N° d'ordre : 1355

THÈSE

présentée en vue de l'obtention du titre de

DOCTEUR DE L'INSTITUT NATIONAL POLYTECHNIQUE DE TOULOUSE

Spécialité **Informatique**

par

Michel CHARPENTIER

Assistance à la Répartition de Systèmes Réactifs

Soutenue le 5 novembre 1997, devant le jury composé de :

M M.	**René JACQUART**	Président
	Joffroy BEAUQUIER **Dominique MÉRY**	Rapporteurs
	Mamoun FILALI **André SCHIPER**	Examinateurs
	Gérard PADIOU	Directeur de thèse

À Jean Christophe

Freedom is the freedom to say that two plus two make four. If that is granted, all else follows.

George Orwell
1984

Résumé

Nous nous intéressons à l'étude et à la preuve de correction d'algorithmes réactifs et répartis au sein du formalisme Unity. Les algorithmes que nous considérons sont supposés s'exécuter de manière parallèle, c'est-à-dire que plusieurs processus se partagent la réalisation des traitements. Nous supposons en outre que le comportement de ces processus est infini dans le temps (réaction) et qu'ils ne partagent ni horloge ni mémoire commune (répartition). Ces algorithmes sont décrits dans le langage Unity (systèmes de transitions équitables) et prouvés au sein de la logique associée (logique temporelle linéaire).

Dans ce contexte, nous pensons que les assistants de preuve sont nécessaires, mais que l'utilisation efficace de démonstrateurs est tributaire d'une modélisation précise, dans le formalisme utilisé, des points caractéristiques de la répartition. Nous nous focalisons plus spécifiquement sur la communication entre les processus et sur la formalisation de l'étape de placement (*mapping*) sur une architecture répartie.

Nous définissons un mécanisme abstrait de communication qui repose sur une relation d'observation entre les variables des processus. Ce mécanisme, au contraire d'une description explicite d'un échange de messages, permet de prendre en compte les aspects de communication séparément des aspects de synchronisation. Nous montrons que cette distinction claire entre les deux notions permet de simplifier la description et la preuve d'algorithmes répartis.

Nous définissons également le produit, un opérateur qui représente le placement d'un programme Unity sur une architecture répartie. Cet opérateur nous permet de montrer que, malgré la sémantique d'entrelacement de Unity, le parallélisme réel d'une implantation répartie d'un programme ne remet généralement pas en cause la preuve de correction de ce programme.

Abstract

We address the problem of studying and proving reactive and distributed programs within the UNITY framework. We consider parallel algorithms where different processes are concurrently executed. Moreover, we assume that the processes have an infinite behaviour (reaction) and that they share neither memory nor a common clock (distribution). The UNITY programming notation (fair transition systems) is used to describe these algorithms, and specifications and proofs are handled within the UNITY logic (temporal linear logic).

In this context, we think that proof assistants are necessary. However, we believe that distribution must be precisely characterized in UNITY before we can efficiently use automatic provers. Two problems are more specifically explored: communication between processes and mapping to distributed architectures.

We propose an abstract communication mechanism based on an observation relation between processes variables. The communication description provided by this mechanism is more abstract than the traditional message-passing model and allows a clear distinction between synchronization and pure communication. This abstraction of communication appears to be helpful to describe and prove distributed algorithms.

We also define the product, a new UNITY operator which can be used to formalize the mapping of programs to distributed architectures. Using this operator, we prove that, in spite of UNITY interleaving semantics, the true parallelism of a distributed implementation preserves most of the properties satisfied by a program.

Remerciements

À tout rapporteur, tout honneur. J'adresse donc mes premiers remerciements à Joffroy Beauquier et à Dominique Méry qui ont accepté d'être les rapporteurs de ce travail. J'ai eu avec eux des discussions qui m'ont apporté beaucoup et j'espère qu'eux aussi ont apprécié ces échanges. Je tiens également à exprimer ma gratitude aux autres membres du jury, René Jacquart, qui a bien voulu en être le président, et André Schiper. Je les remercie du temps qu'ils ont accordé à l'examen de ce travail en dépit de leurs nombreuses activités.

Bien entendu, je n'oublie ni Mamoun Filali ni Gérard Padiou, mais avant d'être des membres du jury, ils restent pour moi "le Gourou" et "le Chef" qui ont formé, avec Philippe Mauran, Philippe Quéinnec et moi-même, l'équipe dite "du commissariat".

Le travail présenté dans ce mémoire est avant tout l'œuvre de cette équipe. Chacun y a sa part, et je n'oublierai jamais les réunions de travail au 17 bis, rue Riquet. Elles reposaient essentiellement sur la rigueur mathématique du Gourou, les élucubrations de Philippe Mauran, les intuitions du Chef et les contre-exemples de Philippe Quéinnec. Quant à moi, je faisais le café. À chaque réunion, l'un de nous arrivait avec ses petits papiers, les distribuait aux autres et, debout au tableau, feutre en main, essuyait les multiples critiques de ses collègues. Après quoi, commençait un *brainstorming* forcené, avec ses schémas incompréhensibles, ses révélations aussi limpides que : "*et donc le dernier événement précède nécessairement tous les autres*" et parfois aussi ses crises de fou rire. Pour ces moments inoubliables, je tiens ici à remercier chaleureusement toute cette équipe.

Le moniteur de l'enseignement supérieur que j'ai été pendant la durée de cette thèse est également reconnaissant à Philippe Quéinnec pour l'aide qu'il m'a apportée dans cette tâche. Ses réponses patientes à mes innombrables questions m'ont permis de combler un peu mes profondes lacunes en matière de systèmes opératoires, centralisés et répartis. C'est à lui que je dois aussi mes premières bases en LaTeX, sans lesquelles ce document aurait sans doute une tout autre forme.

Je tiens aussi à souligner ici combien Gérard Padiou a été un Chef formidable. Il a su à la fois me laisser une grande liberté et me "pousser" quand il le fallait. Sa bonne humeur inébranlable et son éternel optimisme m'ont beaucoup aidé. Merci Gérard.

Enfin, je termine en mentionnant le groupe d'amateurs de café après le déjeuner pour leur bonne humeur lors de ces pauses quotidiennes, et je remercie Annie pour ses patientes leçons d'utilisation de la photocopieuse la plus compliquée du monde.

Table des matières

Introduction

Contexte de la recherche

L'utilisation de plus en plus importante de machines informatiques pose la question, de manière concrète, de la validité des programmes. La sémantique des langages de programmation étant mathématiquement définie, il est effectivement possible de démontrer la correction (ou la non correction) d'un programme. Ce type de preuve n'en reste pas moins un problème très complexe.

Cette complexité s'accroît encore lorsque les programmes et les architectures des machines qui les utilisent font intervenir la concurrence et la répartition. En effet, celles-ci engendrent deux difficultés supplémentaires par rapport à la programmation séquentielle. Le parallélisme conduit à une explosion des états possibles du programme lors de son exécution et la répartition entraîne la perte de toute connaissance immédiate d'un état global et d'un référentiel de temps unique.

Pour faire face à ces problèmes, il semble nécessaire de chercher à accroître la rigueur et la formalisation des méthodes et outils utilisés lors d'un développement. Cet objectif est celui de l'équipe *Modélisation et Validation de la Répartition* de l'IRIT (*Institut de Recherche en Informatique de Toulouse*) au sein de laquelle les travaux présentés ici ont été effectués.

Intéressée à l'origine par le domaine des systèmes opératoires répartis, l'équipe *Modélisation et Validation de la Répartition* s'attache plus particulièrement à l'étude et à la validation de programmes *répartis* et *réactifs*. Les programmes étudiés sont *concurrents* en ce sens qu'ils sont supposés s'exécuter de manière parallèle, c'est-à-dire que plusieurs agents actifs (processus) se partagent la réalisation des traitements, ce qui implique l'introduction de mécanismes de synchronisation. Ces programmes sont également *répartis*. Nous associons la notion de répartition à un renforcement des hypothèses sur la concurrence : les différents agents sont alors supposés ne partager ni horloge ni mémoire commune. Les conséquences immédiates de cette répartition sont donc l'absence d'un référentiel de temps commun et la perte de la connaissance d'un état global. Enfin, les programmes considérés sont *réactifs*, c'est-à-dire qu'ils ne réalisent pas le calcul d'une certaine fonction de paramètres donnés, mais qu'ils réagissent continuellement à un environnement extérieur. Leur exécution se présente alors sous la forme d'un processus infini dans le temps. Les programmes réactifs sont nécessairement concurrents (en concurrence avec leur environnement) et généralement répartis (communiquant avec l'environnement).

Problématique étudiée

Dans le contexte précédent, nous étudions la définition d'une approche méthodologique de développement axée sur deux principes : la preuve automatique de programme et la conception par raffinements successifs.

En effet, la preuve formelle que le comportement d'un programme correspond bien aux objectifs pour lesquels ce programme a été écrit constitue, a priori, la meilleure forme de validation possible. Dans un contexte de programmation répartie, cette approche par la preuve est d'autant plus intéressante que le non déterminisme de ce type de programmes (et donc la non reproductibilité de leur comportement) rend délicate une mise au point au moyen de tests.

Néanmoins, la preuve formelle d'un programme reste, plus encore que le test, un processus extrêmement fastidieux et complexe. La mise en œuvre d'une telle preuve nécessite donc, pour le concepteur, de bénéficier d'une aide à deux niveaux : d'une part la prise en charge du côté le moins complexe et le plus fastidieux de la preuve et d'autre part la possibilité de valider la preuve elle-même et de garantir sa correction. C'est la raison pour laquelle la preuve de programmes nous semble devoir être assistée par des outils informatiques et donc, dans une mesure qui reste à préciser, bénéficier d'un certain automatisme.

Malheureusement, la preuve de correction de programmes complexes ne saurait être entièrement automatique. Une intervention humaine est donc nécessaire au cours de ce processus de preuve. Elle peut se situer vers l'aval de la preuve, c'est-à-dire au niveau des obligations de preuves manipulées par les outils d'assistance. C'est alors, en quelque sorte, l'utilisateur qui fournit une assistance au démonstrateur incapable de résoudre le problème de façon entièrement automatique.

Nous préférons faire porter l'effort de conception de programme le plus possible vers l'amont de la preuve. Ceci permet au concepteur de programmes de raisonner au niveau de son problème algorithmique et du formalisme associé, sans qu'il soit besoin pour lui de se plonger dans les théories manipulées par l'outil qu'il veut utiliser.

Pour cela, nous cherchons à utiliser le raffinement de programmes comme moyen pour simplifier le problème de départ, jusqu'à ce qu'il soit abordable par un outil entièrement automatique. En particulier, dans le contexte où nous nous plaçons, le raffinement est utilisé pour masquer au maximum les difficultés propres à la répartition.

Thèse soutenue

Nous pensons que l'utilisation efficace de démonstrateurs, dans le cadre de la preuve de correction d'algorithmes répartis, est tributaire d'une modélisation précise, dans le formalisme utilisé, des points caractéristiques de la répartition.

Dans le but de permettre cette modélisation de la répartition dans le formalisme UNITY, nous nous focalisons sur la représentation de la communication et la formalisation de l'étape de placement sur une architecture répartie.

Nous définissons un mécanisme abstrait de communication qui repose sur une relation d'observation entre les variables des processus. Ce mécanisme, au contraire d'une description explicite d'un échange de messages, permet de prendre en compte les aspects de communication séparément des aspects de synchronisation. Nous montrons que cette distinction claire entre les deux notions permet de simplifier la description et la preuve d'algorithmes répartis.

Nous définissons également le produit, un opérateur qui représente le placement d'un programme UNITY sur une architecture répartie. Cet opérateur nous permet de montrer que, malgré la sémantique d'entrelacement de UNITY, le parallélisme réel d'une implantation répartie d'un programme ne remet généralement pas en cause la preuve de correction de ce programme.

Plan général

Le chapitre 1 rappelle les bases de la formalisation des systèmes réactifs. Nous présentons succinctement différents formalismes de programmation et de spécification, puis nous détaillons plus particulièrement les systèmes de transitions. À partir d'une définition formelle de ces systèmes, nous décrivons leur interprétation opérationnelle, ainsi que différentes formes d'équité qui peuvent leur être associées. Enfin, nous rappelons, dans ce contexte, la définition formelle des notions de sûreté et de vivacité.

Nous nous attachons, dans le chapitre 2, à un formalisme particulier, UNITY, qui est utilisé dans toute la suite. Nous reprenons les grandes lignes de la présentation d'origine en décrivant le langage UNITY, la logique associée, l'interprétation opérationnelle des programmes, le système de preuve et les notions de composition. Nous montrons comment les programmes UNITY peuvent être interprétés en termes de systèmes de transitions équitables, ce qui conduit à la définition opérationnelle de la logique. Nous terminons cette première partie par un exemple de programme et de spécification.

Une deuxième partie de ce chapitre se focalise sur les problèmes de consistance et de complétude liés à l'axiome de substitution. Après une mise en évidence des difficultés issues de l'utilisation de cet axiome, des explications intuitives sont données et la solution proposée par B.A. Sanders est présentée.

Enfin, une dernière partie présente brièvement trois redéfinitions et extensions du formalisme UNITY qui visent à améliorer la compositionnalité du système de preuve.

Le chapitre 3 propose une ébauche de classification des systèmes d'assistance à la preuve de programmes dans le contexte du formalisme UNITY ou de formalismes voisins. À partir de cette classification, nous présentons quelques outils existants dont les particularités nous ont semblé intéressantes.

La notion de raffinement de programmes est détaillée dans le chapitre 4. Nous y distinguons une notion de raffinement d'instruction et une notion de raffinement de comportement. Différentes notions de raffinement de comportement sont décrites et comparées dans le formalisme UNITY, et la notion de raffinement d'instructions UNITY est évoquée. Le chapitre se termine par la présentation de deux transformations de programmes possédant la propriété de raffinement. Nous détaillons un exemple qui utilise ces deux transformations pour modifier le système de communication entre deux processus sans remettre en cause la correction du programme.

Le chapitre 5 présente la relation d'observation sur laquelle se fonde notre abstraction de la communication dans les systèmes répartis. Le rôle de cette abstraction est de faciliter la description et la preuve d'algorithmes répartis. Nous commençons par définir formellement la relation d'observation et nous montrons que cette relation peut s'apparenter à un opérateur de logique temporelle muni de propriétés le reliant aux autres opérateurs de la logique UNITY. Nous proposons ensuite une extension du langage UNITY dans laquelle la notion d'observation fournit un mécanisme abstrait permettant de décrire la communication entre les processus d'un système. Nous définissons une sémantique opérationnelle pour les programmes UNITY ainsi étendus. Nous appliquons alors cette nouvelle forme de communication à différents algorithmes classiques. Le chapitre se termine par la modélisation et la preuve détaillée d'un algorithme d'exclusion mutuelle qui utilise l'observation comme moyen de communication.

Nous développons une comparaison entre le mécanisme d'observation et l'affectation UNITY dans le chapitre 6. À travers leurs points communs et leurs différences, cette étude met en évidence la nécessité, pour un processus qui observe son environnement, de percevoir en plus les liens de causalité entre les événements qu'il observe.

Le chapitre 7 s'attache à formaliser le placement de programmes UNITY sur une architecture répartie. Nous commençons par montrer les difficultés particulières, en terme de parallélisme, que présente ce type de placement. Nous définissons ensuite un opérateur de composition de programmes UNITY permettant de décrire précisément le placement d'un programme sur une architecture répartie. Nous montrons que le passage de l'union de programmes traditionnelle à cette forme de composition preserve généralement la correction du système, ce qui, d'une certaine manière, justifie l'emploi de l'opérateur d'union pour la preuve de systèmes répartis en UNITY.

Enfin, le chapitre 8 présente le prototype d'assistant de preuve que nous avons développé. Cet assistant se présente sous la forme d'un éditeur syntaxique du langage et de la logique UNITY. Il fait appel à un démonstrateur automatique de théorèmes et à une procédure de décision arithmétique pour vérifier dynamiquement la consistance entre la spécification et le programme décrits. Nous décrivons quelques exemples de programmes UNITY validés à l'aide de cet outil.

Notations

Application de fonction

Nous utilisons une notation pointée pour l'application de fonction. La notation $f.x$ représente l'application de la fonction f à l'élément x. Les fonctions à plusieurs paramètres utilisent les parenthèses comme constructeurs de n-uplets : $f.(x, y)$ représente l'application de f à x et à y ou l'application de f au couple (x, y).

Nous supposons l'opérateur "." associatif à droite : $f.g.h$ représente $f.(g.h)$.

Enfin, nous conservons une notation traditionnelle pour les indices. Nous parlerons de l'état σ_i ou du programme P_i.

Quantification

Nous reprenons la notation utilisée pour la quantification dans le livre de K.M. Chandy et J. Misra sur UNITY. Elle utilise les deux délimiteurs $\langle$ et $\rangle$. Elle est de la forme :

$$\langle \; \textit{opérateur variable(s)} \; : \; \textit{prédicat} \; :: \; \textit{expression quantifiée} \; \rangle$$

Cette notation représente la composition, par l'*opérateur* de toutes les *expressions quantifiées* pour lesquelles les *variables* vérifient le *prédicat*.

L'opérateur utilisé doit être associatif et commutatif. Nous utilisons les opérateurs min, max, $\wedge$, $\vee$, $\cap$, $\cup$, $[\!]$, $[\![\!]\!]$, etc. Nous notons souvent $\langle\forall \cdots\rangle$ pour $\langle\wedge \cdots\rangle$ et $\langle\exists \cdots\rangle$ pour $\langle\vee \cdots\rangle$.

Lorsqu'il n'y a pas d'ambiguïté sur le domaine de variation des variables, le prédicat est omis. Dans ce cas, les ":" qui séparent les variables du prédicat sont inutiles et disparaissent. Par exemple, on note $\langle\forall i :: \cdots\rangle$ pour $\langle\forall i : i \in \mathbb{N} :: \cdots\rangle$ ou $\langle\forall s :: \cdots\rangle$ pour $\langle\forall s : 1 \leqslant s \leqslant N :: \cdots\rangle$.

Lorsque nous quantifions des propriétés de logique UNITY, nous omettons les symboles $\langle$ et $\rangle$. Par exemple, on écrit :

$$\forall k :: \texttt{stable}\ x \geqslant k$$

Séquences d'états

Nous utilisons souvent des séquences d'états pour représenter les exécutions des programmes. Nous notons généralement ces séquences σ, σ', etc. Le $i^{ème}$

état de la séquence σ est noté σ_i. Par extension, nous désignons parfois un état quelconque d'un programme par σ_i alors qu'il n'est fait référence à aucune exécution ni aucun indice particulier. On écrit, par exemple : *soit* σ_i *un état du programme* F, ou $\langle \forall \sigma_i :: \cdots \rangle$.

Un exemple de séquence est généralement donné sous la forme :

$$(x, y) : (a, b) \longrightarrow (c, d) \cdots$$

ce qui signifie que la séquence débute dans un état où $x = a$ et $y = b$, que dans l'état suivant $x = c$ et $y = d$, etc.

Chapitre 1

Formalisation des programmes réactifs

Ce chapitre commence par une présentation succincte des différents modèles formels susceptibles de décrire les systèmes réactifs. Il s'attache ensuite plus spécifiquement au modèle de parallélisme par entrelacement le plus répandu, en l'occurrence les systèmes de transitions. Leur définition est rappelée ainsi que la construction des exécutions opérationnelles de ces systèmes. Ces exécutions fournissent, sous la forme de séquences d'états, une vision abstraite du comportement du système décrit. Nous rappelons ensuite les principales formes d'équité qui peuvent être utilisées pour étendre ces systèmes. Enfin, les notions de sûreté et de vivacité sont formellement définies dans ce contexte.

1.1 Présentation des modèles

Raisonner sur des programmes nécessite de disposer de formalismes rigoureux pour décrire le comportement de ces programmes et les propriétés attendues. Nous distinguons ici les formalismes de programmation des formalismes de spécification. Nous verrons que les objets décrits par ces formalismes peuvent éventuellement être les mêmes comme, par exemple, un ensemble de séquences d'états. Néanmoins, les formalismes de programmation fournissent une description plus explicite et constructive, adaptée à une implantation, alors que les formalismes de spécification proposent une vision plus abstraite et plus intuitive.

1.1.1 Formalismes de programmation

Le langage choisi pour la description d'un modèle doit être capable d'exprimer toute la gamme des applications auxquelles nous nous intéressons, en particulier leurs aspects *concurrents* et *réactifs*. Un programme réactif est un programme dont le rôle est de maintenir une perpétuelle interaction avec son environnement plutôt que de calculer une valeur finale à sa terminaison. Cette notion de réaction nous interdit de considérer les programmes en terme de fonctions réalisant le calcul d'une valeur de sortie à partir de la donnée de paramètres d'entrée. En particulier, les programmes réactifs ne se terminent pas et leur comportement se répète à l'infini.

Il convient donc d'adopter une description, moins abstraite que celles utilisées pour les programmes transformationnels, qui consiste à décrire, d'une manière ou d'une autre, les différentes actions (ou événements) dont le système est capable au cours du temps, ainsi que leurs effets sur lui-même et sur son environnement. Ce type de description permet l'expression des phénomènes propres à ces systèmes comme par exemple l'interblocage, l'exclusion mutuelle ou la famine.

Un autre aspect important que ces modèles doivent prendre en compte est celui de la *concurrence* et donc du parallélisme. On peut remarquer que les notions de concurrence et de réaction sont étroitement liées. En effet, un programme réactif est nécessairement concurrent puisqu'il s'exécute en concurrence avec son environnement. D'autre part, les différentes parties d'un programme concurrent peuvent être vues comme des programmes réactifs, chacun ayant pour environnement tous les autres.

On distingue généralement deux approches pour exprimer le parallélisme au sein d'un modèle :

– les modèles par entrelacement. À partir d'une hypothèse *d'atomicité* de certaines instructions, ces modèles manipulent les entrelacements possibles entre les suites d'instructions atomiques des programmes s'exécutant en concurrence. Le parallélisme est représenté par le *non déterminisme* du choix de la séquence réellement exécutée. L'exécution parallèle de deux instructions atomiques s_1 et s_2 correspond donc à l'une des exécutions séquentielles s_1 puis s_2 ou s_2 puis s_1.

  L'avantage majeur de ce type d'approche est de permettre la description d'applications parallèles à partir d'une sémantique séquentielle. Par contre, les hypothèses d'atomicité et de non simultanéité des instructions peuvent s'avérer contraignantes vis-à-vis de la réalité du système que l'on veut modéliser. Nous reviendrons sur le problème de l'exécution simultanée d'instructions atomiques au chapitre 7.

