R2 ? M:MSG ? T:TYPEMSG ; Crash_Transmitter [R_T_R1, R_T_R2, CRASH_T])
     []
      (R_T_R1 ? M:MSG ? T:TYPEMSG ; CRASH_T ; stop)
     []
```

```
      (R_T_R2 ? M:MSG ? T:TYPEMSG ; CRASH_T ; stop)

  endproc
```

Pour résoudre le second problème (i.e., le fait qu'un récepteur en panne puisse empêcher un émetteur de communiquer avec des récepteurs en fonctionnement), il est également nécessaire d'affiner la modélisation du *fail-silent behaviour*. Dans le cas du processus **Receiver_Node**, une panne ne doit donc pas empêcher le processus **Transmitter_Node** ou un autre processus **Receiver_Node** de se synchroniser par rendez-vous avec ce processus, ce qui implique d'accepter tout rendez-vous sur une porte de *réception*.

On obtient alors la spécification LOTOS suivante pour le processus **Fail_Receiver_Node**, toujours dans le cas où trois messages sont émis :

```
   hide DEPOSE in (
        (
            (
                Receiver [RT, RR, DEPOSE, GET] (A)

                           |[DEPOSE]|
                (
                     Thread [DEPOSE, RE] (1 of MSG)
                                |||
                     Thread [DEPOSE, RE] (2 of MSG)
                                |||
                     Thread [DEPOSE, RE] (3 of MSG)
                )
            )
                [> ( CRASH_R !A ; Ear [RT, RR] )
        )

          |[RT, RE, RR, GET, DEPOSE, CRASH_R]|

        Crash_Receiver [RT, RE, RR, GET, DEPOSE, CRASH_R] (A)
   )
```

avec,

```
process Crash_Receiver [RT, RE, RR, GET, DEPOSE, CRASH_R] (A:ADR) : noexit :=

     (RT ? M:MSG ? T:TYPEMSG ; Crash_Receiver [RT, RE, RR, GET, DEPOSE, CRASH_R] (A))
   []
     (RE ? M:MSG ? T:TYPEMSG ; Crash_Receiver [RT, RE, RR, GET, DEPOSE, CRASH_R] (A))
   []
     (RR ? M:MSG ? T:TYPEMSG ; Crash_Receiver [RT, RE, RR, GET, DEPOSE, CRASH_R] (A))
   []
     (GET ? A:ADR ? M:MSG ; Crash_Receiver [RT, RE, RR, GET, DEPOSE, CRASH_R] (A))
   []
     (DEPOSE ? M:MSG ; Crash_Receiver [RT, RE, RR, GET, DEPOSE, CRASH_R] (A))
   []
     (RT ? M:MSG ? T:TYPEMSG ; CRASH_R !A ; Ear [RT, RR])
   []
     (RE ? M:MSG ? T:TYPEMSG ; CRASH_R !A ; Ear [RT, RR])
```

```
   []
     (RR ? M:MSG ? T:TYPEMSG ; CRASH_R !A ; Ear [RT, RR])
   []
     (GET ? A:ADR ? M:MSG ; CRASH_R !A ; Ear [RT, RR])
   []
     (DEPOSE ? M:MSG ; CRASH_R !A ; Ear [RT, RR])

endproc

process Ear [RT, RR] : noexit :=

     (RT ? M:MSG ? T:TYPEMSG ; Ear [RT, RR])
   []
     (RR ? M:MSG ? T:TYPEMSG ; Ear [RT, RR])

endproc
```

B.3 Modélisation sous forme de processus communicants

A partir du programme LOTOS obtenu dans la section précédente, il est possible de déduire une description du protocole sous la forme d'une expression de composition entre systèmes de transitions étiquetées, exprimée dans le langage présenté au chapitre 6, et qui autorise une vérification "à la volée" à l'aide du logiciel ALDÉBARAN.

Plus précisément, l'objectif consiste à transformer le programme LOTOS initial en un programme ne comportant plus que des définitions de processus dits *élémentaires*, composées à l'aide de l'opérateur de composition parallèle ("`|[ ... ]|`") et de l'opérateur d'abstraction ("`hide ... in ...`"). En général, plusieurs approches sont envisageables, selon le nombre de processus élémentaires que l'on souhaite obtenir.

La stratégie que nous avons retenue pour traiter cet exemple consiste à maximiser cette valeur en choisissant comme processus élémentaires les processus du programme LOTOS initial qui comportent d'autres opérateurs que la composition parallèle et l'abstraction. On obtient alors l'expression de composition suivante, qui reflète l'architecture du protocole :

```
hide R_T_R1, R_T_R2 in
  (
      (
        hide CRASH_T in
          (
            Transmitter
             |[R_T_R1, R_T_R2]|
            Crash_Transmitter
          )
      )
     |[R_T_R1, R_T_R2]|
      (
        hide R_R1_R2, R_R2_R1 in
          (
             (
```

```
              hide DEPOSE, CRASH_R in
               (
                 Receiver_1
                  |[R_T_R1, R_R1_R2, R_R2_R1, GET, DEPOSE]|
                 Crash_Receiver_1
               )
            )
           |[R_R1_R2, R_R2_R1 ]|
            (
              hide DEPOSE, CRASH_R in
               (
                 Receiver_2
                  |[R_T_R2, R_R1_R2, R_R2_R1, GET, DEPOSE]|
                 Crash_Receiver_2
               )
            )
         )
      )
  )
```

Remarque B-3
Dans l'implémentation que nous avons réalisée dans ALDÉBARAN le passage de valeurs n'est pas autorisé lors du rendez-vous, et il est donc nécessaire de le remplacer par de la synchronisation "pure". Pour ce faire, on substitue dans l'expression de composition chaque occurrence des portes utilisées dans un processus élémentaires pour recevoir une valeur d'un type donné par la liste des portes correspondant à l'émission de toutes les valeurs définis dans ce type.

Par exemple, s'il existe dans un processus élémentaire une occurrence d'une porte G de la forme `G ? X:S` alors G sera remplacée dans l'expression de composition par la liste