– les modèles non entrelacés, encore appelés modèles de "vrai" parallélisme. Ces modèles abandonnent les hypothèses d'atomicité et d'entrelacement et se définissent par rapport à une notion de *causalité* entre instructions. Par rapport au temps, qui suppose un ordre total, la causalité se décrit à partir d'une relation d'ordre partielle entre les instructions. Ceci permet de définir le séquencement (deux instructions sont causalement liées) et le parallélisme (deux instructions sont causalement indépendantes).

  Bien que ces modèles permettent d'éviter les difficultés liées aux modèles entrelacés, leur description est cependant beaucoup plus abstraite. Ceci les rend plus délicats à manipuler pour la description explicite d'algorithmes.

Parmi les modèles à entrelacement, on compte principalement les systèmes de transitions qui sont détaillés au paragraphe suivant. Les modèles non entrelacés les plus répandus sont les structures d'événements et les systèmes de transitions asynchrones [1].

Notons enfin que des modèles usuels comme les algèbres de processus ou les réseaux de Petri constituent des descriptions de plus haut niveau qui peuvent s'interpréter dans l'un des modèles précédents (systèmes de transitions par exemple). Notons également que de nombreux formalismes, d'apparences différentes, sont en fait fondés sur les systèmes de transitions (UNITY, TLA, les systèmes utilisés par Z. Manna et A. Pnueli, ceux de A.U. Shankar), parfois en leur ajoutant des spécificités de communication (CSP, CCS, LOTOS, ESTELLE) ou de synchronisme (LUSTRE, ESTEREL).

1.1.2 Formalismes de spécification

Les langages de spécification se caractérisent par la volonté d'exprimer des propriétés des systèmes tout en restant simple et expressif. Les logiques temporelles sont bien adaptées aux comportement infinis des systèmes réactifs et aux notions de sûreté et de vivacité, essentielles pour ces systèmes. Elles allient une grande puissance d'expression à l'utilisation d'opérateurs dont le sens reste intuitif. Il s'agit d'une variété de logiques modales dans lesquelles les mondes possibles de la sémantique de Kripke correspondent à différentes dates dans le temps. Les similitudes entre les structures d'interprétation de ces logiques et les structures utilisées pour la sémantique opérationnelle des programmes concurrents (succession d'états) facilitent l'utilisation de telles logiques dans le cadre de la théorie de la programmation parallèle.

Dans ce cadre, on distingue deux grandes familles de logiques temporelles selon que le temps, que l'on choisit dans tous les cas discret, possède une structure arborescente ou linéaire. Parmi les logiques temporelles à temps arborescent, la logique CTL(*Computation Tree Logic*) est la plus couramment utilisée. Le temps y est linéaire et fini vers le passé (le passé est déterminé) et arborescent et infini vers l'avenir (l'avenir est indéterminé). D'autre part, on utilise également beaucoup une logique temporelle linéaire pour laquelle le passé est borné (instant initial) et l'avenir infini. Cette logique a été abondamment utilisée pour la spécification de systèmes réactifs, à la fois sous sa forme originale et sous une forme généralisée à des formules binaires sur les états (TLA). La logique UNITY peut également être vue comme une restriction de cette logique.

On peut considérer que les méthodes qui reposent sur les systèmes de transitions constituent une famille de formalismes liés par un lien de parenté étroit. Même si une propriété d'un système particulier peut ne pas être vraie dans un autre système, elle peut généralement s'y transposer. En particulier, toute l'étude qui est menée ici autour du formalisme UNITY repose sur des principes généraux qui sont ceux des systèmes de transitions équitables et qui peuvent

1. Le terme qui désigne ces systèmes est ambigu, puisqu'il ne s'agit pas de systèmes de transitions dans lesquels les transitions sont asynchrones, mais de systèmes de transitions faisant intervenir du parallélisme non entrelacé à partir d'une notion d'indépendance entre transitions.

déboucher sur des propriétés analogues au sein de ces systèmes. C'est pourquoi nous choisissons d'introduire ici quelques principes communs à ces modèles, le formalisme UNITY bénéficiant d'une présentation plus complète au chapitre 2.

1.2 Les systèmes de transitions

Le comportement global d'un système peut être formellement décrit par un système de transitions. Certains formalismes comme UNITY (les transitions y sont représentées par des affectations) ou TLA (les transitions y sont représentées par des prédicats binaires) s'appuient directement sur ce type de description. D'autres peuvent également utiliser ces systèmes comme définition rigoureuse de leur sémantique même s'ils s'utilisent sous une forme radicalement différente (réseaux de Petri, langages algébriques).

1.2.1 Définition

Formellement, un systèmes de transitions est un quadruplet $(\mathcal{V}, \Sigma, \mathcal{T}, \phi)$ dans lequel :

- $\mathcal{V}$ représente un ensemble fini de variables d'état ;
- Σ est l'ensemble des états possibles du système. Un état $\sigma_i \in \Sigma$ est une fonction qui associe à chaque variable v de $\mathcal{V}$ une valeur de son domaine notée $\sigma_i.v$;
- $\mathcal{T}$ est l'ensemble fini des transitions du système. Une transition $\tau \in \mathcal{T}$ est une fonction de Σ dans 2^{Σ} qui associe à un état $\sigma_i \in \Sigma$ l'ensemble $\tau.\sigma_i$ de ses successeurs par cette transition. On suppose qu'un système contient toujours la transition *skip* telle que $\langle \forall \sigma_i : \sigma_i \in \Sigma :: \mathit{skip}.\sigma_i = \{\sigma_i\} \rangle$;
- ϕ représente la condition initiale du système. C'est un prédicat portant sur les variables d'état qui caractérise les états initiaux possibles du système. On impose que ce prédicat soit satisfiable.

Puisque le nombre de transitions d'un système est fini, si on a :

$$\langle \forall \tau : \tau \in \mathcal{T} :: \langle \forall \sigma_i : \sigma_i \in \Sigma :: \tau.\sigma_i \text{ est fini } \rangle \rangle$$

on en déduit que chaque état possède un nombre *fini* (éventuellement nul) de successeurs. Dans ce cas, on dit que le système est *à branchement fini*. C'est le cas de la plupart des systèmes et en particulier de ceux décrits dans le modèle UNITY[2].

2. En UNITY, les instructions sont toujours exécutables et déterministes et donc chaque état possède un successeur et un seul pour chaque transition.

1.2.2 Interprétation opérationnelle

Une *exécution* d'un système de transitions est constituée de la suite des différents états pris par le système. Comme on s'intéresse à des programmes réactifs qui ne se terminent pas, cette suite d'état est infinie [3]. Le premier état d'une exécution doit satisfaire au prédicat ϕ et le passage d'un état au suivant doit résulter d'une transition de $\mathcal{T}$. Formellement, une exécution est une suite d'états $\sigma = (\sigma_0, \sigma_1, \cdots)$ vérifiant :

$$\sigma_0.\phi \ \wedge \ \langle \forall i : i \geqslant 0 :: \langle \exists \tau : \tau \in \mathcal{T} :: \sigma_{i+1} \in \tau.\sigma_i \rangle \rangle$$

Sur ces séquences, nous pouvons définir la notion de *bégaiement* qui nous sera utile par la suite. On dit qu'une séquence bégaie lorsqu'elle répète le même état un nombre fini de fois (au moins égal à deux). On définit alors l'opérateur $\natural$ de suppression du bégaiement : la séquence $\natural\sigma$ correspond à la séquence σ dans laquelle tous les segments maximaux finis composés d'états identiques sont remplacés par une seule occurrence de l'état [4]. L'opérateur $\natural$ permet de définir l'opérateur Γ qui associe à un ensemble de séquences toutes les séquences qui leur sont équivalentes au bégaiement près :

$$\Gamma.E = \{\sigma \mid \langle \exists \sigma' : \sigma' \in E :: \natural\sigma = \natural\sigma' \rangle\}$$

L'opérateur Γ est utilisé dans la définition de relations de raffinement au chapitre 4.

On peut donc représenter un système de transitions de deux manières :

- en décrivant explicitement l'ensemble de ses transitions ;
- en décrivant, à l'aide de propriétés logiques, l'ensemble de ses exécutions possibles.

La première description correspond en fait à la donnée d'un *programme* et la seconde à la donnée d'une *spécification*. Dans les deux cas, c'est un système de transitions qui est décrit, soit explicitement, soit implicitement. Nous conserverons tout au long de ce qui suit cette distinction entre programme et spécification selon que la description des transitions d'un système est faite explicitement ou implicitement.

1.2.3 Les systèmes de transitions équitables

On peut enrichir les systèmes de transitions en associant à chaque transition une notion d'équité. On distingue principalement deux types d'équité : l'équité faible (ou *justice*) et l'équité forte (ou *compassion*).

3. On peut néanmoins utiliser ces exécutions infinies pour décrire des programmes qui se terminent. Un tel programme correspond alors à un système dans lequel, à partir d'un certain moment, seule la transition *skip* est utilisée. On représente donc un système qui se termine par un système qui réalise indéfiniment une opération sans effet.

4. Signalons dès à présent que les séquences σ et $\natural\sigma$ sont strictement équivalentes vis-à-vis de la logique Unity.

L'équité faible exprime qu'une transition ne peut pas rester infiniment longtemps (continûment) exécutable[5] sans être finalement exécutée. L'équité forte exprime qu'une transition ne peut pas être exécutable infiniment souvent sans être finalement exécutée. Formellement, les exécutions σ d'un système de transitions équitable doivent vérifier :

- $\langle \ \forall \tau : \tau \in \mathcal{T} \wedge \tau$ faiblement équitable $::$
  $\neg\langle \exists i_0 : i_0 \geqslant 0 :: \langle \forall i : i \geqslant i_0 :: \tau.\sigma_i \neq \emptyset \wedge \sigma_{i+1} \notin \tau.\sigma_i \rangle\rangle$
  $\rangle$
- $\langle \ \forall \tau : \tau \in \mathcal{T} \wedge \tau$ fortement équitable $::$
  $\neg\langle \exists i_0 : i_0 \geqslant 0 :: \langle \forall i : i \geqslant i_0 :: \langle \exists j : j \geqslant i :: \tau.\sigma_j \neq \emptyset \rangle \wedge \sigma_{i+1} \notin \tau.\sigma_i \rangle\rangle$
  $\rangle$

On remarquera que, étant donné que les systèmes que nous considérons sont non étiquetés, on impose qu'une transition est effectuée en imposant seulement de retrouver dans l'exécution les effets de cette transition, éventuellement obtenus grâce à une autre transition.

Ces notions d'équité permettent d'exclure certaines exécutions du système dont le comportement n'est pas conforme à la réalité d'un système concurrent. En effet, si on considère que les différentes transitions d'un système sont exécutées par un ensemble de processus concurrents, l'équité faible permet d'exprimer qu'aucun processus ne s'arrête (c'est-à-dire ne devient infiniment lent) en ce sens que les transitions correspondant à ses instructions sont finalement exécutées. L'équité forte permet quant à elle de spécifier que les transitions correspondant à des synchronisations entre processus seront exécutées à partir du moment où ces synchronisations sont possibles infiniment souvent.

Nous verrons au chapitre 2 que le modèle Unity correspond à un système de transitions équitable dans lequel toutes les transitions sont soumises à l'équité faible.

1.3 Propriétés de sûreté et de vivacité

1.3.1 Présentation

On a vu que l'on pouvait considérer une spécification comme une description des propriétés des exécutions possibles d'un système de transitions. Dans la spécification des systèmes réactifs, on distingue généralement les propriétés de *sûreté* des propriétés de *vivacité*.

Intuitivement, une propriété de sûreté exprime qu'un état interdit ne sera jamais atteint. Par exemple, le fait que deux processus ne seront jamais présents simultanément dans la même section critique est une propriété de sûreté. Ces propriétés ne sont en général pas suffisantes pour décrire complètement un système. En particulier, à moins d'avoir pour état initial un état interdit, un système de transition dont la seule transition est *skip* (c'est-à-dire un programme qui ne fait rien) satisfait toutes les propriétés de sûreté[6].

5. Une transition τ est exécutable dans un état σ_i si et seulement si $\tau.\sigma_i \neq \emptyset$.
6. "He who does not, errs not" [MP92].

Les propriétés de vivacité expriment, quant à elles, qu'un état utile sera atteint. Par exemple, le fait qu'un processus qui demande l'accès à une section critique finit par y entrer est une propriété de vivacité.

Le "bon" fonctionnement d'un programme transformationnel classique est généralement spécifié par une propriété de sûreté (la correction partielle) et une propriété de vivacité (la terminaison). Un résultat similaire, dû à B. Alpern et F.B. Schneider, existe pour les programmes réactifs : toute propriété (c'est-à-dire tout ensemble d'exécutions) peut se définir par la conjonction d'une propriété de sûreté et d'une propriété de vivacité.

1.3.2 Définition formelle

Formellement, une exécution $\sigma = (\sigma_0, \sigma_1, \cdots)$ d'un système de transitions $(\mathcal{V}, \Sigma, \mathcal{T}, \phi)$ est un élément de Σ^ω et une propriété P est un ensemble d'exécutions, c'est-à-dire un élément de 2^{Σ^ω}. De la même manière, Σ^* représente l'ensemble des préfixes (sous-entendus stricts, donc finis) des exécutions, c'est-à-dire les suites *finies* $(\sigma_0, \sigma_1, \cdots, \sigma_n)$. On a alors les définitions suivantes :

–
$$\begin{gathered} P \text{ est une propriété de sûreté} \\ \equiv \\ \langle \forall \sigma : \sigma \in \Sigma^\omega :: \sigma \notin P \Rightarrow \langle \exists i : i \geqslant 0 :: \langle \forall \beta : \beta \in \Sigma^\omega :: \sigma_0 \cdots \sigma_i \beta \notin P \rangle\rangle\rangle \end{gathered}$$

–
$$\begin{gathered} P \text{ est une propriété de vivacité} \\ \equiv \\ \langle \forall \alpha : \alpha \in \Sigma^* :: \langle \exists \beta : \beta \in \Sigma^\omega :: \alpha\beta \in P \rangle\rangle \end{gathered}$$

On voit donc qu'une propriété de sûreté est une propriété caractérisée par les préfixes finis en ce sens que si une exécution ne vérifie pas la propriété, elle possède nécessairement un préfixe dont aucune des extensions possibles ne vérifie la propriété (c'est-à-dire qu'un état interdit a été atteint et qu'à partir de là, il devient impossible de respecter la propriété). À l'inverse, un préfixe quelconque d'une exécution peut toujours être étendu de manière à satisfaire une propriété de vivacité donnée.

Les notions d'équité et de vivacité sont intimement liées. En effet, un système de transitions à branchement fini *sans équité d'aucune sorte* peut être caractérisé par les préfixes finis de ses exécutions, c'est-à-dire par ses propriétés de sûreté : à partir du moment où deux systèmes ont les mêmes propriétés de sûreté, ils ont le même ensemble d'exécutions. Ce résultat n'est plus valide dans le cas des systèmes à branchement infini. En particulier, les différentes notions d'équité peuvent être réalisées au sein d'un système sans équité à l'aide de l'instruction *choose*(x) qui affecte à la variable x un nombre indéterminé de $\mathbb{N}$. On voit donc que les propriétés de vivacité n'ont d'intérêt pour des systèmes à branchement fini que dans la mesure où ces systèmes sont équitables.

1.4 Bibliographie

Une comparaison des différents modèles de représentation des systèmes concurrents est présentée dans [WN95]. Elle s'appuie sur la théorie des catégories pour représenter au sein d'un formalisme commun ces différents modèles.

On peut également consulter des présentations spécifiques aux modèles que nous avons cités comme TLA [Lam94], les systèmes de Z. Manna et A. Pnueli [MP92], les systèmes utilisés par A.U. Shankar [Sha93] ou CSP [Hoa84].

Une introduction à la logique temporelle et à son utilisation pour la spécification des systèmes concurrents se trouve dans [AEdC90].

La définition des systèmes de transitions et des notions d'équité associées sont largement détaillées dans [MP92, Sha93]. On trouve dans [MP92] une réalisation de l'équité faible à l'aide de l'instruction *choose*.

La notion de bégaiement, ainsi que les définitions des opérateurs $\natural$ et Γ sont présentées dans [AL91].

Les définitions formelles des notions de sûreté et de vivacité sont reprises de [AS85] ainsi que le résultat concernant la décomposition de toute propriété en une conjonction d'une propriété de sûreté et d'une propriété de vivacité.

Chapitre 2

Le formalisme UNITY

Ce chapitre a pour objectif de présenter, de manière assez complète, le formalisme Unity. Deux raisons justifient le choix d'une présentation aussi détaillée.

D'abord, ce qui sera présenté au cours des chapitres suivants s'appuie sur ce formalisme. Même si rien n'empêche a priori d'adapter les résultats présentés à des formalismes voisins, la syntaxe utilisée sera toujours celle de Unity et les preuves présentées ne seront valables que dans le contexte précis de ce formalisme.

Ensuite, nous préférons, dès le départ, tout faire pour éviter des confusions possibles liées à certaines ambiguïtés du formalisme Unity. En effet, outre le fait que Unity reste relativement récent et qu'aucun formalisme ne possède dès la naissance une forme parfaite et immuable, ce formalisme a souffert énormément d'une faute dans sa définition d'origine. Le célèbre axiome de substitution a fait couler beaucoup d'encre et a incité de nombreuses personnes à proposer des versions modifiées du formalisme. Ces modifications ont vite dépassé le cadre de l'axiome de substitution et, malheureusement, aucune d'entre elles ne fait l'unanimité. Il devient donc nécessaire de préciser, lorsque l'on parle de Unity, de quel formalisme il s'agit sous peine de grande confusion. Cet aspect est d'autant plus important, dans le cas présent, que nous sommes amenés, dans la suite, à nous appuyer plus spécifiquement sur l'une ou l'autre des définitions selon le problème abordé.

Pour ces deux raisons, nous présentons ici les principales définitions du langage et de la logique Unity. Chaque aspect du formalisme que nous utilisons dans les chapitres suivants se voit attribuer une notation propre qui sera reprise dans la suite. Nous nous attachons également à lier entre elles ces différentes définitions et extensions de manière à pouvoir regrouper plusieurs propositions sous une syntaxe commune.

Ce chapitre commence donc par une présentation détaillée du formalisme tel qu'il a été défini à l'origine. Le langage est d'abord présenté puis la logique est introduite. Les notions de composition de programmes sont brièvement décrites. Le langage Unity est alors relié aux systèmes de transitions d'états et sa logique aux propriétés de ces systèmes. Cette première partie se termine par un exemple de programme et de spécification.

Une deuxième partie s'attache plus spécifiquement aux problèmes de consistance et de complétude liés à l'axiome de substitution. Après une mise en évidence des difficultés issues de l'utilisation de cet axiome, des explications intuitives sont données et la solution proposée par B.A. Sanders est présentée.

Enfin, une dernière partie présente brièvement trois redéfinitions et extensions du formalisme qui visent à améliorer la compositionnalité du système de preuve.

2.1 Présentation d'origine

2.1.1 Introduction

Le formalisme Unity a été défini par K.M. Chandy et J. Misra en 1988. Il tire son appellation de la vision qu'ont les auteurs d'une certaine *unité* dans le développement des programmes. Leur conviction est que les programmes peuvent (et doivent) être conçus selon une méthode qui ne dépend ni de l'architecture choisie ni du domaine d'application des programmes. C'est pourquoi ils proposent avec Unity une théorie simple et universelle capable de représenter de manière uniforme les différents types d'applications, qu'elles soient synchrones ou asynchrones, à mémoire partagée ou à base d'échanges de messages, etc. Le formalisme Unity permet en outre de représenter un programme sous différents angles, du plus abstrait (la spécification) au plus opérationnel (l'implantation) au sein de la même théorie. Cet aspect est particulièrement important dans l'optique d'une conception de programmes par raffinements.

Le formalisme Unity propose donc un langage de programmation, inspiré du langage des commandes gardées de E. Dijkstra, un langage de spécification s'appuyant sur une logique temporelle linéaire et une théorie permettant de prouver sur les programmes des propriétés exprimées dans le langage de spécification.

2.1.2 Le langage UNITY

L'objectif de K.M. Chandy et de J. Misra n'a jamais été de définir complètement et rigoureusement un véritable langage de programmation. Comme nous allons le voir, la syntaxe de programmation de Unity n'est qu'une expression commode d'un système de transitions équitable tel que nous les avons introduits au chapitre 1. Toutes les personnes qui ont travaillé sur le formalisme Unity se sont permis des entorses à la syntaxe d'origine. Ces écarts ne sont pas gênants à partir du moment où l'on reste capable de distinguer le système de transitions sous-jacent.

Un programme Unity est composé de quatre parties :

Declare :

Il s'agit d'un ensemble de déclarations de variables tel qu'on peut le trouver dans les langages de programmation classiques. Les types de base utilisés sont les entiers et les booléens, à partir desquels les autres types sont définis par les constructeurs habituels d'enregistrement, de tableau, d'ensemble ou de liste.

Initially :

Cette partie concerne l'initialisation des variables. Chaque variable est initialisée au plus une fois avec une valeur de son type. Les variables non initialisées ont des valeurs non déterminées. Une partie *initially* n'est valide que dans la mesure où ses initialisations ne sont pas circulaires : il doit exister un ordre entre les affectations d'initialisation de façon à ce que chaque variable initialisée le soit en fonction de constantes ou de variables déjà initialisées.

Always :

La partie *always* comporte un ensemble de macro-définitions. Un certain nombre de variables, dites "transparentes", sont définies en fonction d'autres variables (transparentes ou non). La même restriction que pour la partie *initially* s'applique : il doit exister un ordre d'évaluation des définitions qui permette de définir chaque variable transparente en fonction de variables non transparentes et de variables transparentes déjà définies. Une restriction supplémentaire est ajoutée interdisant aux variables transparentes d'être affectées par le programme : ces variables ne peuvent apparaître que dans les gardes et les membres droits des affectations du programme.

La partie *always* n'est là que pour permettre la définition de raccourcis syntaxiques. Elle n'est pas nécessaire en ce sens que tout programme UNITY possède un équivalent sans partie *always* : il suffit de remplacer récursivement toutes les variables transparentes avec leur définition en terme de variables non transparentes[1]. C'est la raison pour laquelle, si cette partie est effectivement utile à l'écriture de programmes UNITY, elle n'est par contre jamais considérée dans les études sémantiques du langage. Dans les considérations formelles des paragraphes suivants, nous ne considérerons que des programmes sans variables transparentes.

Assign :

Cette partie définit le cœur du programme, c'est-à-dire les différentes instructions qui le composent. Pour un programme F, la notation "s `in` F" signifie que s est une instruction du programme F, c'est-à-dire une des instructions de sa partie *assign*. Le langage UNITY possède une unique forme d'instruction : l'affectation vectorielle gardée. Un programme UNITY est composé d'un nombre *fini* et *non nul* d'affectations séparées par le symbole $[\!]$.