```
G !s1, G !s2, ..., G !sn
```

où s1, s2, ..., sn sont les éléments du type S. ■

B.4 Vérification

On s'intéresse à la vérification de l'une des deux propriétés énoncées dans la description informelle du service du protocole *rel/REL*, qui concerne la *préservation de l'ordre des messages*. En effet, le fait que, par hypothèse, la couche transport *rel* préserve cet ordre ne permet pas directement de déduire cette propriété pour l'ensemble du protocole. En particulier, rien ne garantit à priori que le mécanisme du "relais" introduit par les processus "threads" ne risque pas d'inverser les réceptions de deux messages pour certaines stations réceptrices.

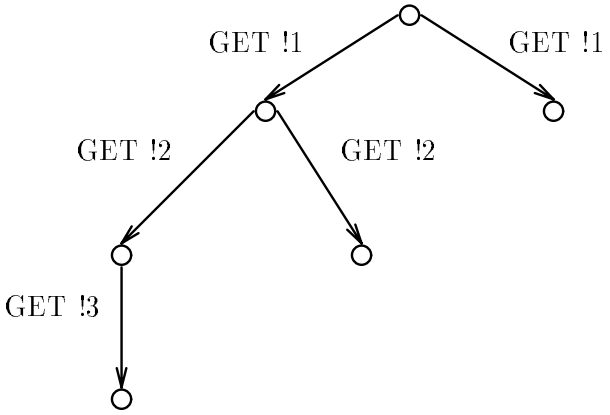
B.4.1 Description formelle de la propriété à vérifier

Il s'agit d'une *propriété de sûreté* : elle indique que, si les messages sont reçus par une station, alors ils le sont selon un ordre donné, mais elle n'impose pas que la réception ait lieu *inévitablement*. En fait, il existe même de nombreux scénarios valides pour lesquels des réceptions de message n'ont pas lieu (par exemple suite à une panne, soit de la station émettrice avant qu'elle n'entame une diffusion, soit de la station réceptrice que l'on considère).

Du point de vue théorique, cette propriété est donc préservée par n'importe quelle relation d'équivalence incluse dans l'équivalence de sûreté, mais la spécification comportementale qui lui est associée dépend néanmoins de la relation que l'on choisit pour la vérification. Plus précisément, on donne le comportement attendu du protocole lorsque l'on considère comme visible les seules actions de réception effectués par l'une des stations réceptrices (action GET dans le programme LOTOS), et si l'on suppose que trois messages (numérotés 1, 2 et 3) sont diffusés en séquence par la station émettrice :

- Vis-à-vis de l'*équivalence de sûreté*, ce comportement est représenté par le système de transitions étiquetées (très simple) de la figure suivante :

- Si l'on considère la τ^*a-*bisimulation*, il est nécessaire de spécifier également le fait que certains des messages peuvent ne pas être reçus à la suite d'une panne de l'émetteur ou du récepteur que l'on observe. Le comportement attendu du protocole est donc représenté par le système de transitions étiquetées de la figure suivante :



- Enfin, si l'on considère une relation plus forte (comme l'*équivalence observationnelle* ou la *bisimulation de branchement*), la présence des pannes ne permet pas de donner une description intuitive de ce comportement. En particulier, aucun des systèmes de transitions étiquetées des deux figures précédentes n'est une spécification comportementale correcte du protocole vis-à-vis de ces deux relations.

B.4.2 Vérification à partir des systèmes de transitions étiquetées

Nous avons tout d'abord effectué une première vérification de cette propriété en générant à l'aide du compilateur CÆSAR le système de transitions étiquetées associé au programme LOTOS décrit dans la section B.2. Pour un réseau constitué d'une station émettrice et deux stations réceptrices, la taille de ce système était de l'ordre de 600 000 états et de 1 900 000 transitions.

En raison de sa taille, la comparaison de ce système de transitions étiquetées avec celui de la spécification comportementale n'a pu être effectué qu'en utilisant l'algorithme de vérification "à la volée" (les algorithmes classiques provoquant des débordements de capacité mémoire). Dans le cas de

l'équivalence de sûreté et dans le cas de la $\tau^* a$-bisimulation, le temps de la comparaison est alors de l'ordre de quelques heures sur une station de travail de type SUN 3.

Remarque B-4
Notons que pour la $\tau^* a$-bisimulation la spécification comportementale n'est pas déterministe, et que c'est donc l'algorithme "général" (algorithme 5-3) qui est mis en œuvre. ■

B.4.3 Vérification "à la volée"

Nous avons par la suite effectué une seconde vérification, sans générer au préalable le système de transitions étiquetées correspondant au programme LOTOS complet, mais directement "à la volée", à partir de la description sous forme de processus communicants présentée dans la section B.3. Le principe consiste alors :

1. à *générer* à l'aide du compilateur CÆSAR les systèmes de transitions étiquetées correspondants à chacun des processus élémentaires,
2. à *réduire* ces systèmes à l'aide d'ALDÉBARAN, modulo une relation d'équivalence R, qui doit préserver la relation utilisée lors de la vérification,
3. et enfin à *comparer*, toujours à l'aide d'ALDÉBARAN, la spécification comportementale avec le système obtenu en composant les systèmes réduits, et ce au fur et à mesure de sa génération.

Pour une configuration du réseau identique à celle de la section précédente, nous avons obtenu les résultats suivants :

- Lorsque les systèmes de transitions étiquetées correspondants aux processus élémentaires sont réduits modulo la *bisimulation forte* la comparaison s'effectue en quelques dizaines de minutes.
- Lorsque les systèmes de transitions étiquetées correspondants aux processus élémentaires sont réduits modulo la $\tau^* a$*-bisimulation* la comparaison s'effectue en quelques minutes.

B.4.4 Discussion

Nous terminons par quelques remarques plus générales sur la vérification de ce protocole :

- La propriété d'*atomicité* est plus difficile à exprimer à l'aide d'une spécification comportementale. En fait, il s'est avéré que cette propriété était plus facile à caractériser à l'aide d'une formule de logique temporelle. On trouve une expression de cette propriété en logique LTAC dans [FGM+91], et sa vérification complète est décrite dans [BM91].
- La vérification de ce protocole à permis d'améliorer la connaissance de son comportement, et en particulier sous certaines hypothèses de borner la taille de la structure de donnée utilisée dans le processus *Receiver* pour détecter les messages dupliqués. Par suite, il a été possible d'obtenir une nouvelle version de ce processus dans laquelle il n'y a plus de création dynamique d'un processus *Thread* pour chaque message reçu (le nombre de *Thread* est fixé en fonction du nombre total de stations du réseau).

Enfin, notons que la vérification d'un protocole de même nature spécifié en ESTELLE est décrite dans [BGR+90].

Bibliographie

[AHU74] A. Aho, J. Hopcroft, and J. Ullman. *Design and analysis of computer Algorithms.* Addison Wesley, 1974.

[AS87] B. Alpern and F. B. Schneider. Recognizing Safety and Liveness. *Distributed Computing*, 2(2–3):117–126, May 1987.

[BB88] T. Bolognesi and E. Brinksma. Introduction to the ISO Specification Language LOTOS. *Computer Networks and ISDN Systems*, 14(1):25–29, January 1988.

[BCM+90] J.R. Burch, E.M. Clarke, K.L. McMillan, D.L. Dill, and L.J. Hwang. Symbolic Model Checking : 10^{20} states and beyond. In *Proceedings 5th Annual Symposium on Logic in Computer Science (LICS 90), Philadelphia USA*, pages 118–129, Los Alamitos, CA, June 1990. IEEE Computer Society Press.

[BFG+91] A. Bouajjani, J.C. Fernandez, S. Graf, C. Rodríguez, and J. Sifakis. *Safety for Branching Time Semantics.* In *Proceedings of the 18th ICALP, Madrid, Spain*, volume 510 of *Lecture Notes in Computer Science.* Springer Verlag, Berlin, July 1991.

[BFH90a] A. Bouajjani, J.C. Fernandez, and N. Halbwachs. Minimal Model Generation. In R.P. Kurshan and E.M. Clarke, editors, *Proceedings of the Workshop on Computer-Aided Verification (Rutgers, USA).* DIMACS, June 1990.

[BFH90b] A. Bouajjani, J.C. Fernandez, and N. Halbwachs. On the Verification of Safety Properties. Rapport technique SPECTRE L12, Laboratoire de Génie Informatique — Institut IMAG, Grenoble, March 1990.

[BGR+90] M. Baptista, S. Graf, J.-L. Richier, L. Rodrigues, C. Rodríguez, P. Verissimo, and J. Voiron. Formal Specification and Verification of a Network Independent Atomic Multicast Protocol. In Juan Quemada, José Manas, and Enrique Vázquez, editors, *Proceedings of the 3rd International Conference on Formal Description Techniques FORTE'90 (Madrid, Spain)*, Amsterdam, November 1990. North-Holland.