Chaque affectation est de la forme :

$$\begin{array}{rcll} v_1, v_2, \cdots, v_n & := & e_1^1, e_2^1, \cdots, e_n^1 & \texttt{if}\ g^1 \\ & \sim & e_1^2, e_2^2, \cdots, e_n^2 & \texttt{if}\ g^2 \\ & \sim & \cdots & \texttt{if}\ \cdots \\ & \sim & e_1^p, e_2^p, \cdots, e_n^p & \texttt{if}\ g^p \end{array}$$

1. Il s'agit d'une équivalence *sémantique*. La construction de cet équivalent est valable pour toutes les sémantiques présentées dans les paragraphes suivants.

Elle signifie que les variables[2] $v_1, v_2 \cdots, v_n$ se voient affecter *simultanément* et *atomiquement* les valeurs $e_1^j, e_2^j, \cdots, e_n^j$ si l'expression g^j est évaluée à *true*.

Une instruction UNITY est *toujours* exécutable : si aucune garde g^j n'est vraie, l'instruction est sans effet. Nous noterons *skip* ce type d'instruction qui ne modifie pas l'état du système. Une différence majeure entre les affectations UNITY et les commandes gardées de Dijkstra réside dans le fait que les instructions UNITY sont tenues d'être *déterministes*. En particulier, si plusieurs gardes g^j sont vraies, les valeurs affectées aux variables doivent être les mêmes pour tous ces indices j. Formellement, on doit avoir, pour toute instruction UNITY de la forme précédente :

$$\langle \forall j, k : 1 \leqslant j, k \leqslant p :: g^j \wedge g^k \Rightarrow \langle \forall i : 1 \leqslant i \leqslant n :: e_i^j = e_i^k \rangle \rangle$$ [3]

Par exemple :

$$\begin{array}{lrl} x \ := & y & \texttt{if} \ \ y \geqslant 0 \\ \sim & -y & \texttt{if} \ \ y \leqslant 0 \end{array}$$

est une instruction valide, qui affecte à la variable x la valeur absolue de y. En effet, on a bien :

$$y \leqslant 0 \wedge y \geqslant 0 \Rightarrow y = -y$$

Par contre, l'instruction :

$$\begin{array}{lrl} x \ := & 1 & \texttt{if} \ \ y \geqslant 0 \\ \sim & -1 & \texttt{if} \ \ y \leqslant 0 \end{array}$$

n'est pas valide puisque son comportement n'est pas déterminé dans le cas où $y = 0$.

Une autre notation est possible pour les instructions UNITY. Elle utilise le symbole $\|$ pour séparer les différentes parties d'une même instruction. La notation $x := a \parallel y := b$ est une autre forme pour $x, y := a, b$.

On peut exprimer une instruction comportant plusieurs cas séparés par le symbole $\sim$ à l'aide de l'opérateur $\|$. En effet, l'instruction[4] :

$$\begin{array}{lrl} \bar{v} \ := & \bar{e} & \texttt{if} \ \ g \\ \sim & \bar{e}' & \texttt{if} \ \ g' \end{array}$$

peut s'écrire :

$$\bar{v} := \bar{e} \ \texttt{if} \ g \parallel \bar{v} := \bar{e}' \ \texttt{if} \ g'$$

Outre son aspect esthétique, qui permet de partitionner proprement une instruction complexe, et son intérêt évident dans la définition de la *superposition* de programmes (cf § 2.1.6), l'opérateur $\|$ présente l'avantage sur le symbole $\sim$

2. v_i désigne en fait une *référence* qui peut être une variable (de n'importe quel type) ou un élément de tableau de la forme $tab[expression]$.

3. La formule doit être un théorème de la logique des prédicats. On ne peut pas se contenter d'imposer qu'elle soit un *invariant* du programme étant donné que le programme lui-même n'est défini que dans la mesure où l'instruction est valide.

4. $\bar{v}$ représente le vecteur de références $v_1, v_2, \cdots, v_n$. Idem les expressions $\bar{e}$ et $\bar{e}'$.

de pouvoir être *quantifié*, ce qui permet de définir une instruction UNITY sans en énumérer explicitement tous les cas. Plus précisément, l'instruction :

$$\bar{v}_1 := \bar{e}_1 \texttt{ if } g_1 \parallel \bar{v}_2 := \bar{e}_2 \texttt{ if } g_2 \parallel \cdots \parallel \bar{v}_p := \bar{e}_p \texttt{ if } g_p$$

peut s'écrire :

$$\langle \parallel i : 1 \leqslant i \leqslant p :: \bar{v}_i := \bar{e}_i \texttt{ if } g_i \rangle$$

Cette notation se généralise de la façon suivante :

$$\langle \parallel i : P.i :: Stmt(i) \rangle$$

où P est un prédicat et $Stmt(i)$ une instruction UNITY (éventuellement elle-même quantifiée) paramétrée par i. On impose néanmoins que le prédicat P soit constant (c'est-à-dire ne dépende pas des variables du programme) et qu'il ne soit vrai que pour un nombre *fini* de valeurs de l'indice i. Si P n'est pas satisfiable, l'instruction est équivalente à *skip*. Il ne faut pas perdre de vue que, aussi compliquée soit-elle, la notation $\langle \parallel i : 1 \leqslant i \leqslant p :: \bar{v}_i := \bar{e}_i \texttt{ if } g_i \rangle$ désigne *une* instruction UNITY, synchrone et atomique.

Les différentes instructions de la partie *assign* sont séparées par le symbole $[\!]$. Là encore, une définition quantifiée peut être utilisée à la place d'une énumération explicite des instructions. L'ensemble d'instructions :

$$\bar{v}_1 := \bar{e}_1 \texttt{ if } g_1 \; [\!] \; \bar{v}_2 := \bar{e}_2 \texttt{ if } g_2 \; [\!] \; \cdots \; [\!] \; \bar{v}_p := \bar{e}_p \texttt{ if } g_p$$

peut s'écrire :

$$\langle [\!] i : i \leqslant i \leqslant p :: \bar{v}_i := \bar{e}_i \texttt{ if } g_i \rangle$$

et la notation se généralise en :

$$\langle [\!] i : P.i :: Stmts(i) \rangle$$

où P est un prédicat et $Stmts(i)$ représente une ou plusieurs instructions. De la même manière que précédemment, le prédicat P est constant et ne doit être vrai que pour un nombre fini de valeurs de l'indice i de façon à ce qu'un programme ait bien un nombre *fini* d'instructions. Si le prédicat P n'est pas satisfiable, le groupe d'instructions est équivalent à une instruction *skip*.

2.1.3 Interprétation opérationnelle

L'ensemble des instructions d'un programme sont implicitement contenues dans une boucle sans fin. À chaque itération, une instruction est choisie arbitrairement puis exécutée de façon atomique. Le choix de l'instruction exécutée doit être équitable, c'est-à-dire que chaque instruction doit être exécutée une infinité de fois au cours d'une exécution. Comme nous le verrons, cette équité du choix de l'instruction correspond à une hypothèse d'équité faible (telle qu'elle a été définie au paragraphe 1.2.3) sur *toutes* les transitions du système de transitions sous-jacent au programme UNITY tel qu'il est défini au paragraphe 2.1.8.

Cette exécution infinie d'instructions choisies arbitrairement correspond en fait à une absence de flot de contrôle pour le programme. Ceci constitue une des originalités du modèle UNITY et lui permet de faire abstraction de l'architecture d'exécution. Par exemple, les instructions peuvent être exécutées séquentiellement de la première à la dernière (dans l'ordre du texte du programme) en recommençant ce même choix indéfiniment, ce qui correspond en quelque sorte à une exécution séquentielle du programme. À l'opposé, on peut considérer que chaque instruction est exécutée par un processeur différent et à des cadences différentes, ce qui correspond à une exécution à fort degré de parallélisme. Tous les choix "intermédiaires" sont également possibles, pourvu que la contrainte d'équité soit respectée.

Lorsqu'on a besoin d'un séquencement précis des instructions [5], il faut le réaliser explicitement à l'aide de variables supplémentaires. Par exemple, le bloc séquentiel :

$$\bar{v}_1 := \bar{e}_1 \texttt{ if } g_1 \ ; \ \bar{v}_2 := \bar{e}_2 \texttt{ if } g_2 \ ; \ \cdots \ ; \ \bar{v}_p := \bar{e}_p \texttt{ if } g_p$$

peut s'écrire :

$$\langle [\!] i : 1 \leqslant i \leqslant p :: \langle \bar{v}_i := \bar{e}_i \parallel pc := (pc \ mod \ p) + 1 \rangle \texttt{ if } g_i \wedge pc = i \rangle$$

où pc est une variable initialisée avec la valeur 1 et qui n'est pas affectée ailleurs dans le programme. On obtient alors la structure de bloc classique des langages concurrents qui acceptent le séquencement d'instructions : le bloc n'est pas atomique et les exécutions du programme font intervenir un entrelacement de ses instructions avec les autres instructions.

2.1.4 La logique UNITY

Dans sa définition première, la logique UNITY s'appuie sur la logique des triplets de Hoare. Ces triplets expriment que, si un prédicat (appelé *pré-condition*) est valide *avant* l'exécution d'une instruction, et si l'exécution de cette instruction se termine, alors un autre prédicat (appelé *post-condition*) est valide *après* l'exécution de l'instruction. Les triplets de Hoare sont généralement notés $\{$*pré-condition*$\}$ *instruction* $\{$*post-condition*$\}$.

La différence majeure entre l'utilisation de la logique de Hoare dans la définition de la logique UNITY par rapport à son utilisation plus classique dans les programmes séquentiels est due à l'absence de flot de contrôle d'un programme UNITY. En effet, il devient impossible de préciser *l'emplacement* dans le programme des pré-conditions et des post-conditions. De la même façon, l'exécution des programmes ne se terminant pas, il est impossible de raisonner sur une pré-condition initiale et une post-condition finale, comme cela peut se faire sur un programme transformationnel.

C'est pourquoi les triplets considérés dans la définition de la logique UNITY ne s'appuient pas sur instruction particulière mais expriment une propriété de *l'ensemble* des instructions du programme. Ils sont donc de deux formes, selon que la quantification des instructions est universelle ou existentielle :

$$\langle \forall s :: \{p\}\, s\, \{q\} \rangle$$
$$\langle \exists s :: \{p\}\, s\, \{q\} \rangle$$

5. Ce besoin d'un séquencement précis s'est avéré relativement rare dans les types d'applications que nous avons considérés.

Les différents opérateurs de la logique Unity (*unless*, *stable*, *constant*, *invariant*, *ensures*, *leads-to*, *until* et *detects*) sont définis à partir de ces deux types d'écriture de la façon suivante :

Unless :

$$p \;\texttt{unless}\; q \;\texttt{in}\; F^{6} \quad \equiv \quad \langle \forall s : s \;\texttt{in}\; F :: \{p \wedge \neg q\}\, s\, \{p \vee q\} \rangle$$

Intuitivement, p `unless` q `in` F signifie que, si le prédicat p devient vrai à un instant de l'exécution du programme F, ce prédicat restera vrai au moins jusqu'à ce que le prédicat q devienne vrai.

Il convient de remarquer que :

- le prédicat p n'est tenu à rien si, au moment où il devient vrai, le prédicat q est déjà vrai. En effet, on a alors le triplet $\{false\}\, s\, \{p \vee q\}$ qui est trivialement vrai pour toute instruction s ;
- le prédicat p n'est pas tenu de rester vrai au moment où le prédicat q devient vrai. En particulier, les prédicats p et q peuvent ne jamais être vrais simultanément. Par exemple, p `unless` $\neg p$ est une tautologie de la logique Unity.

L'opérateur *unless* constitue la brique élémentaire de la logique permettant de construire les autres opérateurs :

Stable :

$$\texttt{stable}\; p \;\texttt{in}\; F \quad \equiv \quad p \;\texttt{unless}\; \mathit{false} \;\texttt{in}\; F$$

Ceci signifie que si le prédicat p devient vrai, il doit le rester jusqu'à ce que le prédicat *false* devienne vrai. Un prédicat *stable*, s'il est vrai à un instant de l'exécution du programme, le reste continûment et indéfiniment par la suite.

Néanmoins, le prédicat p n'est pas tenu de devenir vrai. En particulier, `stable` *false* est une tautologie de la logique Unity.

Constant :

$$\texttt{constant}\; p \;\texttt{in}\; F \quad \equiv \quad \texttt{stable}\; p \;\texttt{in}\; F \;\wedge^{7}\; \texttt{stable}\; \neg p \;\texttt{in}\; F$$

Si le prédicat p est vrai, il doit rester vrai, et s'il est faux, il doit rester faux. Un prédicat *constant* ne change donc pas de valeur de vérité au cours de l'exécution du programme.

6. Lorsqu'il n'y a pas d'ambiguïté sur le programme considéré, nous omettons la partie " `in` F" d'une propriété Unity.

7. Une propriété de la logique Unity, appliquée à un (ou deux) prédicats et à un programme, est une valeur booléenne sur laquelle nous utiliserons les connecteurs de la logique propositionnelle.

Invariant :

$$\texttt{invariant}\ p\ \texttt{in}\ F \quad \equiv \quad \text{condition initiale} \Rightarrow p \ \wedge\ \texttt{stable}\ p\ \texttt{in}\ F$$

Un prédicat *invariant* est vrai au commencement de l'exécution du programme et, étant *stable*, doit le rester par la suite. Il est donc vrai tout au long de l'exécution du programme.

Ensures :

$$p\ \texttt{ensures}\ q\ \texttt{in}\ F \quad \equiv \quad p\ \texttt{unless}\ q\ \texttt{in}\ F\ \wedge\ \langle \exists s : s\ \texttt{in}\ F :: \{p \wedge \neg q\}\ s\ \{q\}\rangle$$

La propriété p $\texttt{ensures}$ q $\texttt{in}$ F signifie que si le prédicat p devient vrai, il doit le rester jusqu'à ce que le prédicat q devienne vrai (partie *unless*) et qu'en plus il existe une instruction du programme qui permet de rendre vrai le prédicat q à partir du prédicat p. Étant donné que le choix de l'instruction exécutée est équitable, on est assuré que, si le prédicat p devient vrai, l'instruction en question sera exécutée *après* que p est devenu vrai et donc on est certain que le prédicat q deviendra finalement vrai.

Leads-to :

La propriété *leads-to* est notée $\mapsto$. Intuitivement, $p \mapsto q$ $\texttt{in}$ F signifie que, si le prédicat p est vrai à un instant donné de l'exécution du programme F, le prédicat q deviendra vrai à un instant postérieur.

Cette propriété est définie de manière axiomatique à partir de l'opérateur *ensures*. Les propriétés *leads-to* valides sont celles qui se déduisent des trois axiomes suivants :

$$\frac{p\ \texttt{ensures}\ q\ \texttt{in}\ F}{p \mapsto q\ \texttt{in}\ F}$$

$$\frac{p \mapsto q\ \texttt{in}\ F\ \wedge\ q \mapsto r\ \texttt{in}\ F}{p \mapsto r\ \texttt{in}\ F}$$

$$\text{Pour tout ensemble } W : \frac{\langle \forall m : m \in W :: p(m) \mapsto q\ \texttt{in}\ F\rangle}{\langle \exists m : m \in W :: p(m)\rangle \mapsto q\ \texttt{in}\ F}$$

Les deux premiers axiomes expriment que la fermeture transitive de relations *ensures* valides constitue une propriété de type *leads-to* valide. Le dernier axiome, appelé *axiome de la disjonction infinie*, exprime que, si l'on peut déduire la validité à venir du prédicat q à partir de la validité de tout prédicat $p(m)$, alors on peut également la déduire à partir de la validité d'un prédicat $p(m)$ quelconque non déterminé. La propriété *leads-to*, définie à partir de ces trois axiomes, est parfois appelée fermeture transitive et disjonctive de l'opérateur *ensures*.

On peut remarquer que :

– *tout* instant où le prédicat p est vrai doit être suivi d'un instant où le prédicat q est vrai. Ceci signifie que la propriété *leads-to* se prolonge dans le temps : il ne suffit pas que le *premier* instant de l'exécution pour lequel le prédicat p est vrai soit suivi d'un instant où le prédicat q est vrai. Par contre, un instant où le prédicat q est vrai satisfait au *leads-to* pour *tous* les instants qui le précèdent pour lesquels le prédicat p est vrai. En particulier, si le prédicat q devient vrai et le reste à partir d'un certain moment, le propriété $p \mapsto q$ est valide pour tous les prédicats p.

- si le prédicat q est déjà vrai au moment où le prédicat p devient vrai, la propriété $p \mapsto q$ est valide (il ne s'agit pas d'un avenir strict).

- le prédicat p n'est pas tenu de rester vrai jusqu'à ce que le prédicat q devienne vrai pour que la propriété $p \mapsto q$ soit valide. Cette contrainte supplémentaire est décrite par l'opérateur *until* défini ci-après :

Until :

$$p \texttt{ until } q \texttt{ in } F \;\equiv\; p \texttt{ unless } q \texttt{ in } F \;\wedge\; p \mapsto q \texttt{ in } F$$

Cette propriété exprime que si le prédicat p devient vrai, alors il reste vrai jusqu'à ce que le prédicat q devienne vrai (partie *unless*) et que le prédicat q deviendra finalement vrai (partie *leads-to*)[8]. On peut remarquer que, si la propriété p `ensures` q `in` F est valide, alors la propriété p `until` q `in` F l'est aussi.

Detects :

$$p \texttt{ detects } q \texttt{ in } F \;\equiv\; \texttt{invariant } p \Rightarrow q \texttt{ in } F \;\wedge\; q \mapsto p \texttt{ in } F$$

Cette propriété exprime que le prédicat p peut être utilisé pour détecter la véracité du prédicat q. En effet, si le prédicat p est vrai, le prédicat q l'est aussi, et si le prédicat q est vrai, le prédicat p le deviendra.

La propriété *detects* est très utile pour exprimer la détection de propriétés globales et stables d'un système (comme l'interblocage ou la terminaison) à partir d'une propriété locale. On a alors une relation de la forme :

Condition locale `detects` État global

La partie *invariant* du *detects* garantit contre les fausses détections, et la partie *leads-to* assure que la détection sera finalement faite.

8. Cet opérateur n'exprime pas tout-à-fait la même chose que l'opérateur until de la logique temporelle linéaire tel qu'il est décrit par exemple dans [MP92]. En particulier, p `until` q `in` F en UNITY n'empêche pas le prédicat p d'être continuellement faux lors de l'exécution du programme F.

Le prédicat de point-fixe

En plus des opérateurs logiques dont nous venons de rappeler la définition, la logique UNITY définit un prédicat d'état particulier, noté *FP*, et appelé *prédicat de point-fixe*.

Un point-fixe d'un programme UNITY est un état à partir duquel l'exécution de n'importe quelle instruction du programme laisse cet état inchangé. Si le programme décrit un système fermé, cet état de point-fixe correspond à la terminaison du programme.

Le prédicat *FP* est défini de la façon suivante. Une instruction UNITY peut toujours s'écrire sous la forme :

$$s \;=\; \langle \| i : c_i :: \bar{v}_i := \bar{e}_i \;\; \texttt{if} \;\; p_i \rangle$$

On définit alors :

$$\text{fp}.s \quad \equiv \quad \langle \forall i : c_i \wedge p_i :: \bar{v}_i = \bar{e}_i \rangle$$

Le prédicat *FP* est défini de la façon suivante :

$$\text{FP}.F \quad \equiv \quad \langle \forall s : s \;\texttt{in}\; F :: \text{fp}.s \rangle$$

La propriété fondamentale de ce prédicat est la suivante :

$$\sigma_i \text{ état de point-fixe de } F \quad \equiv \quad \sigma_i.(\text{FP}.F)$$

Par contre, il faut bien noter que le prédicat *FP* ne caractérise pas les points-fixes *accessibles* : il existe des états satisfaisant *FP* qui ne peuvent pas être atteints par le programme. Considérons, par exemple, le programme suivant :

```
Program pf
Initially
   p = true  ‖ x = 0
Assign
   x:=1 if p
‖
   p:=false
End pf
```

On a :

$$\begin{aligned} \text{FP}.pf \;&\equiv\; (p \Rightarrow x = 1) \wedge (p \equiv \mathit{false}) \\ &\equiv\; \neg p \end{aligned}$$

Mais, par exemple, $FP_1 \equiv \neg p \wedge x = 2$ caractérise un état de point-fixe de pf non accessible. $FP_2 \equiv \neg p \wedge x = 1$ caractérise un état de point-fixe accessible, mais pas tous. Enfin, $FP_3 \equiv \neg p \wedge (x = 0 \vee x = 1)$ caractérise l'ensemble des états de point-fixe accessibles. Dans tous les cas, les états satisfaisant FP_1 ou FP_2 ou FP_3 sont bien des états de point-fixe. On a bien $\langle \forall i : 1 \leqslant i \leqslant 3 :: FP_i \Rightarrow \text{FP}.pf \rangle$.

Comme nous nous intéressons dans la suite uniquement à des programmes réactifs qui ne se terminent pas (dont le prédicat *FP* est *false* et qui n'ont donc pas d'état de point-fixe accessible), nous n'utiliserons pas le prédicat *FP*.

Remarques à propos de l'opérateur ensures :

- La relation *ensures* est une propriété du *texte* du programme. Elle n'est pas à proprement parler une propriété au sens du chapitre 1, c'est-à-dire une caractéristique d'une partie de l'ensemble des exécutions possibles du programme. Considérons, par exemple, les deux programmes suivants :

```
Program F                    Program G
Assign                       Assign
   x:=0 if x > 0                x:=0
[]                           []
   x:=0 if x < 0                skip
End F                        End G
```

  Le programme G vérifie la propriété $x \neq 0$ `ensures` $x = 0$ alors que le programme F ne la vérifie pas. En effet, il n'y a pas dans le programme F d'instruction s telle que $\{x \neq 0\}\, s\, \{x = 0\}$. Néanmoins, ces deux programmes vérifient la propriété $x \neq 0$ `until` $x = 0$. Nous verrons au paragraphe 2.1.9 que les programmes F et G ne peuvent pas être différentiés en terme d'exécutions et qu'ils ont donc les mêmes propriétés (au sens du chapitre 1).

- L'axiome de la disjonction infinie :

$$\text{Pour tout ensemble } W : \frac{\langle \forall m : m \in W :: p(m) \ \mapsto\ q \texttt{ in } F \rangle}{\langle \exists m : m \in W :: p(m) \rangle \ \mapsto\ q \texttt{ in } F}$$

  n'est nécessaire que dans le cas où l'ensemble W est infini. En effet, les propriétés de l'opérateur *ensures* et les deux premiers axiomes du *leads-to* (fermeture transitive) suffisent à montrer la propriété de disjonction finie :

$$\frac{p_1 \ \mapsto\ q \texttt{ in } F \ \wedge\ p_2 \ \mapsto\ q \texttt{ in } F \ \wedge\ \cdots\ \wedge\ p_n \ \mapsto\ q \texttt{ in } F}{p_1 \vee p_2 \vee \cdots \vee p_n \ \mapsto\ q \texttt{ in } F}$$

  Par contre, l'axiome de la disjonction infinie est nécessaire aux preuves des *leads-to* les plus élémentaires. Par exemple, le programme :

```
Declare   x : Int;
Assign    x := x - 1
```

  vérifie la propriété *true* $\mapsto\ x < 0$. Mais si la relation *leads-to* n'était *que* la fermeture transitive de l'opérateur *ensures*, cette propriété ne serait pas démontrable.