[BHR84] S. D. Brookes, C.A.R Hoare, and A.W. Roscoe. Theory of Communicating Sequential Processes. *JACM*, 31(3), 1984.

[BKO88] J.A. Bergstra, J.W. Klop, and E.-R. Oldeborg. Readies and failures in the algebra of communicating processes. *SIAM Journal of Computing*, 17(6):1134–1177, 1988.

[BM91] Simon Bainbridge and Laurent Mounier. Specification and Verification of a Reliable Multicast Protocol. Software Engineering Department Technical Report HPL-91-63, Hewlett-Packard Laboratories, Bristol, U.K., October 1991.

[Bou85a] G. Boudol. Calculs de Processus et Vérification. Technical Report RR424, INRIA, Sophia-Antipolis, 1985.

[Bou85b] G. Boudol. Le calcul Meije. In J.P Verjus et G. Roucairo, editor, *Parallélisme, Communication, Synchronisation.* CNRS, 1985.

[Bry86] R. E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computers*, C-35(8), 1986.

[BvG87] J.C.M. Baeten and R.J. van Glabbeek. *Another look at abstraction in process algebra.* In Th. Ottman, editor, *Proceedings ICALP 87 (Karlsruhe)*, volume 267 of *Lecture Notes in Computer Science*, pages 84–94. Springer Verlag, Berlin, 1987.

[CES83] E. Clarke, E.A. Emerson, and A.P. Sistla. Automatic Verification of Finite State Concurrent Systems Using Temporal Logic. In *10th. Annual Symp. on Principles of Programming Languages*, 1983.

[CG87] E. Clarke and O. Grumberg. Avoiding the State Explosion Problem in Temporal Logic Model Checking Algorithm. In *6th. ACM SIGACT-SIGOPTS Symp. on Principles of Distributed Computing*, 1987.

[CH89] R. Cleaveland and M. Hennessy. *Testing Equivalence as a Bisimulation Equivalence.* In Joseph Sifakis, editor, *Proceedings of the Workshop on Automatic Verification Methods for Finite State Systems (Grenoble, France)*, volume 407 of *Lecture Notes in Computer Science.* Springer Verlag, Berlin, June 1989.

[Cle90] Rance Cleaveland. *On Automatically Distinguishing Inequivalent Processes.* In R.P. Kurshan and E.M. Clarke, editors, *Proceedings of the Workshop on Computer-Aided Verification (Rutgers, USA).* June 1990.

[CLM89] E.M. Clarke, D.E. Long, and K.L. McMillan. Compositional Model Checking. In *Proceedings 4th Annual Symposium on Logic in Computer Science (LICS 89), Asilomar (CA) USA*, Los Alamitos, CA, June 1989. IEEE Computer Society Press.

[CMB90] O. Coudert, J.C. Madre, and C. Berthet. Verifying Temporal Properties of Sequential Machine Without Building their State Diagram. In R.P. Kurshan and E.M. Clarke, editors, *Proceedings of the Workshop on Computer-Aided Verification (Rutgers, USA).* DIMACS, June 1990.

[CPS89] R. Cleaveland, J. Parrow, and B. Steffen. *The Concurrency Workbench.* In Joseph Sifakis, editor, *Proceedings of the Workshop on Automatic Verification Methods for Finite State Systems (Grenoble, France)*, volume 407 of *Lecture Notes in Computer Science.* Springer Verlag, Berlin, June 1989.

[CS91] R. Cleaveland and B. Steffen. *Computing Behavioural Relations, Logically.* In *18th ICALP*, volume 510 of *Lecture Notes in Computer Science*, pages 12,12. Springer Verlag, Berlin, July 1991.

[CVWY90] C. Courcoubetis, M. Vardi, P. Wolper, and M. Yannakakis. Memory Efficient Algorithms for the Verification of Temporal Properties. In R.P. Kurshan and E.M. Clarke, editors, *Proceedings of the Workshop on Computer-Aided Verification (Rutgers, USA).* DIMACS, June 1990.

[CW87] D. Coppersmith and S. Winograd. Matrix multiplication via arithmic progressions. In *Proceedings 19th ACM Symposium on Theory of Computing*, pages 1–6, New York City, NY, 1987.

[dSV89] R. de Simone and D. Vergamini. Aboard AUTO. Technical Report 111, INRIA, 1989.

[EåFJ90] Patrik Ernberg, Lars åke Fredlung, and Bengt Jonsson. Specification and Validation of a Simple Overtaking Protocol using LOTOS. T 90006, Swedish Institute of Computer Science, Kista, Sweden, October 1990.