2.1.5 Le système de preuve

Outre le langage de programmation et le langage de spécification précédemment décrits, le formalisme Unity propose un système de preuve permettant de démontrer la cohérence entre la description opérationnelle et la description logique d'un programme.

Ce système repose sur un ensemble de théorèmes parmi lesquels on peut distinguer :

- les théorèmes de la logique appliqués à un programme ;

– les théorèmes de composition de programmes ;

– l'axiome de substitution.

Un théorème de la logique appliqué à un programme exprime que si ce programme vérifie les hypothèses du théorème, il en vérifie la conclusion. Par exemple, un théorème extrêmement utile, appelé PSP (*Progress-Safety-Progress*), est :

$$\frac{p \;\mapsto\; q, \quad r \;\; \texttt{unless} \;\; b}{p \wedge r \;\mapsto\; (q \wedge r) \vee b}$$

Nous n'allons pas nous livrer ici à une énumération des théorèmes de la logique qui se trouvent dans les références citées. Par contre, nous renvoyons au chapitre 5 pour une utilisation détaillée de ces théorèmes au sein de la preuve effective d'un algorithme.

À la différence des précédents, les théorèmes de composition de programmes font intervenir plusieurs programmes et garantissent des propriétés du programme composé à partir des propriétés des programmes composants. Ils sont donc liés aux opérateurs de composition de programmes présentés au paragraphe 2.1.6.

Enfin, l'axiome de substitution s'inspire de la règle de Leibniz en logique qui exprime que l'on peut substituer dans une formule logique une partie par une formule équivalente sans changer la valeur de vérité de la formule. Dans le cas de la logique Unity, l'axiome de substitution affirme que l'on peut remplacer une grandeur X par une grandeur Y pourvu que $X = Y$ soit un invariant du programme.

Cet axiome est malheureusement incohérent avec le reste de la logique Unity : la logique Unity avec l'axiome de substitution est inconsistante. Par contre, la logique est incomplète sans cet axiome dans la mesure où la preuve de la plupart des propriétés nécessite l'utilisation de la substitution (voir l'exemple du paragraphe 2.1.7).

Les difficultés liées à l'utilisation de l'axiome de substitution sont à présent bien comprises et ont été contournées de plusieurs manières. Nous discutons plus longuement de ce point au paragraphe 2.2.1. D'ici là, la théorie sera supposée ne pas contenir l'axiome de substitution.

Un des points forts du système de preuve Unity réside dans la façon de démontrer les propriétés de vivacité (*leads-to*). En effet, l'opérateur *leads-to* est défini de manière inductive à partir de l'opérateur *ensures*. Cet opérateur *ensures*, compte tenu de l'équité faible du modèle, permet d'établir des propriétés élémentaires de vivacité sans utiliser de raisonnement par récurrence. Les propriétés plus complexes (*leads-to*) sont ensuite obtenues par récurrence sur la structure de la preuve. En aucun cas, on est amené à raisonner par récurrence sur les séquences d'exécution comme c'est le cas pour les opérateurs opérationnels définis au paragraphe 2.1.9. Cette approche est d'une grande force, en particulier pour la preuve des systèmes asynchrones.

2.1.6 Composition de programmes

Le langage UNITY offre deux opérateurs de composition qui permettent de structurer les programmes. Les deux formes de composition sont *l'union* et *la superposition.*

L'union de programmes est symbolisée par l'opérateur $[\!]$ appliqué cette fois à des programmes. Cet opérateur a pour fonction de réunir les différentes variables et instructions des programmes composés. Bien sûr, une telle composition n'est possible que dans la mesure où les programmes composés offrent une certaine forme de compatibilité. En particulier, ils ne doivent pas déclarer la même variable avec des types différents, ni initialiser une même variable avec des valeurs différentes.

L'opérateur d'union est commutatif et associatif. Il peut donc facilement être quantifié. La syntaxe UNITY est la suivante :

$$\langle [\!] i : 1 \leqslant i \leqslant N :: P_i \rangle$$

pour représenter l'union des N programmes P_i :

$$P_1 \;[\!]\; P_2 \;[\!]\; \cdots \;[\!]\; P_N$$

Cette forme de composition est notamment utilisée :

- pour décrire les programmes de manière modulaire ;
- pour représenter les différents processus d'un programme concurrent.

Nous verrons au chapitre 7 que des critiques peuvent être formulées à l'encontre de l'opérateur d'union dans son rôle de composition de processus et nous présenterons alors notre proposition pour remédier à ce problème.

La composition par superposition a un tout autre rôle. Elle consiste à enrichir un programme au niveau de chacune de ses instructions en y ajoutant de manière synchrone de nouvelles affectations. Ces nouvelles affectations ne doivent pas perturber l'ancien comportement du programme. C'est pourquoi on impose qu'elles ne modifient pas les anciennes variables du programme. La composition par superposition admet également l'ajout (asynchrone) de nouvelles instructions avec les mêmes restrictions. Plus précisément, chaque instruction t du programme est transformée en $t \| s$ où s est une instruction ne modifiant pas les anciennes variables du programme. Éventuellement, sont également ajoutées des instructions $s_1 [\!] s_2 [\!] \cdots [\!] s_n$ avec les mêmes restrictions.

La propriété fondamentale de cette forme de composition est qu'elle préserve *toutes* les propriétés du programme (exprimées en terme des anciennes variables). En d'autres termes, elle constitue un raffinement de programmes (cf chap. 4).

Cette forme de transformation est utile :

- pour concevoir un programme en couches : chaque couche ne fait que consulter les variables des couches inférieures, sans les modifier ;
- pour introduire des variables auxiliaires : ce type de variable permet de décrire plus précisément l'état d'un système et donc de faciliter la preuve, tout en ne modifiant pas le comportement initial.

2.1.7 Exemple de programme et de spécification

Le programme suivant décrit le comportement de N processus vis-à-vis de l'acquisition d'une ressource critique. Les N processus sont numérotés de 0 à $N-1$ et sont répartis autour d'un anneau. Le successeur sur l'anneau du nœud i est donc le nœud $(i+1)\ mod\ N$.

Le processus numéro i demande l'accès à la ressource en positionnant à *true* la variable $Dem[i]$. Il accède à la ressource, lorsqu'il y est autorisé, en positionnant la variable $Excl[i]$ à *true* et libère la ressource en repositionnant ces deux variables à *false*. La concurrence d'accès est gérée par un jeton circulant sur l'anneau. La variable $Jeton[i]$ est vraie si et seulement si le jeton se situe au niveau du site i. Ce site est alors autorisé à accéder à la ressource. À la libération de la ressource, il donne le jeton, et donc le droit d'accès, à son successeur sur l'anneau. Un site fait également passer le jeton s'il n'est pas demandeur de la ressource.

Le comportement d'un site est modélisé par le programme suivant :

```
Program Site(i)
Declare
  N : Int;
  Dem, Excl, Jeton : Array 0..N − 1 of Bool;
Initially
  ⟨|| i : 0 ⩽ i ⩽ N − 1 :: Dem[i] = false || Excl[i] = false⟩ []
  ⟨|| i : 1 ⩽ i ⩽ N − 1 :: Jeton[i] = false⟩ || Jeton[0] = true
Assign
  { dépot de demande }
  Dem[i] := true
[]
  { entrée en zone critique }
  Excl[i] := true if Dem[i] ∧ Jeton[i]
[]
  { sortie de zone critique }
  Excl[i], Dem[i], Jeton[i], Jeton[(i + 1) mod N] := false, false, false, true if Excl[i]
[]
  { circulation du jeton }
  Jeton[i], Jeton[(i + 1) mod N] := false, true if Jeton[i] ∧ ¬Dem[i]
End Site
```

La première instruction simule la demande d'accès à la ressource. La deuxième instruction représente l'accès effectif à cette ressource pourvu que le site soit demandeur et qu'il possède le jeton. La troisième instruction décrit la libération de la ressource et (en même temps) le passage du jeton au site suivant. Enfin, la dernière instruction représente le passage du jeton au site suivant si le site possédant le jeton n'est pas demandeur.

On utilise ensuite la composition par union pour représenter le système global :

$$Mutex = \langle [\!]\, i : 0 \leqslant i \leqslant N-1 :: Site(i) \rangle$$

La logique Unity permet d'exprimer les propriétés essentielles de sûreté et de vivacité de ce programme. Par exemple, la propriété d'exclusion mutuelle se traduit par l'invariant suivant :

$$\texttt{invariant}\ \langle \forall i,j : 0 \leqslant i,j \leqslant N-1 :: Excl[i] \wedge Excl[j] \Rightarrow i=j \rangle\ \texttt{in}\ Mutex$$

Cet invariant exprime qu'à tout instant de toute exécution du programme, deux processus distincts ne peuvent avoir accès à la ressource critique. Une telle propriété (de sûreté) est en général insuffisante pour décrire le bon comportement d'un programme. En effet, le programme qui refuserait toujours et à tout processus l'accès à la ressource critique vérifierait également cette propriété.

C'est pourquoi, il est souvent nécessaire d'ajouter à la description d'un programme des propriétés de vivacité. Dans le cas du programme $Mutex$, l'équité (faible) d'accès à la ressource est garantie par la propriété (de vivacité) suivante :

$$\forall i : 0 \leqslant i \leqslant N-1 :: Dem[i] \mapsto Excl[i] \texttt{ in } Mutex$$

Cette propriété exprime que tout processus demandeur finira par accéder à la ressource.

La preuve de ces propriétés nécessite l'application de l'axiome de substitution. Strictement parlant, les propriétés ne sont donc pas vraies dans la théorie UNITY telle que nous venons de la présenter. Par contre, elles sont vraies pour les opérateurs $invariant_{\mathcal{O}}$, $leads\text{-}to_{\mathcal{O}}$, $invariant_{\mathrm{S}}$ et $leads\text{-}to_{\mathrm{S}}$ qui sont définis aux paragraphes 2.1.9 et 2.2.2.

2.1.8 Interprétation en terme de système de transitions

Nous avons défini au chapitre 1 la notion de système de transitions ainsi que différentes notions d'équité qui peuvent lui être associées. Nous allons voir qu'un programme UNITY représente en fait un système de transitions dans lequel chaque instruction correspond à une transition et une seule et où toutes les transitions sont faiblement équitables.

En effet, nous pouvons construire, à partir de la description d'un programme UNITY, le quadruplet $(\mathcal{V}, \Sigma, \mathcal{T}, \phi)$ défini précédemment :

– l'ensemble des variables d'état $\mathcal{V}$ est l'ensemble des variables définies dans la section *declare* ;

– l'ensemble des états Σ est l'ensemble des fonctions définies sur $\mathcal{V}$ qui associent à chaque variable une valeur de son type ;

– pour construire l'ensemble des transitions $\mathcal{T}$ correspondant à un programme UNITY, on construit une transition pour chaque instruction de la façon suivante : soit une instruction s de la forme :

$$\begin{array}{rcl} \bar{v} & := & \bar{e}^1 \texttt{ if } g^1 \\ & \sim & \bar{e}^2 \texttt{ if } g^2 \\ & & \cdots \\ & \sim & \bar{e}^p \texttt{ if } g^p \end{array}$$

On lui associe la transition τ définie comme suit :

$$\begin{array}{l} \langle \forall \sigma_i, \sigma_j : \sigma_i, \sigma_j \in \Sigma :: \\ \quad \sigma_j \in \tau.\sigma_i \;\equiv\; \langle \exists k : 1 \leqslant k \leqslant p :: \sigma_i.g^k \wedge \sigma_j = \sigma_i|^{\bar{v}}_{\bar{e}^k} \rangle \vee \\ \qquad\qquad\qquad\quad \langle \forall k : 1 \leqslant k \leqslant p :: \neg g^k \rangle \wedge \sigma_j = \sigma_i \\ \rangle \end{array}$$

où $\sigma_i|^{\bar{v}}_{\bar{e}^k}$ représente l'état obtenu à partir de σ_i en substituant chaque élément du vecteur $\bar{v}$ par l'élément correspondant du vecteur $\bar{e}^k$. Cette expression définit la fonction τ de manière non ambiguë.

On voit immédiatement que l'ensemble $\tau.\sigma_i$ n'est jamais vide. En effet, si σ_i vérifie l'une des gardes g^k alors $\sigma_i|^{\bar{v}}_{\bar{e}^k} \in \tau.\sigma_i$ et sinon, $\sigma_i \in \tau.\sigma_i$. Ceci correspond au fait qu'une instruction UNITY est exécutable dans tout état (absence de flot de contrôle). De plus, on a vu que les instructions d'un programme UNITY sont déterministes, c'est-à-dire vérifient la contrainte $\langle \forall i, j : 1 \leqslant i, j \leqslant p :: g^i \wedge g^j \Rightarrow \bar{e}^i = \bar{e}^j \rangle$. On déduit de cette contrainte que l'ensemble $\tau.\sigma_i$ ne peut contenir plus d'un seul élément. En conséquence, pour tout état σ_i, l'ensemble $\tau.\sigma_i$ est un singleton et on utilisera la notation fonctionnelle $s.\sigma_i$ pour représenter l'élément de ce singleton.

– la condition initiale ϕ est obtenue en remplaçant chaque affectation de la section *initially* par l'égalité logique correspondante et en considérant la conjonction de ces égalités. Plus précisément, si la section *initially* est de la forme :

$$\langle [\![i : 1 \leqslant i \leqslant N :: \langle \| j : 1 \leqslant j \leqslant M :: \bar{v}_i^j = \bar{e}_i^j \rangle \rangle$$

on lui associe le prédicat :

$$\phi \equiv \langle \wedge i : 1 \leqslant i \leqslant N :: \langle \wedge j : 1 \leqslant j \leqslant M :: \bar{v}_i^j = \bar{e}_i^j \rangle \rangle$$

où le symbole = représente à présent l'égalité logique et non l'affectation.

On voit qu'un ensemble d'affectations initiales peut toujours être traduit en un prédicat équivalent. Par contre, la transformation inverse n'est pas toujours possible, même pour des prédicats très simples. Par exemple, il est impossible de préciser que l'on souhaite retenir comme états initiaux ceux où les variables x et y sont égales, ou ceux où la variable z est non nulle.

C'est pourquoi, l'extension de UNITY qui consiste à considérer la section *initially* comme un *prédicat* portant sur les variables d'état est communément admise. On impose alors seulement, comme c'est le cas pour les systèmes de transitions, que le prédicat soit satisfiable. Dorénavant, nous admettrons cette petite extension au formalisme de base et, pour un programme F, nous noterons ce prédicat $F.init$.

Nous voyons donc qu'un programme UNITY peut se traduire en un système de transitions équivalent. Dans tous les cas, le système obtenu est à branchement fini puisque chaque transition associe à chaque état un nombre fini de successeurs (en l'occurrence, un seul). La question de l'équité se pose donc pour de tels systèmes.

On a vu que l'interprétation opérationnelle d'un programme UNITY supposait que chaque instruction était exécutée infiniment souvent. En fait cette contrainte correspond à une hypothèse d'équité faible sur chacune des transitions du système équivalent. En effet, on a vu que chaque transition correspondait à une affectation et que ces transitions étaient constamment exécutables

($\tau.\sigma_i$ n'est jamais vide). L'hypothèse d'équité faible, qui interdit à une transition d'être indéfiniment continûment exécutable sans être exécutée, oblige donc chaque transition (et donc chaque instruction) à être exécutée une infinité de fois au cours d'une exécution. Réciproquement, si chaque instruction est exécutée infiniment souvent, la transition correspondante l'est aussi et vérifie donc nécessairement la contrainte d'équité faible.

2.1.9 Sémantique opérationnelle des opérateurs logiques

Dès lors qu'un programme UNITY est considéré comme la description textuelle d'un système de transitions équitable, on peut définir précisément l'ensemble de ses exécutions possibles en terme de séquences d'états.

Une sémantique opérationnelle, notée $\mathcal{O}_1$, est donc définie. Pour un programme F, $\mathcal{O}_1.F$ est l'ensemble des séquences d'états σ telles que :

- $\sigma_0.(F.init)$;
- $\langle \forall i : i \geqslant 0 :: \langle \exists s : s \texttt{ in } F :: \sigma_{i+1} = s.\sigma_i \rangle\rangle$;
- $\langle \forall s : s \texttt{ in } F :: \langle \forall k : k \geqslant 0 :: \langle \exists i : i \geqslant k :: \sigma_{i+1} = s.\sigma_i \rangle\rangle\rangle$.

Le premier point exprime que l'état initial d'une séquence de $\mathcal{O}_1.F$ vérifie les contraintes imposées par la section *initially* du programme. Le second point impose que chaque transition d'états est due à l'exécution d'une instruction du programme. Enfin, le dernier point traduit l'équité du modèle UNITY : chaque instruction est exécutée une infinité de fois.

Cette sémantique opérationnelle de UNITY correspond à l'expression la plus simple du comportement d'un programme. Elle devra être modifiée pour prendre en compte certaines extensions du modèle comme celles présentées au paragraphe 2.3 ou comme notre système d'observation défini au chapitre 5.

À partir du moment où l'on a défini les exécutions possibles d'un programme UNITY, on peut chercher à caractériser ces séquences par des opérateurs de logique temporelle. À partir de l'interprétation intuitive des opérateurs de la logique UNITY, nous définissons ici les opérateurs logiques (opérationnels) $unless_{\mathcal{O}}$, $invariant_{\mathcal{O}}$ et $leads\text{-}to_{\mathcal{O}}$ ($\mapsto_{\mathcal{O}}$) à partir des opérateurs $unless_\sigma$, $invariant_\sigma$ et $leads\text{-}to_\sigma$ ($\mapsto_\sigma$) définis sur une séquence d'exécution.

Soient un programme UNITY F, une sémantique opérationnelle $\mathcal{O}$ et une séquence σ de $\mathcal{O}.F$. On définit, pour des prédicats d'état p et q :

- $p \texttt{ unless}_\sigma\ q \texttt{ in } \sigma \equiv \langle \forall i : i \geqslant 0 :: \sigma_i.(p \wedge \neg q) \Rightarrow \sigma_{i+1}.(p \vee q)\rangle$ [9]
- $\texttt{invariant}_\sigma\ p \texttt{ in } \sigma \equiv \langle \forall i : i \geqslant 0 :: \sigma_i.p\rangle$ [10]
- $p \mapsto_\sigma q \texttt{ in } \sigma \equiv \langle \forall i : i \geqslant 0 :: \sigma_i.p \Rightarrow \langle \exists j : j \geqslant i :: \sigma_j.q\rangle\rangle$

9. On a naturellement : $\texttt{stable}_\sigma\ p \texttt{ in } \sigma \equiv \langle \forall i :: \sigma_i.p \Rightarrow \sigma_{i+1}.p\rangle$
$\equiv \langle \forall i :: \sigma_i.p \Rightarrow \langle \forall j : j \geqslant i :: \sigma_j.p\rangle\rangle$
$\texttt{invariant}_\sigma\ p \texttt{ in } \sigma \equiv \sigma_0.p \wedge \texttt{stable}_\sigma\ p \texttt{ in } \sigma$

10. On peut lui préférer la définition équivalente, plus conforme aux habitudes de logique temporelle linéaire :

$$\langle \forall i : i \geqslant 0 :: \sigma_i.p \Rightarrow (\langle \forall j : j \geqslant i :: \sigma_j.p\rangle \vee \langle \exists j : j \geqslant i :: \sigma_j.q \wedge \langle \forall k : i \leqslant k \leqslant j :: \sigma_k.p\rangle\rangle)$$

Chacun des opérateurs $unless_{\mathcal{O}}$, $invariant_{\mathcal{O}}$ et $leads\text{-}to_{\mathcal{O}}$ est défini à partir de l'opérateur $unless_{\sigma}$, $invariant_{\sigma}$ et $leads\text{-}to_{\sigma}$ correspondant :

- p $\texttt{unless}_{\mathcal{O}}$ q $\texttt{in}$ $F \equiv \langle \forall \sigma : \sigma \in \mathcal{O}.F :: p \ \texttt{unless}_{\sigma} \ q \ \texttt{in} \ \sigma \rangle$
- $\texttt{invariant}_{\mathcal{O}}$ p $\texttt{in}$ $F \equiv \langle \forall \sigma : \sigma \in \mathcal{O}.F :: \texttt{invariant}_{\sigma} \ q \ \texttt{in} \ \sigma \rangle$
- $p \mapsto_{\mathcal{O}} q$ $\texttt{in}$ $F \equiv \langle \forall \sigma : \sigma \in \mathcal{O}.F :: p \mapsto_{\sigma} q \ \texttt{in} \ \sigma \rangle$

Les autres opérateurs de la logique Unity (*stable*, *constant*, *until*, *detects*) sont définis à partir de ces trois opérateurs de base. Nous utiliserons ce type d'opérateurs dans la suite conjointement avec deux sémantiques opérationnelles différentes, $\mathcal{O}_1$ que nous avons définie dans ce paragraphe, et $\mathcal{O}bs$ qui sera définie au chapitre 5.

On peut remarquer que l'opérateur $leads\text{-}to_{\mathcal{O}}$ est défini directement en fonction des séquences d'exécution et non plus à partir de l'opérateur élémentaire *ensures*. En fait, il n'y a pas d'expression possible de l'opérateur *ensures* à partir des séquences d'état (non étiquetées). En effet, considérons les deux programmes F et G du paragraphe 2.1.4 :

Program F	**Program** G
Assign	**Assign**
x := 0 **if** x > 0	x := 0
▯	▯
x := 0 **if** x < 0	*skip*
End F	**End** G

On a vu que le programme G vérifiait la propriété $x \neq 0$ $\texttt{ensures}$ $x = 0$ et que le programme F ne la vérifiait pas. Or, dans l'interprétation opérationnelle $\mathcal{O}_1$, ces deux programmes ont *exactement* le même ensemble d'exécutions possibles. Il s'agit de toutes les séquences de la forme :

$$(x) : (x_0) \longrightarrow (x_0) \longrightarrow (x_0) \longrightarrow \cdots \longrightarrow (x_0) \longrightarrow (0) \longrightarrow (0) \longrightarrow \cdots$$

constituées d'un préfixe fini pour lequel la variable x conserve sa valeur initiale x_0 suivi d'un suffixe infini pour lequel la variable x est nulle. Ceci montre que l'opérateur *ensures* est étroitement lié au *texte* du programme et ne peut être défini au sein du modèle opérationnel $\mathcal{O}_1$.

Un autre point, plus important, qu'il convient de souligner est que les opérateurs d'origine définis au paragraphe 2.1.4 et les opérateurs opérationnels indicés par $\mathcal{O}_1$ ne coïncident pas. La différence entre ces deux classes d'opérateurs provient du fait que nous avons supprimé l'axiome de substitution pour éviter que la théorie soit contradictoire. Différentes possibilités existent pour réintroduire cet axiome ou le remplacer de façon à retrouver la cohérence entre les opérateurs définis de façon axiomatique et les opérateurs opérationnels. Cet aspect de la théorie Unity est étudié dans les paragraphes qui suivent.

2.2 L'apport de B.A. Sanders

Cette partie présente la modification de la théorie UNITY proposée par B.A. Sanders en 1991. Cette nouvelle axiomatisation de la théorie est née de la volonté de résoudre les contradictions liées à l'utilisation de l'axiome de substitution. Elle a permis de bien comprendre les liens existant entre les opérateurs définis par K.M. Chandy et J. Misra et leurs pendants opérationnels.

2.2.1 L'inconsistance introduite par l'axiome de substitution

Nous avons présenté au paragraphe 2.1.5 l'axiome de substitution qui permet, dans l'expression d'une propriété UNITY d'un programme, de remplacer une expression par une autre pourvu que l'égalité de ces deux expressions soit un invariant de ce programme. Le contre-exemple suivant montre que cette règle est contradictoire avec la définition des opérateurs de la logique.

Considérons le programme suivant :

```
Program Axiome
Declare
   p, q : Bool;
Initially
   p ∧ ¬q;
Assign
  p:=¬q
End Axiome
```

Tout d'abord, il est très facile de démontrer que ce programme vérifie la propriété `invariant` $\neg q$: la condition initiale implique $\neg q$ et le triplet $\{\neg q\}\, p := \neg q\, \{\neg q\}$ est valide.

Ce programme vérifie également la propriété p `unless` q. En effet :

$$\begin{array}{ll} & p \ \texttt{unless}\ q \\ \equiv & \langle \forall s : s \ \texttt{in}\ Axiome :: \{p \wedge \neg q\}\, s\, \{p \vee q\}\rangle \\ \equiv & \{p \wedge \neg q\}\, p := \neg q\, \{p \vee q\} \\ \equiv & \langle \forall p, q :: p \wedge \neg q \ \Rightarrow\ wp(p := \neg q, p \vee q)\rangle \\ \equiv & \langle \forall p, q :: p \wedge \neg q \ \Rightarrow\ \neg q \vee q\rangle \\ \equiv & \langle \forall p, q :: p \wedge \neg q \ \Rightarrow\ true\rangle \\ \equiv & true \end{array}$$

Compte tenu de l'axiome de substitution qui permet de remplacer le prédicat q par *false* dans la propriété p `unless` q, le programme devrait vérifier la propriété p `unless` *false*. Or, ce n'est pas le cas. En effet :

$$\begin{array}{ll} & p \ \texttt{unless}\ false \\ \equiv & \langle \forall s : s \ \texttt{in}\ Axiome :: \{p\}\, s\, \{p\}\rangle \\ \equiv & \{p\}\, p := \neg q\, \{p\} \\ \equiv & \langle \forall p, q :: p \ \Rightarrow\ wp(p := \neg q, p)\rangle \\ \equiv & \langle \forall p, q :: p \ \Rightarrow\ \neg q\rangle \\ \equiv & false \end{array}$$

Ces deux façons différentes de calculer la valeur de vérité de la propriété p `unless` *false* conduisent à une contradiction.

La dernière étape de la preuve montre que la propriété p `unless` *false* n'est pas vérifiée à cause de l'état (p, q) : $(true, true)$. En effet, partant d'un état vérifiant $p \wedge q$, l'instruction $p := \neg q$ conduit à $\neg p$ et donc invalide la propriété `stable` p, c'est-à-dire p `unless` *false*. Pourtant, cet état vérifiant $p \wedge q$ n'apparaît dans aucune exécution du programme *Axiome* puisque ce programme vérifie la propriété `invariant` $\neg q$.

On voit donc que la difficulté provient du fait que la définition des opérateurs UNITY conduit à considérer des états inaccessibles du programme. Dans une approche opérationnelle où l'on considère explicitement les séquences d'états, la difficulté disparaît. En effet, le programme *Axiome* vérifie les trois propriétés :

$$p \ \texttt{unless}_{\mathcal{O}_1}\ q, \qquad \texttt{invariant}_{\mathcal{O}_1}\ \neg q, \qquad p\ \texttt{unless}_{\mathcal{O}_1}\ \mathit{false}$$

En ce qui concerne ces opérateurs opérationnels, l'axiome de substitution n'induit aucune contradiction, Il constitue même en fait un théorème aisément démontrable.

On peut montrer qu'une propriété établie de manière axiomatique sera vérifiée sur l'ensemble des exécutions $\mathcal{O}_1$, mais nous voyons ici que la réciproque n'est pas vraie. Notamment, une propriété peut être toujours vraie dans tous les états de toutes les exécutions d'un programme et ne pas être un invariant de ce programme. C'est en ce sens que les opérateurs d'origine et les opérateurs opérationnels diffèrent.

La figure 2.1 illustre le lien existant entre les invariants intuitifs (opérationnels) et ceux de la logique de K.M. Chandy et J. Misra sans l'axiome de substitution (inductifs). Le prédicat P qui est décrit par la grande zone grisée

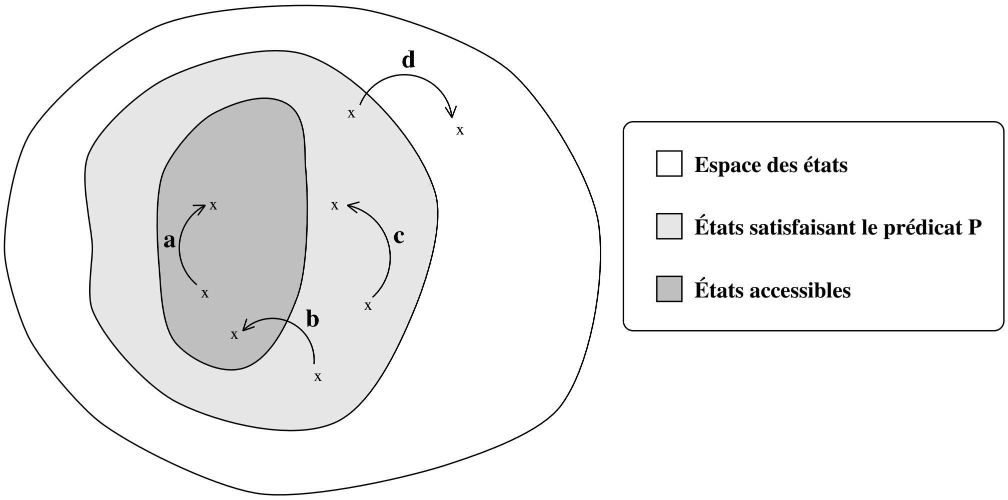

FIG. 2.1 – *Invariants opérationnels et invariants inductifs.*

est un invariant opérationnel. En effet, puisque l'ensemble des états accessibles du programme est inclus dans cette zone, le prédicat P est nécessairement vérifié par tout état de toute exécution de ce programme.

Par contre, le prédicat P n'est pas nécessairement un invariant inductif. En effet, d'après la définition du paragraphe 2.1.4, il faut pour cela que toute transition effectuée depuis un état satisfaisant P conduise à un état satisfaisant P. S'il existe dans le programme une transition de type **d**, ce n'est pas le cas, même si pour falsifier P la transition **d** doit être exécutée depuis un état inaccessible.

2.2.2 Notion de "plus fort invariant" d'un programme

Le contre-exemple présenté au paragraphe précédent montre la distinction importante qu'il convient de faire entre les états accessibles et les états inaccessibles d'un programme. Le problème qui se pose est donc de ne pas considérer les états inaccessibles dans la preuve des propriétés d'un programme. C'est là le rôle du "plus fort invariant".

Pour un programme donné F, les contraintes d'invariance (être vrai dans les états initiaux et être maintenu par toutes les transitions) constituent une équation en terme de prédicats décrite par une fonction monotone (au sens de l'implication). On peut alors montrer que cette équation admet une solution maximale baptisée *plus fort invariant* du programme F et notée $F.SI$ (pour "*strongest invariant*").

Ce prédicat peut être vu comme la caractérisation des états accessibles d'un programme : un état est accessible au cours d'une exécution du programme F si et seulement si il satisfait le prédicat $F.SI$. On peut donc toujours supposer, au cours d'une preuve, que le plus fort invariant est vrai, car s'il ne l'était pas, cela signifierait que l'on considère un état inaccessible par le programme. C'est cette idée que l'on retrouve dans la nouvelle définition des opérateurs UNITY proposée par B.A. Sanders. Nous noterons ces opérateurs avec un indice s et ils sont définis de la façon suivante[11] :

- ◇ $p\ \texttt{unless}_{\mathrm{S}}\ q\ \texttt{in}\ F \quad \equiv p \wedge F.SI\ \texttt{unless}\ q\ \texttt{in}\ F$
- ◇ $\texttt{invariant}_{\mathrm{S}}\ p\ \texttt{in}\ F \equiv (F.init \Rightarrow p) \wedge p\ \texttt{unless}_{\mathrm{S}}\ \mathit{false}\ \texttt{in}\ F$
- ◇ $p\ \texttt{ensures}_{\mathrm{S}}\ q\ \texttt{in}\ F \equiv p\ \texttt{unless}_{\mathrm{S}}\ q\ \texttt{in}\ F\ \wedge$ $\langle \exists s : s \in F :: \{p \wedge \neg q \wedge F.SI\}\, s\, \{q\} \rangle$

Les autres opérateurs (*stable*$_{\mathrm{S}}$, *constant*$_{\mathrm{S}}$, *leads-to*$_{\mathrm{S}}$ ($\mapsto_{\mathrm{S}}$), *until*$_{\mathrm{S}}$ et *detects*$_{\mathrm{S}}$) sont définis à partir de ces trois opérateurs de manière analogue à la définition des opérateurs d'origine.

On peut remarquer que l'on a l'équivalence :

$$\texttt{invariant}_{\mathrm{S}}\ p\ \texttt{in}\ F\ \equiv\ (F.SI \Rightarrow p)$$

et donc que cette nouvelle définition de l'invariance correspond bien à la notion de propriétés toujours vraies dans toutes les exécutions d'un programme. Plus généralement, on peut montrer que les opérateurs indicés par s (sauf *ensures*) coïncident avec ceux définis sur la sémantique opérationnelle $\mathcal{O}_1$, c'est-à-dire, par exemple :

$$p\ \mapsto_{\mathrm{S}}\ q\ \texttt{in}\ F\ \equiv\ p\ \mapsto_{\mathcal{O}_1}\ q\ \texttt{in}\ F$$

11. En fait, il ne s'agit pas exactement de la définition proposée par B.A. Sanders mais plutôt de théorèmes qui en découlent.

On montre également que ces opérateurs vérifient tous les théorèmes de la théorie initiale dans le cas de propriétés d'un seul programme (c'est-à-dire les théorèmes autres que les théorèmes de composition de programmes) et ce, sans besoin de l'axiome de substitution. Par contre, les théorèmes de composition de programmes ne sont malheureusement pas vérifiés comme nous allons le voir dans le paragraphe suivant.

2.2.3 Les points faibles de la théorie

La redéfinition des opérateurs de la logique Unity que nous venons de présenter fournit une définition axiomatique cohérente et relativement complète en ce sens que les propriétés définies coïncident avec les propriétés correspondantes sur les séquences d'états. Plus exactement, la complétude relative de cette axiomatisation de Unity signifie que, si une formule est vraie dans la sémantique opérationnelle d'un programme, alors il existe une axiomatisation (finie et consistante) de la logique sous-jacente (en l'occurrence la logique des prédicats du premier ordre) telle que la formule soit démontrable. Néanmoins, on ne retrouve pas, pour cette définition des opérateurs, les propriétés de composition de programmes de la théorie initiale. En particulier, un prédicat peut être invariant à la fois dans F et dans G et ne pas l'être dans $F[\!]G$. C'est le cas, par exemple, du prédicat $x = 0$ dans les deux programmes suivants :

Program F	**Program** G
Declare	**Declare**
x : Int;	x : Int;
b : Bool;	b : Bool;
Initially	**Initially**
$(x = 0) \wedge \neg b$;	$x = 0$;
Assign	**Assign**
x:=1 **if** b	b:=true
End F	**End** G

En effet, les états pour lesquels $b \equiv \mathit{true}$ sont inaccessibles dans le programme F, ce qui permet de démontrer $\texttt{invariant}_{\mathrm{S}}\ x = 0\ \texttt{in}\ F$. Mais ces états sont accessibles dans $F[\!]G$ ce qui permet à la variable x de prendre la valeur 1 (on a en fait $\mathit{true} \mapsto x = 1\ \texttt{in}\ F[\!]G$). En fait, on constate que le rôle particulier joué par les états accessibles empêche l'émergence de propriétés de composition utiles. En effet, d'une manière générale, on ne peut pas calculer le prédicat $F[\!]G.SI$ à partir des prédicats $F.SI$ et $G.SI$.

Un autre point qu'il convient de noter est que, si l'existence du plus fort invariant fournit une base solide pour la définition des opérateurs, ce prédicat n'est en général pas calculable explicitement pour un programme donné. En particulier, dans le contexte d'une assistance mécanique à la preuve, il est difficile de vérifier une propriété en s'appuyant directement sur la définition de B.A. Sanders. On utilise donc généralement le résultat suivant :

$$\frac{\texttt{invariant}\ I \wedge p}{\texttt{invariant}_{\mathrm{S}}\ p}$$

(et des règles similaires pour les autres opérateurs de la logique).

La difficulté est de trouver un prédicat I tel que $I \wedge p$ soit un invariant au sens de la théorie initiale (sans axiome de substitution), même si l'on sait qu'un tel invariant existe nécessairement si la propriété $\texttt{invariant}_S\ p$ est valide (le plus fort invariant du programme convient). Le rôle de ce prédicat I est de restreindre l'ensemble des états considérés de façon à approcher suffisamment l'ensemble exact des états accessibles. Les conséquences pratiques d'une telle technique seront détaillées au chapitre 3.

Pour ces deux raisons que sont la volonté de conserver des propriétés de composition et de pouvoir mécaniser des étapes de preuve, on est généralement amené à considérer à la fois, au sein d'un raisonnement, les propriétés dans leur définition d'origine (sans l'axiome de substitution) et dans leur définition à partir du plus fort invariant. Dans le premier cas, nous parlerons de propriétés "*fortes*" ou *inductives* et nous les noterons sans indice. Dans le second cas, nous parlerons de propriétés "*faibles*" et nous les noterons indicées par s. Nous utiliserons également l'expression "*propriété toujours vraie*" pour désigner un invariant faible. Enfin, le vocabulaire des propriétés faibles sera également utilisé pour les opérateurs définis à partir de la sémantique opérationnelle $\mathcal{O}_1$, puisque ces opérateurs coïncident avec ceux de B.A. Sanders.

2.3 Autres extensions et modifications du formalisme

Nous présentons ici brièvement trois extensions et redéfinitions du formalisme UNITY. Ces extensions visent, par exemple, à réintroduire une forme d'axiome de substitution, mais s'attachent surtout à trouver une formulation de la logique qui permette l'émergence de propriétés de composition utiles.

2.3.1 Les modifications apportées par J. Misra

En 1994, J. Misra, l'un des concepteurs du formalisme UNITY, a présenté une nouvelle définition de la théorie s'appuyant sur des opérateurs logiques légèrement différents de ceux de la théorie initiale. Il a également proposé une interprétation cohérente de l'axiome de substitution et un enrichissement du modèle facilitant la composition de programmes. Nous résumons dans ce paragraphe ces trois aspects.

Les opérateurs co et transient

Les opérateurs *co* et *transient* constituent une alternative aux opérateurs *unless* et *ensures* pour la définition des propriétés de la logique UNITY. Ils sont définis de la façon suivante :

- $p\ \texttt{co}\ q\ \texttt{in}\ F \equiv \langle \forall s : s\ \texttt{in}\ F :: \{p\}\, s\, \{q\} \rangle$
- $\texttt{transient}\ p\ \texttt{in}\ F \equiv \langle \exists s : s\ \texttt{in}\ F :: \{p\}\, s\, \{\neg p\} \rangle$

L'opérateur *co* devient l'expression de base des propriétés de sûreté. On définit :

- $\texttt{stable}\ p\ \texttt{in}\ F \equiv p\ \texttt{co}\ p\ \texttt{in}\ F$
- $\texttt{invariant}\ p\ \texttt{in}\ F \equiv F.init \Rightarrow p\ \wedge\ \texttt{stable}\ p\ \texttt{in}\ F$
- $\texttt{constant}\ p\ \texttt{in}\ F \equiv \texttt{stable}\ p\ \texttt{in}\ F\ \wedge\ \texttt{stable}\ \neg p\ \texttt{in}\ F$

On peut remarquer que, pour que la propriété $p \texttt{ co } q$ soit vérifiée par un programme F, il faut que $p \Rightarrow q$ soit un théorème de la logique. En effet, la présence de l'instruction *skip* dans F engendre la contrainte $\{p\}\, skip\, \{q\}$, c'est-à-dire $p \Rightarrow q$.

La relation $p \texttt{ co } q$ peut donc s'interpréter de la façon suivante : si à un instant donné le prédicat p est vrai (et donc le prédicat q également), et si lors d'une transition ultérieure p devient faux, le prédicat q sera maintenu vrai par cette transition.

L'opérateur *co* joue un rôle similaire à celui de l'opérateur *unless* dans la théorie initiale. Ces deux opérateurs ont d'ailleurs la même expressivité logique. En effet, on a :

- ⋄ $p \texttt{ co } q \equiv (p \texttt{ unless } \neg p \wedge q) \;\wedge\; (p \Rightarrow q)$
- ⋄ $p \texttt{ unless } q \equiv p \wedge \neg q \texttt{ co } p \vee q$

L'intérêt de l'opérateur *co* réside dans la formulation plus simple qu'il permet des théorèmes. Par exemple, la conjonction et la disjonction s'expriment plus simplement que pour l'opérateur *unless* :

$$\frac{p \texttt{ co } q, \quad p' \texttt{ co } q'}{\begin{array}{c} p \wedge p' \texttt{ co } q \wedge q' \\ p \vee p' \texttt{ co } q \vee q' \end{array}} \qquad \frac{p \texttt{ unless } q, \quad p' \texttt{ unless } q'}{\begin{array}{l} p \wedge p' \texttt{ unless } \;\;(p \wedge q') \vee (p' \wedge q) \vee (q \wedge q') \\ p \vee p' \texttt{ unless } (\neg p \wedge q') \vee (\neg p' \wedge q) \vee (q \wedge q') \end{array}}$$

Dans cette reformulation de la logique Unity, la propriété *transient* représente la forme élémentaire des propriétés de vivacité. Intuitivement, cette forme de propriété exprime le "contraire" de la stabilité : alors qu'un prédicat *stable* ne peut jamais devenir faux, un prédicat *transient* est sûr d'être falsifié (étant donnée l'équité faible du modèle). L'opérateur *transient* permet de définir l'opérateur *ensures*, et donc l'opérateur *leads-to* :

$$p \texttt{ ensures } q \;\equiv\; (p \wedge \neg q \texttt{ co } p \vee q) \wedge (\texttt{transient } p \wedge \neg q)$$

L'opérateur *leads-to* est alors strictement identique à ce qu'il était dans la théorie initiale (fermeture transitive et disjonctive de l'opérateur *ensures*), et la structure des preuves de vivacité ne change pas considérablement.

Le problème de l'axiome de substitution

Dès 1990, J. Misra proposait une nouvelle interprétation de l'axiome de substitution qui évite le problème de l'inconsistance. Elle consiste à considérer qu'il n'y a plus d'équivalence logique au niveau de la définition des opérateurs Unity en terme de triplets de Hoare, mais seulement une implication. Par exemple, l'opérateur *unless* est défini à partir de l'axiome de substitution et de la règle d'inférence suivante :

$$\frac{\langle \forall s : s \texttt{ in } F :: \{p \wedge \neg q\}\, s\, \{p \vee q\}\rangle}{p \texttt{ unless } q \texttt{ in } F}$$

On voit immédiatement que cette nouvelle définition, en interdisant de déduire un triplet à partir d'une propriété Unity, invalide le contre-exemple présenté au paragraphe 2.2.1 qui établissait une contradiction au sein de la théorie. En fait, J. Misra montre que le système obtenu est consistant et que les théorèmes de la théorie (exceptés ceux de composition de programmes) sont conservés.

Le principe de la démonstration consiste à montrer que toute preuve d'une propriété UNITY peut être traduite en une preuve équivalente qui ne fait pas intervenir l'axiome de substitution excepté dans la dernière étape. En effet, il suffit pour cela de conserver explicitement dans la preuve tous les invariants nécessaires, puis de les substituer par *true* dans une dernière étape. Par exemple, pour une propriété à prouver de la forme p `unless` q, on va en fait démontrer

$$p \wedge I_1 \wedge I_2 \wedge \cdots \wedge I_n \texttt{ unless } q$$

sans utiliser l'axiome de substitution, puis substituer *true* à l'invariant $I_1 \wedge I_2 \wedge \cdots \wedge I_n$. Ces preuves, où l'axiome de substitution n'est utilisé que lors de la dernière étape sont dites "*preuves en forme normale*".

À partir du moment où une preuve ne fait pas intervenir l'axiome de substitution, la déduction des propriété *unless* et *ensures* à partir des triplets de Hoare devient la seule règle d'inférence utilisable, et on retrouve donc l'équivalence logique de la définition d'origine. Les théorèmes sont donc applicables sous leur ancienne forme. En appliquant ces théorèmes aux résultats obtenus à l'avant-dernière étape d'une preuve en forme normale et en appliquant l'axiome de substitution à la conclusion de ces théorèmes, on montre la validité, dans ce nouveau système, des anciens théorèmes de la logique.

On peut remarquer des similitudes entre ce raisonnement et l'approche choisie par B.A. Sanders pour sa redéfinition de la logique. En effet, là où B.A. Sanders ajoutait le plus fort invariant d'un programme, J. Misra se contente de la conjonction de tous les invariants nécessaires au cours d'une preuve donnée. La notion de plus fort invariant d'un programme permet par contre de montrer facilement l'existence de preuves en forme normale. On peut considérer que la redéfinition de la logique UNITY proposée par J. Misra et celle proposée par B.A. Sanders constituent deux variations sur la même idée, l'une accordant la priorité à l'écriture effective de preuves et l'autre préférant formaliser précisément les fondements théoriques.