[EFT91] R. Enders, T. Filkorn, and D. Taubner. Generating BDDs for Symbolic Model Checking in CCS. In K. G. Larsen, editor, *Proceedings of the 3rd Workshop on Computer-Aided Verification (Aalborg, Denmark)*, July 1991.

[Fer88] J.C. Fernandez. *ALDEBARAN : un système de vérification par réduction de processus communicants.* Thèse de doctorat, Université Joseph Fourier (Grenoble), May 1988.

[Fer89] J.C. Fernandez. ALDEBARAN : A Tool for Verification of Communicating Processes. Rapport technique SPECTRE C14, Laboratoire de Génie Informatique — Institut IMAG, Grenoble, September 1989.

[Fer90] J.C. Fernandez. An Implementation of an Efficient Algorithm for Bisimulation Equivalence. *Science of Computer Programming*, 13(2–3):219–236, May 1990.

[FGM+91] Jean-Claude Fernandez, Hubert Garavel, Laurent Mounier, Anne Rasse, Carlos Rodríguez, and Joseph Sifakis. Une boîte à outils pour la vérification de programmes LOTOS. In Omar Rafiq, editor, *Actes du Colloque Francophone pour l'Ingénierie des Protocoles CFIP'91 (Pau, France)*, Paris, September 1991. Hermès.

[FM90] Jean-Claude Fernandez and Laurent Mounier. Verifying Bisimulations "On the Fly". In Juan Quemada, José Manas, and Enrique Vázquez, editors, *Proceedings of the 3rd International Conference on Formal Description Techniques FORTE'90 (Madrid, Spain)*, Amsterdam, November 1990. North-Holland.

[FM91a] Jean-Claude Fernandez and Laurent Mounier. "On the Fly" Verification of Behavioural Equivalences and Preorders. In K. G. Larsen, editor, *Proceedings of the 3rd Workshop on Computer-Aided Verification (Aalborg, Denmark)*, volume 575 of *Lecture Notes in Computer Science*, Berlin, July 1991. Springer Verlag.

[FM91b] Jean-Claude Fernandez and Laurent Mounier. A Tool Set for Deciding Behavioral Equivalences. In J.C.M Baeten and J.F. Groote, editors, *Proceedings of the 2nd International Conference on Concurrency Theory (CONCUR'91)*, pages 23–42, Amsterdam, August 1991. North-Holland.

[Gar89] Hubert Garavel. *Compilation et vérification de programmes LOTOS.* Thèse de Doctorat, Université Joseph Fourier (Grenoble), November 1989.

[Gar90] H. Garavel. Spécification de la simulation : I. Décomposition en modules. Document VESAR 017, VERILOG, Grenoble, March 1990.

[Gla90] R.J. van Glabbeek. The Linear Time - Branching Time Spectrum. CS R9029, Centrum voor Wiskunde en Informatica, Amsterdam, 1990.

[GLZ89] J.C Godskesen, K.G. Larsen, and M. Zeeberg. TAV Users Manual. Internal report, Aalborg University Center, Denmark, 1989.

[God90] Patrice Godefroid. *Using Partial Orders to Improve Automatic Verification Methods.* In R.P. Kurshan and E.M. Clarke, editors, *Proceedings of the Workshop on Computer-Aided Verification (Rutgers, USA).* June 1990.

[GRRV89] Suzanne Graf, Jean-Luc Richier, Carlos Rodríguez, and Jacques Voiron. *What are the Limits of Model Checking Methods for the Verification of Real Life Protocols?* In Joseph Sifakis, editor, *Proceedings of the Workshop on Automatic Verification Methods for Finite State Systems (Grenoble, France)*, volume 407 of *Lecture Notes in Computer Science*, pages 275–285. Springer Verlag, Berlin, June 1989.

[GS86] Suzanne Graf and Joseph Sifakis. Readiness Semantics for Processes with Silent Actions. Rapport technique SPECTRE C3, Laboratoire de Génie Informatique — Institut IMAG, Grenoble, November 1986.

[GS90a] Hubert Garavel and Joseph Sifakis. Compilation and Verification of LOTOS Specifications. In L. Logrippo, R. L. Probert, and H. Ural, editors, *Proceedings of the 10th International Symposium on Protocol Specification, Testing and Verification (Ottawa, Canada)*, Amsterdam, June 1990. IFIP, North-Holland.

[GS90b] Suzanne Graf and Bernhard Steffen. Compositional Minimization of Finite State Processes. In *Proceedings of the Workshop on Computer-Aided Verification (Rutgers, USA).* DIMACS, R.P. Kurshan and E.M. Clarke, June 1990.