Dès lors, il n'est pas difficile de montrer que la théorie obtenue par J. Misra est consistante, ni surprenant de constater qu'il rencontre des difficultés au niveau de la composition de programmes. En effet, pour conserver des propriétés de composition, on doit soit ne considérer que des preuves qui n'utilisent pas l'axiome de substitution, soit n'utiliser l'axiome de substitution qu'avec des invariants *du programme composé*. Dans le premier cas, cela oblige à manipuler les propriétés conjointement avec leur preuve, c'est-à-dire à mémoriser, en plus d'une propriété, la manière dont elle a été établie. Dans le second cas, l'utilisation d'invariants vérifiés par le programme composé et non plus par les composantes réduit considérablement l'aspect compositionnel des théorèmes.

Composition de programmes

J. Misra a étudié une autre voie intéressante permettant d'obtenir des propriétés de composition de programmes. Son idée consiste à redéfinir l'union de programmes de façon à limiter les actions possibles de l'environnement d'un programme. En effet, comme nous l'avons vu dans les paragraphes précédents, il y a peu d'espoir de garantir des propriétés intéressantes d'une union de programmes si chaque composante peut agir librement sur l'état du système.

J. Misra propose donc une généralisation de la distinction habituellement faite entre variables locales et variables globales qui permet de préciser le degré de liberté de l'environnement d'un programme vis-à-vis de ses variables. En effet, toutes les variables d'un programme Unity sont a priori globales (c'est-à-dire visibles et modifiables en toute liberté par l'environnement) et définir une notion de variable locale (c'est-à-dire entièrement cachée à l'environnement) ne serait pas suffisant.

On définit donc un nouveau système de typage qui associe au type de base d'une variable (Int, Bool, ...) l'ensemble des opérations possibles sur cette variable. Pour un type de base donné, l'inclusion entre les ensembles d'opérations possibles constitue constitue une relation de sous-typage qui induit une structure de treillis : le type le plus petit est celui où l'ensemble des opérations possibles est vide, le plus grand est celui où cet ensemble est plein. On retrouve dans ces deux extrêmes les notions de variable locale (ensemble d'opérations possibles vide) et de variable globale (ensemble des opérations possibles plein).

Chaque variable est à présent déclarée dans un programme avec *deux* types (s'appuyant sur le même type de base) : un type vis-à-vis de ce programme (type *interne*) et un type vis-à-vis de l'environnement du programme (type *externe*). Ceci permet de préciser quelles sont les opérations possibles sur cette variable, à la fois pour le programme et pour son environnement.

Cette forme de déclaration de variables engendre des contraintes au niveau de la composition de programmes par union. En effet, l'union n'est possible que dans la mesure où les déclarations des variables sont compatibles. Plus précisément, il est nécessaire que chaque variable ait une déclaration interne ne lui autorisant que des opérations permises par les déclarations externes de cette variable dans les autres programmes de la composition, c'est-à-dire que son type interne soit un sous-type de ses types externes dans les autres modules. Cette contrainte est décidable et calculable automatiquement.

Compte tenu de cette contrainte, certaines unions seront valides et d'autres pas, ce qui va permettre de limiter les preuves de correction de propriétés au seules unions compatibles avec les déclarations de type. C'est ainsi que sont définies les "propriétés de fermeture" [12] : une propriété de fermeture d'un programme F est une propriété valide dans le programme $F \rrbracket G$ pour tous les programmes G tels que l'union soit définie. Ces propriétés de fermeture vérifient tous les théorèmes classiques de la logique Unity (à l'exception des théorèmes de composition de programmes) et permettent de reformuler de manière agréable les théorèmes de composition.

En particulier, on a vu que les théorèmes de composition restreignaient l'emploi de l'axiome de substitution aux invariants du programme composé. Plus précisément, on montre que l'axiome de substitution est utilisable à partir du moment où l'invariant utilisé est un invariant de fermeture. En effet, cet

12. "Closure properties" en anglais.

invariant ne peut être invalidé par l'environnement du programme et sera donc bien vérifié par le programme composé.

En conclusion, ces notions de type et de propriétés de fermeture n'apportent pas de théorèmes de composition fondamentalement nouveaux, mais permettent, de façon pratique, de raisonner sur les unions de programmes.

2.3.2 Le système de preuve de P. Collette

Dans sa thèse présentée en 1994, P. Collette propose une méthode pour concevoir des systèmes de preuve possédant des propriétés de composition. À titre d'exemple, il applique cette méthode au formalisme Unity et obtient donc une nouvelle définition de la logique que nous résumons ici.

Sa méthode s'appuie sur un mécanisme d'hypothèses-conclusions. Ce mécanisme étend la notion de spécification de la façon suivante : la spécification d'un programme exprime les propriétés que ce programme vérifie dès que son environnement respecte certaines hypothèses.

Les propriété de la logique Unity sont donc étendues par une contrainte supposée vérifiée par l'environnement. Cette contrainte se compose d'un prédicat d'initialisation (semblable à celui de la partie *initially* d'un programme) et d'un prédicat binaire qui restreint les modifications possibles de l'état du système par l'environnement. P. Collette présente ce système à la fois sous l'angle opérationnel et sous l'angle axiomatique.

Opérationnellement parlant, les exécutions possibles d'un programme sont des séquences infinies d'états pour lesquelles chaque état est obtenu à partir du précédent, soit par l'exécution d'une instruction du programme, soit par une transition due à l'environnement. Les transitions d'états sont donc réparties en deux classes : les internes (dues aux instructions du programme) et les externes (dues à l'environnement).

Un opérateur Unity exprime alors une propriété d'une exécution complète[13] sous l'hypothèse que les transitions de l'environnement vérifient leur contrainte. Cette contrainte s'exprime sous la forme d'un prédicat binaire reflexif[14]. Par exemple, le programme :

```
Declare   x : Int;
Assign    x := x + 1
```

vérifie la propriété :

$$\forall k :: \ x = k \ \mapsto \ x > 2k \text{ à condition que } x' \geqslant x$$

où le prédicat binaire $x' \geqslant x$ exprime le fait que les transitions de l'environnement ne peuvent pas faire décroître la variable x.

Le système logique possède également une axiomatique fondée sur les transformeurs de prédicats. En particulier, comme c'était le cas dans la logique de

13. Sauf l'opérateur *guarantee*, introduit par P. Collette, qui ne contraint que les transitions internes, par symétrie avec la contrainte imposée à l'environnement telle qu'elle est décrite dans ce qui suit.

14. La reflexivité permet de prendre en compte le bégaiement portant sur les variables du programme.

B.A. Sanders, les contraintes d'invariance (être vrai dans l'état initial, être maintenu par les transitions du programme et être maintenu par les transitions de l'environnement) s'expriment sous la forme d'une fonction monotone, ce qui permet de définir le plus fort invariant d'un programme (à partir de la condition initiale, des instructions et du prédicat binaire). À partir de là, il devient possible d'axiomatiser les différentes propriétés de la logique UNITY en considérant des triplets (ou des transformeurs de prédicats) de la forme :

$$\{p \wedge F.SI\}\, s\, \{q\}$$

On retrouve dans cette approche l'idée de J. Misra présentée au paragraphe précédent, mais la liberté de l'environnement est ici exprimée de façon plus fine grâce au prédicat binaire. Par contre, la différenciation entre les compositions possibles et les compositions interdites n'est plus calculable automatiquement. L'utilisation de ce type de prédicat semble suffisante pour caractériser le comportement de l'environnement vis-à-vis de la *sûreté*. Des hypothèses de vivacité, utiles dans certains cas, ne sont malheureusement pas exprimables.

P. Collette fait remarquer que deux prédicats binaires particuliers permettent de relier son système logique à celui de K.M. Chandy et J. Misra et à celui de B.A. Sanders. Si le prédicat binaire vaut *true*, on retrouve l'axiomatique de K.M. Chandy et J. Misra. En effet, dans ce cas les transitions de l'environnement sont complètement libres et on ne peut plus caractériser les états accessibles du programme autrement que par *true* (c'est-à-dire que le plus fort invariant du programme est *true*). On retrouve donc une définition des opérateurs de la logique UNITY équivalente à celle de la définition d'origine. Par contre, si ce prédicat vaut *false*, on retrouve un système fermé dans lequel l'environnement ne peut pas modifier les variables du programme. Dans ce cas, le plus fort invariant du programme composé est égal au plus fort invariant du programme seul, c'est-à-dire le plus fort invariant tel qu'il est défini chez B.A. Sanders, et on retrouve donc bien une sémantique équivalente. Ceci montre que l'axiomatique de Sanders est cohérente avec la vision opérationnelle classique (système fermé, pas de transitions de l'environnement) mais que l'axiomatique de K.M. Chandy et J. Misra (sans l'axiome de substitution) ne l'est pas.

2.3.3 Le formalisme ImpUNITY

R.T. Udink a présenté en 1995 une thèse sur le raffinement de programme au sein du formalisme UNITY. Nous reviendrons au chapitre 4 sur les différentes notions de raffinement qu'il a introduites. En plus de cet aspect raffinement de programmes, R.T. Udink propose une extension du système UNITY baptisée ImpUNITY. Nous présentons ici brièvement les caractéristiques de ImpUNITY.

ImpUNITY propose trois extensions au modèle et à la logique UNITY qui visent toutes les trois à faciliter la composition de programmes. Ces extensions sont la caractérisation de *l'interférence* avec l'environnement, la distinction au sein d'un état entre ce qui est *observable* et ce qui ne l'est pas, et la notion *d'appel procédural*.

L'interférence

Cette extension du modèle est assez similaire à la proposition de P. Collette pour caractériser les environnements que nous avons présentée au paragraphe précédent. R.T. Udink introduit, au niveau du modèle cette fois, une nouvelle section décrivant les actions possibles de l'environnement sur les variables du programme. Cette section, qui joue le rôle du prédicat binaire utilisé par P. Collette, se présente sous la forme d'un transformeur de prédicat.

Dès lors, seules sont autorisées les instructions de l'environnement qui constituent *un raffinement* de ce transformeur (en ce sens que le transformeur, appliqué à un prédicat quelconque, doit impliquer logiquement la *wp* de chaque instruction de l'environnement, appliquée au même prédicat). On impose que l'instruction *skip* soit toujours un raffinement du transformeur et que ce transformeur soit de plus idempotent.

Les contraintes qu'impose le transformeur de prédicats permettent de définir une classe d'invariants d'un programme maintenus vrais lors de toute composition de ce programme avec un environnement compatible[15]. Dès lors, il devient possible de définir axiomatiquement une forme de propriétés de la logique Unity s'appuyant sur ces invariants et qui possède de bonnes propriétés de composition.

L'observabilité

Le transformeur de prédicats utilisé dans l'extension présentée précédemment permet de restreindre les actions permises à l'environnement sur l'une ou l'autre variable. Par exemple, on peut spécifier qu'une variable donnée ne peut pas être modifiée par l'environnement. Par contre, on ne peut pas interdire à l'environnement de *consulter* une variable du programme, ce qui correspondrait à la notion habituelle de variable locale.

R.T. Udink propose donc une seconde extension du modèle qui permet de préciser quelles sont les variables visibles par le programme et quelles sont les variables visibles par l'environnement. Cette notion s'apparente à la forme de typage proposée par J. Misra et présentée au paragraphe 2.3.1. Elle est utilisée ici pour définir de nouvelles équivalences entre programmes et de nouvelles formes de raffinement.

L'appel procédural

Enfin, le formalisme ImpUnity est étendu par une notion d'appel procédural. On y distingue les procédures exportées vers l'environnement, les procédures importées de l'environnement, et les procédures locales. Ces procédures représentent des blocs d'instructions : elles comportent un unique paramètre par valeur, et les appels procéduraux ne peuvent pas être imbriqués. Moyennant ces restrictions, R.T. Udink est capable d'intégrer cette notion de procédure dans ses règles de raffinement et dans sa logique.

15. Néanmoins, contrairement à P. Collette, R.T. Udink ne définit pas de plus fort invariant.

Pour cela, il étend la notion de respect d'un transformeur de prédicats utilisée dans sa définition de l'interférence aux instructions contenant des appels procéduraux et aux déclarations de procédures. Il définit également une notion *d'approximation* des procédures importées qui permet de spécifier certaines propriétés logiques d'un programme sans connaître explicitement les procédures fournies par l'environnement. Néanmoins, cette approximation reste très imprécise et appauvrit considérablement la logique.

2.4 Bibliographie

L'ouvrage de référence sur le formalisme Unity est le livre de K.M. Chandy et J. Misra [CM88], en particulier par la richesse des exemples d'utilisation. La plupart des formes habituelles de programmation (synchrone, asynchrone) et de communication (mémoire partagée, échange de messages, rendez-vous) y sont reprises pour traiter les problèmes classiques de synchronisation (exclusion mutuelle, ressources critiques, détection de terminaison, prise de clichés, consensus, ...) et ainsi illustrer le modèle de programmation, le modèle de spécification logique et le système de preuve.

Nous n'avons bien sûr pas repris ici ces nombreux exemples, mais nous nous sommes attachés à fournir une définition du langage et de la logique au moins aussi précise et parfois plus formelle que celle de ce livre.

L'interprétation du langage et de la logique en terme de système de transitions et de leurs propriétés n'a pas de source précise mais s'inspire à la fois de [MP92] et de [UK93]. La notion d'opérateurs appliqués à *une seule* séquence d'états (propriétés de type "in σ") est celle qui est utilisée dans [Gol92] pour confirmer mécaniquement le système de preuve de la logique Unity.

La proposition de B.A. Sanders est décrite dans [San91]. Néanmoins, le contre-exemple choisi pour montrer l'inconsistance de la logique n'est pas celui utilisé dans cet article mais un exemple plus ancien. Il date de 1988 et a été présenté par J. van de Snepscheut qui l'attribue à un de ses élèves, J.E. Jonker.

L'équivalence entre la définition de la logique à partir du plus fort invariant et la version opérationnelle des opérateurs a été montrée par [Pac92]. La méthode présentée pour démontrer la validité d'invariants sans expliciter le plus fort invariant est classique et se retrouve par exemple dans [Ma94, Sha93, MP92]. Dans [San91], une version des propriétés de la logique Unity, indicées par un invariant, était proposée comme généralisation de cette approche, mais certains résultats énoncés ont été invalidés par la suite [Pra94].

J. Misra a présenté une seconde version du formalisme Unity dans [Mis94c]. La redéfinition de la logique à partir des opérateurs *co* et *transient* est présentée dans [Mis94e] pour la sûreté et dans [Mis94d] pour la vivacité. La remarque de la page 25 à propos de l'axiome de disjonction infinie est tirée du chapitre [Mis94d]. L'utilisation limitée de l'axiome de substitution et la notion de preuve en forme normale sont introduites dans [Mis90d]. La définition des types internes et externes et des propriétés de fermeture est exposée dans le chapitre [Mis94b].

Les systèmes de preuve de P. Collette et de R.T. Udink sont décrits dans leurs thèses respectives, [Col94] et [Udi95]. On y trouve également, ainsi que dans [Mis90b, UK93], des considérations sur le rapport entre les sémantiques opérationnelles et les sémantiques axiomatiques, sur lesquelles nous reviendrons au chapitre 4.

Enfin, d'autres modélisations du formalisme Unity n'ont pas été décrites, comme celle de N. Brown et A. Mokkedem [BM95] qui proposent, dans le but d'une mécanisation, une axiomatique consistante et relativement complète de la logique, ou comme les thèses de J.R. Rao [Rao95] et E. Knapp [Kna90] qui proposent d'autres solutions pour rendre à la logique Unity sa consistance.

Chapitre 3

Assistance à la preuve de programmes réactifs

Les assistants de preuves existant actuellement sont nombreux et de nombreux autres sont en cours de développement. Néanmoins, ces différents systèmes sont loin d'offrir les mêmes possibilités. Ils reposent également sur des bases théoriques qui peuvent être fondamentalement différentes. Il semble donc important de chercher à les comparer, à la fois sur le plan de leurs fonctionnalités et sur le plan de leurs fondements théoriques.

Dans ce chapitre, nous proposons dans une première partie une ébauche de classification des systèmes d'assistance à la preuve de programmes. Cette classification s'appuie sur des critères concernant à la fois l'utilisation d'un système et sa conception.

Dans une seconde partie, nous présentons ensuite quelques systèmes existants en les positionnant vis-à-vis de la classification définie. Ces systèmes sont en nombre restreint, mais ont été choisis parce qu'ils présentaient des particularités qui nous ont semblé intéressantes. D'autres systèmes sont cités à la fin du chapitre.

3.1 Classification des assistants de preuve

Nous utiliserons quatre critères pour comparer les différents outils d'assistance à la preuve de programmes. Ces critères portent sur la manière d'utiliser des spécifications, le choix des formalismes utilisés, le type de raisonnement suivi et le degré général d'automatisation de l'outil.

3.1.1 Utilisation de spécifications

Le problème posé est celui de la *preuve formelle* de la correction d'un programme : il s'agit de prouver que le comportement effectif du programme correspond au comportement souhaité lors de sa conception. Ces deux comportements, différents a priori, doivent donc être exprimés, et ces descriptions, pour pouvoir être utilisées au sein d'une preuve, doivent être faites dans des langages formels.

Bien qu'il s'agisse dans les deux cas d'exprimer le comportement d'un programme, une distinction assez forte peut être faite entre deux types de langages : les langages de spécification (descriptive) pour le comportement souhaité, et les langages de programmation (ou langages de spécification constructive) pour le comportement effectif. En effet, même si sémantiquement il s'agit toujours de l'expression du comportement d'un programme, les objectifs des langages sont différents :

- en ce qui concerne la spécification, l'utilisateur doit pouvoir décider facilement si elle correspond ou non à son attente informelle. Elle doit donc être abstraite et extrêmement concise.

- le langage de programmation obéit lui à une contrainte d'un tout autre ordre : le programme décrit doit rester *exécutable* sous des hypothèses d'architecture et de système opératoire données.

On voit vite que ces deux types de langages répondent à des contraintes assez antagonistes. Plus formellement, une spécification représente un *ensemble* de modèles (ceux qui sont corrects vis-à-vis de cette spécification) alors qu'un programme constitue un modèle particulier.

À partir de là, deux types d'approches sont possibles :

- on ne considère aucune implantation particulière et on se limite à n'utiliser qu'un langage de spécification. On manipule alors différentes spécifications d'un même comportement, correspondant à des descriptions plus ou moins précises, généralement liées par une relation de raffinement (cf chap. 4).

  Comme on souhaite tout de même obtenir finalement un *programme* prouvé, il faut enrichir la spécification de manière à restreindre l'ensemble des modèles possibles jusqu'à pouvoir en extraire un de la spécification de manière systématique. Cette dérivation finale d'un modèle constitue une des difficultés majeures de cette approche.

  L'autre grande difficulté, à l'inverse, consiste à ne pas réduire l'ensemble des modèles possibles à l'ensemble vide, c'est-à-dire à ne pas introduire de contradictions logiques dans la spécification. En effet, une telle spécification possède toutes les propriétés logiques que l'on peut souhaiter, mais ne possède aucun modèle qui puisse correspondre à une implantation réaliste.

- on manipule *à la fois* une spécification du comportement désiré et une implantation particulière décrite dans un langage proche d'un langage de programmation. Outre le fait que la considération d'une implantation oblige souvent à des hypothèses très précises sur l'architecture supposée pour l'exécution, le problème majeur reste alors de *prouver* la correction de ce modèle vis-à-vis de la spécification.

Dans la suite, nous qualifierons la première approche d'approche par spécification *seule* et la seconde d'approche par spécification *et* modèle.

3.1.2 Choix d'un formalisme

Quelle que soit l'approche choisie parmi les deux précédentes se pose la question du choix d'un formalisme d'expression de la spécification et le cas échéant du modèle.

Cet aspect a déjà été discuté aux paragraphes 1.1.1 et 1.1.2. Dans ce chapitre, nous ne considérerons que le langage et la logique Unity ou des systèmes très proches fondés comme Unity sur les systèmes de transitions et la logique temporelle.

3.1.3 Raisonnement utilisé

Une autre distinction entre les systèmes de preuve de programmes parallèles, plus importante peut-être que le choix des formalismes, concerne le type de fonctionnement du système. On peut distinguer deux approches fondamentalement différentes selon que l'on s'appuie sur un *modèle* ou sur une *preuve*. Dans le premier cas, on vérifie explicitement qu'un programme constitue un modèle d'une spécification alors que dans l'autre cas, on se contente d'en apporter une preuve mathématique. C'est la même différence qu'il y a, en logique propositionnelle, entre exhiber la table de vérité d'une formule et démontrer cette formule à l'aide d'axiomes et de règles d'inférence.

Raisonnement par modèle

On trouve dans la première catégorie tous les outils reposant sur la synthèse de modèle (*model-construction*) ou sur la vérification de modèle (*model-checking*). Les systèmes de vérification de modèle sont répandus et leurs principaux avantages sont :

– leur complète automatisation : étant donné un programme et une spécification, le système est capable de valider ou d'invalider le programme sans aucune intervention humaine supplémentaire ;

– la richesse du résultat fourni : en cas d'échec de la vérification, le système est généralement capable de produire un contre-exemple à la formule soumise (par exemple en terme de transitions et d'états) ;

– leur efficacité en terme de performances : l'implantation de ces systèmes profite d'une longue expérience et les outils obtenus sont capables de vérifier des programmes en des temps raisonnables.

Face à ces avantages, il ne subsiste qu'une faiblesse majeure : fondés sur une exploration exhaustive des états, ces outils sont la plupart du temps limités à l'étude de systèmes dans lesquels le nombre d'états possibles est *fini*. En fait, l'explosion combinatoire est telle que de nombreux domaines, pourtant finis, restent hors de portée. Notamment, ces outils ne permettent pas de manipuler des variables entières dans toute leur généralité, que leur domaine soit considéré comme infini ou comme fini mais vaste. Or, ces variables apparaissent très fréquemment dans la description de systèmes réactifs, sous forme d'horloges ou de compteurs d'événements par exemple.

Raisonnement par preuve

Cette incapacité des méthodes de vérification de modèle à traiter les programmes à espace d'états infini conduit à explorer de plus en plus la seconde voie, celle de la vérification par preuve. À partir d'axiomes et de règles d'inférence liés à la sémantique du formalisme considéré (comme ceux que nous avons décrits au chapitre 2 pour le cas particulier du formalisme Unity), le problème est de construire une preuve formelle de la propriété spécifiée.

Le fonctionnement du système de vérification entre alors dans le cadre de la preuve automatique de théorèmes et fait appel à des démonstrateurs automatiques dont le rôle est de construire la preuve formelle souhaitée avec plus ou moins d'aide de la part de l'utilisateur.