[GV90] J.F. Groote and F. Vaandrager. An Efficient Algorithm for Branching Bisimulation and Stuttering Equivalence. CS R9001, Centrum voor Wiskunde en Informatica, Amsterdam, January 1990.

[GW89] R.J van Glabbeek and W. P. Weijland. Branching-Time and Abstraction in Bisimulation Semantics (extended abstract). CS R8911, Centrum voor Wiskunde en Informatica, Amsterdam, 1989. also in proc. IFIP 11th World Computer Congress, San Francisco, 1989.

[GW91a] P. Godefroid and P. Wolper. A Partial Approach to Model Checking. In *Proceedings 6th Annual Symposium on Logic in Computer Science (LICS 91), Amsterdam.* IEEE Computer Society Press, July 1991.

[GW91b] P. Godefroid and P. Wolper. Using Partial Orders for the Efficient Verification of Deadlock Freedom and Safety Properties. In K. G. Larsen, editor, *Proceedings of the 3rd Workshop on Computer-Aided Verification (Aalborg, Denmark)*, July 1991.

[HM85] M. Hennessy and R. Milner. Algebraic Laws for Nondeterminism and Concurrency. *Journal of the Association for Computing Machinery*, 34(1):137–161, january 1985.

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

[Hol85] G.J. Holzmann. Tracing protocols. *ATT Technical Journal*, 64(10):2413–2434, october 1985.

[Hol87] G.J. Holzmann. Automated Protocol Validation in *Argos* : Assertion Proving and Scatter Searching. *IEEE Transaction on Software Engineering*, SE-13(6):683–696, June 1987.

[Hol89] G.J. Holzmann. Algorithms for Automated Protocol Validation. In Joseph Sifakis, editor, *Proceedings of the 1st International Workshop on Automatic Verification Methods for Finite State Systems (Grenoble, France)*, June 1989.

[IP91] Paola Inverardi and Corrado Priami. Evaluation of Tools for the Analysis of Communicating Systems. Pisa Science Center Technical Report HPL-PSC-91-25, Hewlett-Packard Laboratories, Pise, Italie, June 1991.

[ISO87] ISO. LOTOS — A Formal Description Technique Based on the Temporal Ordering of Observational Behaviour. Draft International Standard 8807, International Organization for Standardization — Information Processing Systems — Open Systems Interconnection, Genève, July 1987.

[ISO88] ISO. ESTELLE — A Formal Description Technique Based on an Extended State Transition Model. International Standard 9074, International Organization for Standardization — Information Processing Systems — Open Systems Interconnection, Genève, September 1988.

[J91] Thierry Jéron. *Contribution à la validation des protocoles : test d'infinitude et vérification à la volée*. Thèse de doctorat, Université de Rennes 1, May 1991.

[JJ89] Claude Jard and Thierry Jéron. *On-Line Model-Checking for Finite Linear Temporal Logic Specifications*. In Joseph Sifakis, editor, *Proceedings of the Workshop on Automatic Verification Methods for Finite State Systems (Grenoble, France)*, volume 407 of *Lecture Notes in Computer Science*, pages 189–196. Springer Verlag, Berlin, June 1989.

[JJ91] Claude Jard and Thierry Jéron. Bounded-memory Algorithms for Verification "On-the-fly". In K. G. Larsen, editor, *Proceedings of the 3rd Workshop on Computer-Aided Verification (Aalborg, Denmark)*, July 1991.

[Knu73] Donald E. Knuth. *The Art of Computer Programming*, volume III : Sorting and Searching of *Computer Science and Information Processing*. Addison-Wesley, Reading, Massachusetts, 1973.

[Kor91a] Henry Korver. Computing Distinguishing Formulas for Branching Bisimulation. In K. G. Larsen, editor, *Proceedings of the 3rd Workshop on Computer-Aided Verification (Aalborg, Denmark)*, July 1991.

[Kor91b] Henry Korver. The Current State of Bisimulation Tools. Technical Report P9101, University of Amsterdam - Programming Research Group, Amsterdam, 1991.

[KS83] P.C. Kanellakis and S.A. Smolka. CCS expressions, finite state processes, anf three problems of equivalence. In *Second ACM Symposium on Principles of Distributed Computing (PODC), Montreal, Quebec, Canada*, August 1983.

[Lam77] L. Lamport. Proving the correctness of multiprocess programs. *IEEE Transactions on Software Engineering*, 3(2):125–143, 1977.

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

[Mil89] Robin Milner. *Communication and Concurrency*. Prentice-Hall International, 1989.

[Mou89] Laurent Mounier. *Equivalence des systèmes de transitions étiquetées : Réduction et Diagnostic*. DEA, Institut National Polytechnique de Grenoble, June 1989.

[NH84] R. De Nicola and M. C. B. Hennessy. Testing Equivalences for Processes. *Theoretical Computer Science*, 34:83–133, 1984.

[NMV90] R. De Nicola, U. Montanari, and F. Vaandrager. Back and Forth Bisimulations. CS R9021, Centrum voor Wiskunde en Informatica, Amsterdam, May 1990.

[NV90] R. De Nicola and F. Vaandrager. Three logics for branching bisimulation. In *Proceedings 5th Annual Symposium on Logic in Computer Science (LICS 90), Philadelphia USA*, pages 118–129, Los Alamitos, CA, June 1990. IEEE Computer Society Press.

[Par81] David Park. Concurrency and Automata on Infinite Sequences. In Peter Deussen, editor, *Theoretical Computer Science*, volume 104 of *Lecture Notes in Computer Science*, pages 167–183, Berlin, March 1981. Springer Verlag.

[PT87] Robert Paige and Robert E. Tarjan. Three Partition Refinement Algorithms. *SIAM Journal of Computing*, 16(6):973–989, December 1987.

[Qin91] Huajun Qin. Efficient Verification of Determinate Processes. In J.C.M Baeten and J.F. Groote, editors, *Proceedings of the 2nd International Conference on Concurrency Theory (CONCUR'91)*, pages 470–479, Amsterdam, August 1991. North-Holland.

[QS83] Jean-Pierre Queille and Joseph Sifakis. Fairness and Related Properties in Transition Systems — A Temporal Logic to Deal with Fairness. *Acta Informatica*, 19:195–220, 1983.

[Ras90] Anne Rasse. *CLEO : diagnostic des erreurs en XESAR.* Thèse de doctorat, Institut National Polytechnique de Grenoble, June 1990.

[Ras91] Anne Rasse. Error diagnosis in finite communicating systems. In K. G. Larsen, editor, *Proceedings of the 3rd Workshop on Computer-Aided Verification (Aalborg, Denmark)*, July 1991.

[RHR91] C. Ratel, N. Halbwachs, and P. Raymond. Programming and verifying critical systems by means of the synchronous data-flow programming language LUSTRE. In *ACM-SIGSOFT'91 Conference on Software for Critical Systems,* New Orleans, December 1991.

[Rod88] Carlos Rodríguez. *Spécification et validation de systèmes en XESAR.* Thèse de doctorat, Institut National Polytechnique de Grenoble, May 1988.

[SE90] Santosh K. Shrivastava and Paul. D. Ezhilchelvan. rel/REL: A Family of Reliable Multicast Protocol for High-Speed Networks. Technical report, University of Newcastle, Dept. of Computer Science, U.K, 1990.

[Sor87] Amélia Soriano. *VENUS : un outil d'aide à la vérification des systèmes communicants.* Thèse de doctorat, Université Joseph Fourier (Grenoble), 1987.

[Val90] A. Valmari. *A Stubborn Attack on State Explosion.* In R.P. Kurshan and E.M. Clarke, editors, *Proceedings of the Workshop on Computer-Aided Verification (Rutgers, USA).* June 1990.

[VT91] A. Valmari and M. Tienari. An Improved Failure Equivalence for Finite-State Systems with a Reduction Algorithm. In Bengt Jonsson, Joachim Parrow, and Björn Pehrson, editors, *Proceedings of the 11th IFIP International Workshop on Protocol Specification, Testing and Verification (Stockholm, Suède)*, Amsterdam, June 1991. IFIP, North-Holland.

[Wes86] C.H West. Protocol Validation by Random State Explosion. In *Proceedings of the 6th IFIP International Workshop on Protocol Specification, Testing and Verification (Montréal, Canada)*, Amsterdam, June 1986. IFIP, North-Holland.

[WL89] P. Wolper and V. Lovinfosse. Verifying Properties of Large Sets of Processes with Network Invariants. In Joseph Sifakis, editor, *Proceedings of the 1st International Workshop on Automatic Verification Methods for Finite State Systems (Grenoble, France)*, June 1989.