L'inconvénient principal de cette seconde approche réside dans le faible degré d'automatisation des démonstrateurs automatiques qui lui-même provient de la complexité des théories mises en œuvre. Ces dernières, incluant au moins une partie importante de l'arithmétique entière, sont généralement indécidables, ce qui ôte tout espoir d'un outil entièrement automatique. Néanmoins, l'étude de ces démonstrateurs révèle différents degrés d'automatisation qui sont détaillés au paragraphe 3.1.4.

Enfin, on peut trouver un dernier avantage à l'approche par preuve sur l'approche par modèle. Dans le cas de la preuve, le démonstrateur fournit une preuve complète de la correction d'un programme. En théorie, cette preuve est lisible "à la main" et peut être contrôlée, notamment au moyen d'un autre démonstrateur. Il n'est donc pas absolument nécessaire d'avoir une confiance absolue dans l'outil de preuve que l'on utilise, alors qu'il le faut pour les vérificateurs de modèle. En quelque sorte, on peut dire que la réponse d'un vérificateur de modèle est plus riche en cas d'échec (contre-exemple) et que celle d'un démonstrateur est plus riche en cas de succès (preuve détaillée).

Raisonnement hybride

Comme toujours lorsque deux techniques fondamentalement différentes s'opposent, la notion d'approche hybride apparaît. Elle apparaît ici doublement, à la fois par l'introduction de preuves formelles dans les méthodes fondées sur les modèles et par l'utilisation de la vérification de modèle au sein de systèmes reposant sur la preuve.

Dans le premier cas, il s'agit de ramener, au moyen d'une preuve formelle, l'étude d'un système pour lequel les méthodes à base de modèle ne sont pas adaptées (nombre infini d'états par exemple) à celle d'un système transformé se pliant plus facilement aux contraintes de la méthode. Par exemple, un mécanisme dit *d'abstraction* permet d'extraire d'un problème concret un problème abstrait plus simple. En particulier, le problème abstrait peut être à états finis alors que le problème concret ne l'était pas. Des preuves formelles sont alors nécessaires pour transposer les propriétés démontrées sur le programme abstrait vers le programme concret. Un cas un peu particulier d'abstraction est celui de la *paramétrisation* : de la preuve d'un programme paramétré par une variable entière, on déduit la preuve de toutes ses instanciations selon cette variable.

C'est le cas, par exemple, d'un algorithme d'exclusion mutuelle pour N sites, N jouant le rôle du paramètre[1].

À l'autre extrémité, les systèmes fondés sur la preuve peuvent être amenés à déléguer des sous-problèmes à des *procédures* (ou des *semi-procédures*) *de décision*. C'est le cas, par exemple, du système STeP décrit au paragraphe 3.2.4 qui possède des procédures de décision pour les problèmes arithmétiques et pour la logique temporelle propositionnelle. Nous verrons également au chapitre 8 comment notre outil Dada utilise une procédure de décision de l'arithmétique de Pressburger pour réaliser ses preuves de formules Unity. Bien qu'il ne s'agisse pas à proprement parler de vérification de modèle (en particulier, comme c'est le cas pour l'arithmétique, les domaines peuvent rester infinis), il s'agit tout de même de déléguer la résolution de sous-problèmes à des outils à haut degré d'automatisation.

3.1.4 Degré d'automatisation

Un dernier critère très important de distinction entre les différents outils de preuve est leur degré d'automatisation. Cet aspect, qui est évidemment essentiel du point de vue de l'utilisateur d'un tel outil, est également fondamental pour le concepteur de ce type d'assistants. En effet, selon le degré d'automatisation dont on souhaite disposer en fin de compte, on peut choisir d'une part les fonctionnalités que l'on pourra réaliser et d'autre part les outils de preuve sur lesquels on pourra s'appuyer.

Par exemple, si l'on souhaite réaliser un outil a fort degré d'automatisation, il est illusoire de vouloir résoudre des problèmes indécidables dans le cas général. Un démonstrateur capable de prouver automatiquement *n'importe quelle* spécification vis-à-vis de *n'importe quel* programme n'est pas près de voir le jour. D'autre part, on sait dès le départ qu'il sera très difficile à un outil automatique de s'appuyer sur des démonstrateurs moins automatiques que lui, alors que l'inverse posera certainement moins de difficultés.

Il est donc important de savoir dans quelle catégorie d'automatisation on se trouve. Schématiquement, on peut distinguer deux extrêmes dans le domaine de l'automatisation, ainsi qu'une position intermédiaire :

Les outils entièrement automatiques. Du point de vue de l'utilisateur, ils correspondent à un idéal : il lui suffit de poser son problème et il obtient de la part du démonstrateur la réponse sans autre intervention qu'un simple clic de souris. Évidemment, la contrepartie est la limitation des fonctionnalités. La majeure partie des outils de vérification de programmes entièrement automatiques sont actuellement basés sur la synthèse ou la vérification de modèle, avec les limitations que nous avons vues. Néanmoins, ces outils sont suffisamment performants pour être utilisés sur des problèmes réels et constituent aujourd'hui l'immense majorité des produits commercialisés.

1. Il s'agit en quelque sorte de programmes à états finis non bornés.

Les vérificateurs de preuve a posteriori. Appelés aussi "éditeurs de preuve", ce sont les moins automatiques des démonstrateurs puisqu'ils se contentent de vérifier la cohérence d'une preuve qui leur est fournie intégralement, jusque dans les moindre détails. Aucune aide n'est fournie, a priori, pour la conception même de la preuve.

Néanmoins, l'écriture d'une telle preuve nécessite d'interagir avec le démonstrateur pour déterminer les parties manquantes. De plus, l'utilisation de bibliothèques de théorèmes permet de dispenser l'utilisateur de certaines étapes lorsqu'elles sont similaires à des preuves déjà faites. Pour ces deux raisons, on peut considérer que ces outils possèdent un aspect interactif qui les relie à la catégorie suivante.

Les outils interactifs. Cette catégorie regroupe toute la continuité d'intermédiaires entre les deux extrêmes précédents. Ce qui différencie ces démonstrateurs est ce qu'il parviennent à faire d'eux-mêmes entre les interventions de l'utilisateur. Par exemple, le fait de pouvoir résoudre des sous-problèmes arithmétiques par appel à une procédure de décision plutôt que de construire une preuve dans une théorie arithmétique quelconque, est un avantage important de certains systèmes.

3.2 Exemples d'outils

Dans ce paragraphe, nous présentons brièvement quelques outils qui ont été développés ces dernières années autour des formalismes de systèmes de transitions d'états et de logique temporelle. Nous précisons, pour chacun d'eux, sa position vis-à-vis des critères que nous venons d'énumérer (Fig. 3.1).

Le premier de ces systèmes, construit autour du démonstrateur HOL, constitue une référence majeure du domaine. Les trois autres ont été choisis pour la variété de leurs approches (éditeur de preuve, vérificateur de modèle, système hybride) et parce qu'ils ont chacun une particularité qui nous a semblé intéressante à développer (utilisation d'une sémantique opérationnelle pour le premier, mélange des sémantiques de B.A. Sanders et de K.M. Chandy et J. Misra pour le deuxième, et génération d'invariants pour le dernier).

D'autres systèmes intéressants ne sont pas décrits et sont brièvement cités dans la bibliographie à la fin de ce chapitre.

3.2.1 UNITY-HOL

En 1992, F. Andersen a défini la théorie Unity au sein du démonstrateur HOL (Higher Order Logic). Cet outil manipule, à travers le langage ML, un ensemble d'axiomes, de constantes logiques et de règles d'inférences permettant de décrire des théories logiques et des preuves au sein de ces théories. Il est ainsi possible de décrire, dans cette logique primitive de HOL, les notions d'états, de transitions d'états et les définitions des opérateurs logiques de Unity telles qu'elles ont été proposées par K.M. Chandy et J. Misra. Ainsi, on peut alors traduire un programme Unity et sa spécification en logique HOL et construire au sein du système la preuve de correction.

Ce système s'appuie donc sur le formalisme Unity dans sa sémantique d'origine. Il manipule à la fois un programme et une spécification, mais peut être utilisé pour démontrer des théorèmes de la logique en ne faisant aucune hypothèse explicite sur le programme manipulé. Il se base entièrement sur des preuves axiomatiques et ne souffre donc d'aucune limitation théorique (ensembles d'états infinis, sûreté et vivacité, ...). Par contre, il ne s'agit que d'un éditeur de preuve ne disposant d'aucune forme d'automatisme. Le système ne dispose d'aucune interface spécifique à Unity, et l'utilisateur se situe toujours au niveau du système HOL, dans la syntaxe et la sémantique de ce système, ce qui suppose une grande connaissance de HOL.

Le démonstrateur HOL est basé sur une logique suffisamment puissante pour pouvoir représenter la logique Unity sans axiomes supplémentaires. Par conséquent, si l'on admet que la théorie de base de HOL est consistante, et si (comme c'est le cas) on parvient à redémontrer les différents théorèmes et méta-théorèmes classiques de la logique Unity (sans l'axiome de substitution), on peut en déduire la consistance de la logique Unity et la validité de tous ses théorèmes classiques.

3.2.2 UNITY-Nqthm

En 1992 également, D.M. Goldschlag a présenté une traduction du formalisme Unity en logique de Boyer et Moore, mécanisable par le système Nqthm. La démarche est similaire à celle de F. Andersen, mais les réalisations diffèrent autant que les démonstrateurs sous-jacents sont différents.

Dans le cas de Nqthm, la logique utilisée est moins riche (logique du premier ordre avec quantification universelle implicite sur les variables libres) et a conduit D.M. Goldschlag a considérer Unity dans une sémantique opérationnelle basée sur les séquences d'états. Il s'appuie donc sur une définition des opérateurs de la logique Unity similaire à celle que nous avons présentée au paragraphe 2.1.9. Il raisonne donc toujours sur une séquence d'exécution, éventuellement contrainte par la description d'un programme Unity dans la syntaxe de la logique de Boyer et Moore. De la même manière que pour Unity-HOL, lorsque la séquence considérée n'est pas contrainte, le système peut permettre de démontrer les théorèmes de la logique Unity.

Le système obtenu utilise donc le formalisme Unity, permet de considérer à la fois une spécification et un programme et repose sur la construction de preuves. Bien que le démonstrateur Nqthm soit capable de réaliser de lui-même certains raisonnements par récurrence et qu'il dispose d'une semi-procédure de décision arithmétique, le codage des programmes et de la logique Unity est trop complexe pour que le démonstrateur puisse utiliser efficacement ces possibilités. Le système obtenu n'est donc pas beaucoup plus automatique que celui basé sur HOL, et il oblige également l'utilisateur à être un spécialiste de Nqthm.

Dans ce système, aucun axiome n'est ajouté et les théorèmes et méta-théorèmes de la logique Unity sont des théorèmes de la sémantique opérationnelle. Un modèle est donc explicitement décrit, ce qui garantit la cohérence de la logique.

Par contre, il est intéressant de noter que tous les résultats s'appliquent aux opérateurs de la logique UNITY *définis sur une seule séquence d'états*, c'est-à-dire ceux que nous avons notés $prop_\sigma$ au paragraphe 2.1.9[2]. Par exemple, lorsque D.M. Goldschlag démontre le théorème PSP, il démontre en fait :

$$\frac{p \ \mapsto_\sigma \ q \ \mathtt{in}\ \sigma,\ r \ \mathtt{unless}_\sigma\ b \ \mathtt{in}\ \sigma}{p \wedge r \ \mapsto_\sigma \ (q \wedge r) \vee b \ \mathtt{in}\ \sigma}$$

Mais puisque

$$\begin{array}{ll} p \ \mathtt{unless}_\mathcal{O}\ q \ \mathtt{in}\ F & \equiv \langle \forall \sigma : \sigma \in \mathcal{O}.F :: p \ \mathtt{unless}_\sigma\ q \ \mathtt{in}\ \sigma \rangle \\ p \ \mapsto_\mathcal{O} \ q \ \mathtt{in}\ F & \equiv \langle \forall \sigma : \sigma \in \mathcal{O}.F :: p \ \mapsto_\sigma \ q \ \mathtt{in}\ \sigma \rangle \end{array}$$

le théorème est aussi démontré pour les opérateurs *leads-to*$_\mathcal{O}$ et *unless*$_\mathcal{O}$.

En fait, il se trouve que tous les théorèmes classiques de la logique UNITY sont en fait valables pour les opérateurs $prop_\sigma$, c'est-à-dire sous une forme plus forte que celle sous laquelle ils sont généralement utilisés. C'est ce qui permet à D.M. Goldschlag de les démontrer au sein de son système.

Par contre, les opérateurs $prop_\sigma$ possèdent des propriétés supplémentaires par rapport aux opérateurs classiques. Par exemple, D.M Goldschlag démontre que si $p \ \mapsto \ q$ n'est pas vrai, alors le prédicat $\neg q$ est "finalement invariant"[3]. Ceci n'est vrai que pour les opérateurs s'appliquant à une seule séquence. En effet, dans le cas d'un *leads-to* standard, le fait de ne pas être valide implique seulement qu'il existe *une* séquence d'exécution ne vérifiant pas la propriété et non que *toutes* les séquences finissent par satisfaire un prédicat.

Il faut également remarquer que, les théorèmes étant démontrés *séquence par séquence*, la démonstration ne dépend pas de la manière dont sont obtenues ces séquences. Par conséquent, on en déduit la validité des théorèmes pour n'importe quelle sémantique opérationnelle de UNITY.

3.2.3 UNITY Verifier

M. Kaltenbach a développé en 1994 un outil baptisé UV pour UNITY Verifier. Il s'agit d'un vérificateur de modèle (*model-checker*) pour les programmes UNITY à états finis et les propriétés non quantifiées de la logique, appliquées à ces programmes. En contrepartie de ces limites, le système UV est entièrement automatique, très simple à utiliser et permet la preuve de tous les types de propriétés UNITY, y compris les propriétés de vivacité.

Les propriétés manipulées par UV sont en fait les propriétés définies par B.A. Sanders que nous avons présentées au paragraphe 2.2.2 et que nous avons notées $prop_s$. Par contre, selon la façon dont la propriété est prouvée, il est possible de déterminer si une propriété est vraie au sens de K.M. Chandy et J. Misra ou au sens de Sanders. On peut néanmoins regretter qu'UV ne propose pas deux syntaxes différentes permettant de différencier ces deux types de propriétés.

2. Ceci justifie que l'on puisse utiliser les théorèmes de la logique UNITY sur ces opérateurs, ce que nous faisons dans [CP97], et ce qui n'était pas évident a priori.

3. "If $p \ \mapsto \ q$ is not true, then eventually the negation of q is invariant" [Gol92, (p. 28)]. D.M. Goldschlag définit en UNITY une notion de "finalement invariant" correspondant à $\Diamond\Box$ en logique temporelle linéaire.

Par exemple, en ce qui concerne les invariants d'un programme, l'utilisateur dispose de trois techniques pour démontrer qu'un prédicat est invariant :

- il peut d'abord chercher à démontrer directement l'invariant qu'il propose, c'est-à-dire vérifier si le prédicat est maintenu vrai par toutes les transitions du programme et impliqué par la condition initiale [4] ;

- si la preuve précédente a échoué, l'utilisateur peut alors demander au système d'"enrichir" le prédicat avec les autres invariants déjà prouvés. Le processus précédent est alors appliqué à la conjonction du prédicat proposé et de tous les invariants validés par le système au moment de la vérification ;

- enfin, en cas d'échec de la méthode précédente, l'utilisateur peut alors demander au système de calculer le plus fort invariant du programme et d'enrichir le prédicat proposé avec cet invariant.

On peut donc remarquer que si un invariant est prouvé avec la première technique, il s'agit d'un invariant inductif tel qu'ils sont définis par K.M. Chandy et J. Misra. Par contre, si un invariant est vérifié par l'une des deux autres méthodes, il s'agit d'un invariant au sens de B.A. Sanders, c'est-à-dire une propriété "toujours vraie". Enfin, il faut noter que seul un échec de la troisième méthode permet d'affirmer que le prédicat n'est pas invariant (ni dans un sens, ni dans l'autre) et peut être falsifié par le programme.

Le système UV propose différents algorithmes pour le calcul du plus fort invariant d'un programme. Ces algorithmes sont itératifs et se répartissent schématiquement en deux classes : un algorithme nécessitant peu d'itérations (le logarithme du diamètre de l'espace des états accessibles) mais coûteux en mémoire, et deux algorithmes plus lents (nombre d'itérations égal au diamètre de l'espace des états accessibles) mais plus économiques en mémoire. La terminaison de ces trois algorithmes est théoriquement garantie puisque les programmes considérés ont un espace d'états fini, mais un échec reste possible en raison des limites de la machine. C'est la raison pour laquelle le système UV propose les deux techniques de preuves intermédiaires permettant d'éviter le calcul du plus fort invariant.

3.2.4 STeP

Le système STeP a été développé par une équipe importante dirigée par Z. Manna à partir de 1994. Il s'agit d'un système complexe qui cherche à faire cohabiter différentes techniques de vérification comme la vérification de modèle, la preuve (plus ou moins automatique) ou les procédures de décision. Il s'appuie sur l'utilisation de propriétés spécifiées en logique temporelle linéaire et de systèmes de transitions équitables décrits dans le langage SPL (*Simple Programming Language* : affectation, séquencement, structures de contrôle, primitives

4. En fait, le système enrichit déjà automatiquement le prédicat par la contrainte que toutes les variables prennent une valeur de leur type.

de synchronisation, étiquettes d'instructions). Ces langages de spécification et de programmation restent proches du formalisme Unity[5].

Une des forces de ce système est l'intégration de différents types d'outils au sein d'une syntaxe et d'une sémantique communes. Le vérificateur de modèle peut être utilisé pour les systèmes dont le nombre d'états n'est pas trop important. Sinon, STeP possède un démonstrateur automatique capable d'appliquer un certain nombre de tactiques construites à partir de règles de la logique temporelle linéaire. Ce démonstrateur dispose de plusieurs procédures de décision (arithmétique, logique temporelle propositionnelle, ...) et d'un démonstrateur de prédicats du premier ordre avec égalité dont le fonctionnement est similaire à celui du démonstrateur Otter (cf chap. 8). Enfin, un démonstrateur interactif permet de contrôler précisément l'application des règles d'inférence et des simplifications à travers un calcul de séquents.

L'autre grande force du système STeP est sa capacité à engendrer et à enrichir des invariants. On a vu au paragraphe 2.2.3 qu'une des difficultés de la preuve de la validité d'une propriété vis-à-vis d'un programme était de trouver un invariant suffisamment fort pour restreindre les états considérés au cours de la preuve de façon à se rapprocher le plus possible des seuls états accessibles. Le système UV, présenté au paragraphe précédent, utilise les invariants déjà vérifiés pour enrichir un prédicat supposé invariant. Les programmes manipulés ayant un nombre d'états finis, il peut même calculer le plus fort invariant et déterminer ainsi exactement l'ensemble des états accessibles. Dans le cas des programmes manipulés par STeP, le plus fort invariant n'est en général pas calculable. Néanmoins, le système offre différentes manières d'engendrer des invariants et d'enrichir des prédicats.

Le premier groupe de techniques consiste à déduire des formules invariantes directement à partir du texte du programme, sans considérer les spécifications à démontrer. Par exemple, le système traduit en terme d'invariants les contraintes de type des variables[6]. Il peut aussi traduire directement la sémantique des primitives de synchronisation ou de l'affectation : *à la position qui suit l'instruction $x := 2$, la variable x a pour valeur* 2, ou *à la position qui suit l'instruction* await p*, le prédicat p est vrai.* Enfin, STeP cherche aussi des invariants appelés *linéaires*. Ces invariants sont des relations linéaires entre les variables du programme. En traduisant les contraintes d'invariance de ce type de formules, le système est capable d'engendrer un système d'équations linéaires et de le résoudre. Il peut ainsi construire différentes formes d'invariants linéaires à partir du texte du programme.

L'autre méthode utilisée par STeP cherche à enrichir un prédicat donné de façon à obtenir un invariant inductif qui implique le prédicat de départ. Par exemple, si p est le prédicat supposé invariant, le système choisit une transition τ qui falsifie le triplet de Hoare $\{p\}\,\tau\,\{p\}$ (s'il n'en existe pas, le prédicat p est invariant par induction, pourvu qu'il soit vérifié par les conditions initiales). Le prédicat p est ensuite remplacé par le prédicat $p \wedge wp(\tau, p)$. Ce prédicat est alors

5. Même si les notions de séquencement et de blocs conduit à une manipulation assez fastidieuse des étiquettes.

6. Ce que fait également UV. Cf note 4.

nécessairement stable pour la transition τ, mais peut ne plus l'être vis-à-vis de transitions pour lesquelles le prédicat p était stable. On peut donc recommencer le processus à partir de ce nouveau prédicat. Après plusieurs itérations, si on obtient un prédicat stable vis-à-vis de toutes les transitions et valide dans les conditions initiales, l'invariance du prédicat p est établie. Si à un moment du processus, on obtient un prédicat non valide dans l'état initial, on peut en conclure que le prédicat p n'est pas un invariant du programme. Si le processus ne s'arrête pas, on ne peut rien conclure (la terminaison du processus n'est garantie que dans le cas des systèmes dont l'espace d'états est fini).

Critères → Exemples ↓	Spécifications	Formalisme	Raisonnement	Automatisation
UNITY-HOL	Spécif./Spécif. Spécif/Modèle	UNITY	par preuve	nulle
UNITY-NQTHM	Spécif./Spécif. Spécif/Modèle	UNITY	par preuve	nulle
UV	Spécif/Modèle	UNITY	par modèle	totale
STeP	Spécif/Modèle	SLP logique temporelle	hybride	partielle

FIG. 3.1 – *Positionnement des exemples vis-à-vis des critères.*

3.3 Bibliographie

Les critères proposés pour les systèmes d'assistance sont personnels, subjectifs et ne constituent pas l'unique possibilité de classification. On trouve, par exemple, des critères sensiblement différents dans [HK93].

En ce qui concerne les outils présentés, chacun est associé à une bibliographie importante.

Le UNITY-HOL de F. Andersen fait l'objet de sa thèse [And92, APP93, APP95]. Le démonstrateur HOL quant à lui est décrit dans [GM93]. UNITY-NQTHM est décrit dans la thèse de D.M. Goldschlag [Gol92], le démonstrateur NQTHM et sa logique faisant l'objet du livre [BM88]. D'autres systèmes basés sur le formalisme UNITY ont vu le jour à partir d'autres démonstrateurs. Citons le système de P. Ladagnous [Lad96] à partir du démonstrateur DEVA [WSL93] ou celui de B. Heyd et P. Crégut [HC96] à partir de COQ [Coq96]. Ce dernier utilise à la fois une sémantique axiomatique comme UNITY-HOL et une sémantique opérationnelle comme UNITY-NQTHM, ce qui permet de montrer formellement des relations entre les deux sémantiques. Le système de B. Chetali [Che96, Che95] s'appuie sur le calcul de *wp* et utilise le Larch Prover [GG91], celui de N. Brown [BM93, Bro94, BM95] utilise l'outil de preuve associé à la méthode B [BP 91].

La thèse de M. Kaltenbach [Kal96] présente le système UV. On peut aussi consulter [Kal94a] pour le fonctionnement et [Kal94c] pour l'utilisation de ce système. Certains aspects théoriques sous-jacents au système (calcul du plus fort invariant, disjonction finie de la propriété *leads-to*) sont présentés dans [Mis94c].

Le système STeP est présenté dans [Ma94, Ma95]. La syntaxe de programmation et la logique sur lesquels il s'appuie sont définis dans [MP92]. La technique utilisée dans STeP pour enrichir un prédicat supposé invariant apparaissait déjà dans [Sha93] où le semi-algorithme est décrit dans le détail.

Chapitre 4

Le raffinement de programmes

Ce chapitre est consacré à la présentation formelle des notions de raffinement de programme et de spécification. Dans une première partie, nous rappelons le principe de la notion de raffinement, en particulier dans le cadre de la preuve de programmes réactifs. Nous montrons que dans ce cadre, deux formes relativement différentes de raffinement sont possibles : l'une s'attache à transformer des instructions ou des blocs d'instructions ; l'autre s'applique aux comportements globaux des programmes. Nous présentons ensuite une définition formelle, d'abord du raffinement d'instruction, puis du raffinement de comportement.

Une deuxième partie étudie les notions de raffinement précédentes dans le cadre particulier du formalisme Unity. Selon que l'on considère le formalisme Unity sous l'angle de sa logique, ou sous l'angle des systèmes de transitions décrits, deux classes de relations de raffinement de comportement émergent : le raffinement de propriétés, fondé sur la conservation de propriétés logiques, et le raffinement opérationnel, fondé sur la non génération de séquences d'états. Ces deux classes de relations de raffinement se divisent encore, en particulier en fonction de la sémantique choisie pour les opérateurs de la logique. Nous comparons sémantiquement les différentes formes de raffinement de comportement obtenues. Enfin, cette partie se termine en évoquant l'intégration, au formalisme Unity, d'une notion de raffinement d'instruction.

La dernière partie de ce chapitre s'attache à deux transformations particulières de programmes Unity : la superposition et le renforcement des gardes. Ces transformations sont confrontées aux différentes relations de raffinement précédemment définies. En particulier, nous montrons que le théorème de renforcement des gardes se présente sous des formes relativement différentes selon la relation de raffinement choisie. Cette partie se termine par un exemple : une succession de raffinements, dont la superposition et le renforcement des gardes, sont appliqués pour transformer une communication entre deux processus via un canal non borné en une communication via un buffer à une case.

4.1 Généralités

4.1.1 Principe du raffinement

La notion de raffinement a été introduite pour faciliter la conception des programmes. Intuitivement, un raffinement est une transformation qui ne remet pas en cause les fonctionnalités d'un programme. Une telle transformation permet donc de modifier une partie d'un programme sans remettre en cause le reste du développement, puisque la partie modifiée conserve ses propriétés. En particulier, des raffinements successifs peuvent permettre de passer d'une vision abstraite (spécification) d'un programme à une vision concrète (implantation).

On retrouve cette idée de raffinement, très légèrement modifiée, lorsqu'on s'intéresse à l'étude formelle et à la correction des programmes. Un raffinement est alors, intuitivement, une transformation qui conserve les propriétés d'un programme ou d'une spécification.

Une transformation par raffinements peut donc permettre de construire un programme prouvé correct par rapport à une spécification à l'aide de la seule relation de raffinements, sans l'effort d'une preuve supplémentaire de correction. Idéalement, une succession de raffinements peut permettre de passer d'une spécification formelle (correcte vis-à-vis d'elle-même,par définition) à une implantation sans perdre la correction.

Dans le domaine des systèmes réactifs, deux visions de la notion de raffinement sont possibles. La première s'attache à raffiner des instructions ou des blocs d'instructions. Elle dérive directement de l'approche classique de raffinement dans les programmes séquentiels. Seules quelques notions supplémentaires, quasiment inexistantes dans les programmes séquentiels, sont à considérer. C'est le cas notamment des notions d'équité et de vivacité.

La seconde forme de raffinement considère les programmes au niveau de leur comportement global. Elle s'avère assez différente de la précédente. En particulier, les notions de pré-condition et post-condition, applicables au niveau de l'instruction, du bloc d'instructions ou du programme transformationnel, ne sont plus utilisables au niveau du comportement (infini dans le temps) d'un programme réactif.

Naturellement, ces deux visions du raffinement sont étroitement liées. En particulier, on peut espérer, lorsqu'une instruction d'un programme réactif est raffinée (au sens de la première forme de raffinement), que le nouveau comportement global du programme raffine l'ancien (au sens de la seconde forme de raffinement).

La première forme de raffinement est parfois appelée raffinement de *programme* par opposition à l'autre qui est considérée comme du raffinement de *spécification*. Dans ce chapitre, nous préférons plutôt parler de raffinement d'*instruction* et de raffinement de *comportement*.

Toutes les relations de raffinement, que ce soit d'instruction ou de comportement, ont en commun d'être des relations de pré-ordre (relation reflexive et transitive). Ce pré-ordre peut ensuite être utilisé pour définir une forme d'équivalence entre instructions, programmes ou spécifications : si deux systèmes se raffinent mutuellement, ils sont considérés comme équivalents. Compte tenu de cette équivalence, la relation de raffinement devient alors une relation d'ordre.

4.1.2 Le raffinement d'instruction

C. Morgan utilise une métaphore arithmétique pour présenter le raffinement d'instruction. Pour lui, concevoir un programme revient à résoudre une équation. Par exemple, l'équation $x^2 - x - 1 = 0$ peut être vue comme une spécification (non implantable) de la variable x. La solution $x = \frac{1+\sqrt{5}}{2}$ constitue alors une implantation possible de la spécification précédente. Des critères sur la forme des équations permettent de distinguer facilement les expressions qui correspondent à des spécifications de celles qui peuvent être implantées. Ces critères dépendent évidemment des possibilités offertes par la machine cible : n'est-elle capable que d'affectation, ou peut-elle résoudre des équations du second degré ?

Voir la conception de programmes comme la résolution d'équations a deux conséquences importantes :

- nécessité d'un formalisme commun : pour pouvoir considérer les programmes comme les solutions des spécifications, il faut disposer d'un formalisme commun permettant d'exprimer à la fois les équations et leurs solutions, c'est-à-dire les spécifications et les programmes ;
- nécessité de règles de calcul : de la même façon qu'on résout rarement une équation en exhibant une valeur puis en montrant que cette valeur est solution de l'équation, les programmes solutions doivent être *calculés* à partir de leur spécification.

C. Morgan propose donc à la fois un langage unifiant spécifications et programmes, ainsi que des règles de calcul permettant de construire des programmes dans ce langage. Le langage qu'il utilise est celui des commandes gardées de Dijkstra auquel sont ajoutées des déclarations qui jouent le rôle d'"instructions de spécification". Ces déclarations permettent de passer d'une pré-condition donnée à une post-condition donnée, mais ne fournissent pas le moyen concret de réaliser cette transformation[1]. L'objectif de la conception d'un programme est de remplacer ces déclarations par des blocs d'instructions en faisant apparaître des instructions concrètes, jusqu'à obtenir un programme ne comportant plus de déclarations. Un tel programme constitue une implantation de la spécification de départ. Dans ce contexte, le mot *programme* représente un programme contenant éventuellement des déclarations. Il peut donc s'agir d'une spécification de départ, d'une implantation finale, ou de n'importe quel intermédiaire.

La relation de raffinement utilisée est notée $\sqsubseteq$. Elle s'applique à des programmes ou parties de programmes. On dit que s *se raffine* en s' ou que s' est *un raffinement* de s, et on note $s \sqsubseteq s'$, si et seulement si, pour tout prédicat p, $wp(s,p) \Rightarrow wp(s',p)$. Il s'agit d'une relation de raffinement en ce sens que

1. Il existe même des déclarations qualifiées de "miraculeuses" (qui ne respectent pas la loi du miracle exclu) pour lesquelles il n'existe pas de réalisation concrète.

si s vérifie une spécification, s' la vérifiera aussi. Plus précisément, si s vérifie la spécification $pre \Rightarrow wp(s, post)$ et si $s \sqsubseteq s'$, alors s' vérifie la spécification $pre \Rightarrow wp(s', post)$.

De manière à éviter la preuve d'une relation de spécification du type $s \sqsubseteq s'$ à partir de sa définition, des règles de calcul sont introduites. Ces règles de calcul permettent, à partir d'un programme s, de déduire un programme s' tel que la relation de raffinement $s \sqsubseteq s'$ soit garantie sans aucune obligation supplémentaire de preuve. Ces règles permettent, par exemple, d'introduire ou d'éliminer des affectations, des variables auxiliaires, des constantes, des gardes, etc. Pour poursuivre la métaphore mathématique, les auteurs comparent ce calcul au calcul intégral : il est correct (l'application des règles ne remet pas en cause la correction du programme) mais incomplet et non systématique (partant d'une spécification donnée, on n'est pas certain de pouvoir obtenir un programme solution à l'aide de ce calcul).

4.1.3 Le raffinement de comportement

Nous avons vu au paragraphe 1.2.2 que le comportement d'un système réactif pouvait se caractériser par l'ensemble des exécutions possibles du système. Les relations de raffinement de comportement que nous décrivons ici vont porter sur les sémantiques opérationnelles des systèmes, c'est-à-dire les ensembles de leurs exécutions possibles.

Une exécution est constituée de la séquence infinie des états du système. Dans le cadre du raffinement, il est souvent utile de distinguer au niveau de l'état une partie interne et une partie externe. Seule la partie externe est supposée observable et les spécifications du système ne portent que sur cette partie.

On considère donc qu'un état σ_i du système est composé d'une partie externe et d'une partie interne. On note $\Pi_E.\sigma_i$ la partie externe de l'état σ_i et $\Pi_E.\sigma$ la restriction de σ aux parties externes de ses états. La relation de raffinement de comportement est alors définie de la façon suivante :

$$S \sqsubseteq S' \ \equiv\ \langle \forall \sigma' : \sigma' \in \Gamma.\mathcal{O}.S' :: \langle \exists \sigma : \sigma \in \Gamma.\mathcal{O}.S :: \Pi_E.\sigma = \Pi_E.\sigma' \rangle \rangle$$

$S \sqsubseteq S'$ signifie que tout comportement (externe) du système S' peut être produit par le système S. En d'autres termes, le système S' n'introduit pas de comportement nouveau. Il possède donc toutes les propriétés du système S.

L'opérateur Γ, défini au paragraphe 1.2.2, est utilisé pour permettre à un raffinement d'introduire ou de supprimer du bégaiement. Cette précaution s'avère nécessaire, par exemple, pour la transformation présentée au paragraphe 4.3.2 ou pour des transformations faisant intervenir le produit défini au chapitre 7.

Nous avons dit précédemment qu'un ensemble d'exécutions possibles était caractéristique aussi bien d'un programme que d'une spécification : au texte d'un programme correspond, dans une sémantique donnée, l'ensemble de ses exécutions possibles ; aux formules d'une spécification correspond, en sémantique opérationnelle, l'ensemble des exécutions qui satisfont ces formules. Par conséquent, la notion de raffinement que nous venons de présenter est aussi bien un raffinement de programmes qu'un raffinement de spécifications. C'est pourquoi nous préférons parler de raffinement de comportement.

On distingue souvent deux cas particuliers de la notion de raffinement précédente. Le premier cas est celui où le système est décrit par un système de transitions équitable. On définit dans ce cas la notion de *fonction de raffinement*.

Soient $S = (\mathcal{V}, \Sigma, \mathcal{T}, \phi)$ et $S' = (\mathcal{V}', \Sigma', \mathcal{T}', \phi')$ deux systèmes de transitions équitables (cf § 1.2.3). Une fonction $f : \Sigma' \rightarrow \Sigma$ est dite fonction de raffinement si et seulement si elle vérifie les contraintes suivantes :

– $\langle \forall \sigma'_i : \sigma'_i \in \Sigma' :: \Pi_E.(f.\sigma'_i) = \Pi_E.\sigma'_i \rangle$
(Une fonction de raffinement ne modifie pas la partie externe des états.)

– $\langle \forall \sigma'_i : \sigma'_i \in \Sigma' :: \sigma'_i.\phi' \Rightarrow (f.\sigma'_i).\phi \rangle$
(À un état initial du système S', une fonction de raffinement associe un état initial du système S.)

– $\langle \forall \sigma'_i, \sigma'_j : \sigma'_i, \sigma'_j \in \Sigma' \wedge \langle \exists \tau' : \tau' \in \mathcal{T}' :: \sigma'_j \in \tau'.\sigma'_i \rangle ::$
$\quad \langle \exists \tau : \tau \in \mathcal{T} :: f.\sigma'_j \in \tau.(f.\sigma'_i) \rangle \rangle$ [2]
(À deux états du système S', liés par une transition de $\mathcal{T}'$, une fonction de raffinement associe deux états du système S liés par une transition de $\mathcal{T}$.)

– $\langle \forall \sigma' : \sigma' \in \mathcal{O}.S' ::$
$\quad (f.\sigma'_0, f.\sigma'_1, \cdots, f.\sigma'_i, \cdots)$ satisfait aux contraintes d'équité de $S \rangle$
(Les images, par la fonction de raffinement, des exécutions du système S' satisfont aux contraintes d'équité du système S.)

On peut démontrer que, s'il existe une fonction de raffinement d'un système S' vers un système S, alors le système S' est un raffinement du système S. Exhiber une fonction de raffinement constitue donc un moyen efficace de démontrer une relation de raffinement. En effet, cela permet de raisonner au niveau des transitions et des états et non au niveau des exécutions complètes, excepté pour le dernier point qui concerne l'équité. En ce qui concerne ce dernier point, le raisonnement fait intervenir les exécutions complètes, mais peut encore tirer profit du fait que f est une fonction s'appliquant sur les états et non sur les exécutions.

Cette existence d'une fonction de raffinement est parfois prise comme définition de la relation de raffinement. La relation de raffinement telle que nous l'avons définie est alors appelée relation d'implantation. En effet, il faut noter que ces deux notions ne sont pas équivalentes : il peut y avoir relation de raffinement entre deux systèmes alors qu'il n'existe aucune fonction de raffinement les reliant.

Un autre cas particulier important, qui sera souvent utilisé dans la suite, est celui où l'état complet du système est considéré comme état externe. On doit alors pouvoir retrouver toute exécution du programme transformé dans la sémantique opérationnelle du programme d'origine. La relation devient alors :

$$S \sqsubseteq S' \;\equiv\; \Gamma.\mathcal{O}.S' \subset \Gamma.\mathcal{O}.S$$

2. En théorie, il faut prendre en compte le cas $f.\sigma'_j = f.\sigma'_i$, mais la présence imposée de l'instruction *skip* ramène ce cas particulier au cas général.

Dans ce cas particulier, les notions de relation de raffinement et d'existence d'une fonction de raffinement sont équivalentes : si $S \sqsubseteq S'$, la fonction identité est fonction de raffinement de S' dans S.

On peut remarquer que l'application de l'opérateur Γ à $\mathcal{O}.S'$ n'est pas nécessaire et n'est ajoutée que par symétrie. En effet, il suffit de vérifier que les séquences de $\mathcal{O}.S'$ sont présentes dans $\Gamma.\mathcal{O}.S$ pour assurer la relation de raffinement. Autrement dit :

$$\frac{\mathcal{O}.S' \subset \Gamma.\mathcal{O}.S}{S \sqsubseteq S'} \tag{4.1}$$

Ceci provient du fait que la fonction Γ est croissante pour l'inclusion et idempotente :

$$\mathcal{O}.S' \subset \Gamma.\mathcal{O}.S \;\Rightarrow\; \Gamma.\mathcal{O}.S' \subset \Gamma.(\Gamma.\mathcal{O}.S) \;\Rightarrow\; \Gamma.\mathcal{O}.S' \subset \Gamma.\mathcal{O}.S$$

4.2 Raffinement dans le formalisme UNITY

4.2.1 Introduction

Telle qu'elle est proposée par K.M. Chandy et J. Misra dans leur livre, la conception de programmes en UNITY repose principalement sur le raffinement de comportement. L'idée est de raffiner progressivement une spécification de départ jusqu'à obtenir une spécification suffisamment détaillée pour pouvoir en extraire simplement un programme. Si la spécification finale est bien détaillée, la preuve de la correction du programme extrait vis-à-vis de cette spécification reste simple. Le principe de raffinement garantit alors la correction de ce programme vis-à-vis de la spécification initiale.

On dispose donc naturellement d'une notion de raffinement de comportement qui s'appuie sur les opérateurs de la logique UNITY. Mais. comme nous l'avons vu, les programmes UNITY peuvent être assimilés à des systèmes de transitions équitables. Les notions de raffinement de comportement que nous venons de présenter pour ces systèmes s'appliquent donc également aux programmes UNITY. Nous présentons dans cette partie une comparaison de ces différentes formes de raffinement de comportement.

Enfin, la définition du langage UNITY repose sur les affectations gardées. Ces instructions ont une sémantique précisément définie et la notion de raffinement d'instruction que nous avons décrite pour les commandes gardées peut leur être appliquée. Nous évoquerons donc dans cette partie des travaux qui ont visé à intégrer une notion de raffinement d'instruction au formalisme UNITY.

4.2.2 Raffinement de propriétés

Une spécification UNITY est un ensemble de propriétés de la logique (*invariant*, *unless*, *stable*, *constant*, *ensures*, *leads-to*, *until*) portant sur les variables d'état. Pour une spécification S, nous définissons sa fermeture $Cl.S$ comme l'ensemble de toutes les propriétés que l'on peut déduire des propriétés de S dans une sémantique donnée de la logique :

$$Cl.S = \{P : P \text{ propriété de la logique UNITY} \mid S \vdash P\}$$

Si l'on considère des spécifications limitées à une partie des opérateurs de la logique, la fermeture correspondante sera composée uniquement d'opérateurs logiques de cette partie (cf § 4.2.3).

On définit donc différentes formes de fermeture en fonction de la sémantique choisie. Nous utiliserons, dans ce chapitre, Cl_{CM} pour la sémantique d'origine (cf § 2.1.4), Cl_{S} pour la sémantique définie par B.A. Sanders (cf § 2.2.2) et $Cl_{\mathcal{O}_1}$ pour la sémantique opérationnelle de la logique (cf § 2.1.9). Ces différentes formes de fermeture coïncident lorsque les sémantiques des opérateurs logiques sont les mêmes. C'est le cas notamment de Cl_{S} et $Cl_{\mathcal{O}_1}$ lorsqu'on ne considère pas les opérateurs de type *ensures*. Par contre, les fermetures diffèrent en général d'une sémantique à l'autre. Par exemple, on a toujours :

$$Cl_{\mathrm{S}}.\{\texttt{invariant}\ A \wedge B\} \supset \{\texttt{invariant}\ A, \texttt{invariant}\ B\}$$

ce qui n'est pas le cas, a priori, pour la fermeture Cl_{CM}. On notera Cl sans indice pour désigner l'une quelconque des différentes formes de fermeture. On retrouve donc dans $Cl.S$ les propriétés contenues dans S, ainsi que toutes leurs conséquences logiques, y compris les tautologies de la logique Unity.

L'idée naturelle de raffinement de comportement en Unity est de transformer une spécification S en une spécification S' de telle façon que la spécification S' vérifie toutes les propriétés de la spécification S. Formellement, on définit les relations de raffinement $\sqsubseteq_L$ de la façon suivante :

$$S \sqsubseteq_L S' \;\equiv\; Cl.S \subset Cl.S'$$

On obtient ainsi, selon la sémantique choisie pour les opérateurs logiques, différentes relations de raffinement. Nous considérerons dans la suite les relations $\sqsubseteq_{L_{\mathrm{CM}}}$, $\sqsubseteq_{L_{\mathrm{S}}}$ et $\sqsubseteq_{L_{\mathcal{O}_1}}$. La notation $\sqsubseteq_L$ désigne l'une quelconque de ces relations.

Ces relations de raffinement s'appliquent également aux programmes Unity, en assimilant un programme à l'ensemble de *toutes* les propriétés valides pour ce programme. Cette généralisation permet en particulier de considérer la relation de raffinement entre une spécification et un programme. Un programme est correct vis-à-vis d'une spécification si et seulement si toutes les propriétés de la spécification sont démontrables pour ce programme. En d'autres termes, si F est un programme Unity et S une spécification :

$$F \text{ est correct vis-à-vis de } S \;\equiv\; S \sqsubseteq_L F$$

Telle que nous venons de la définir, cette relation de raffinement pose néanmoins problème. Considérons, par exemple, les deux programmes F et G suivants :

```
Program F
Declare
  x : Int;
Initially
  x = 0;
Assign
  x:=x + 1  if x < 10
End F
```

```
Program G
Declare
  x : Int;
  fin : Bool;
Initially
  (x = 0) ∧ ¬fin;
Assign
  x:=x + 1  if x < 10
[]
  fin:=(x = 10)
End G
```

Le programme F gère un compteur croissant qui est incrémenté de 0 à 10 par pas de 1. Le programme G gère ce même compteur mais ajoute une variable booléenne fin qui détecte que la variable x a atteint la valeur 10.

Toutes les propriétés de F portant sur la variable x sont bien entendu valides dans G. Par contre, que dire de la propriété :

$$\texttt{constant}\ fin\ \texttt{in}\ F\ ?$$

Il est clair que l'instruction $s : x := x + 1$ **if** $x < 10$ vérifie bien les conditions $\{fin\}\, s\, \{fin\}$ et $\{\neg fin\}\, s\, \{\neg fin\}$. Néanmoins, si on considère que la propriété `constant` fin `in` F est valide, on n'a pas $F \sqsubseteq_L G$ puisque la variable fin n'est pas constante dans G. Or, la transformation de F en G constitue une *superposition*, telle que nous l'avons définie au paragraphe 2.1.6, et nous avons mentionné dans ce paragraphe qu'un des intérêts de cette transformation est qu'elle constitue un raffinement de programme.

Deux solutions à cette difficulté sont possibles. On peut considérer que, pour qu'une propriété non tautologique soit valide dans un programme, elle ne doit faire référence qu'à des variables déclarées dans ce programme. Dans ce cas, malgré la validité des triplets $\{fin\}\, s\, \{fin\}$ et $\{\neg fin\}\, s\, \{\neg fin\}$, la propriété `constant` fin `in` F n'est pas valide et on a bien $F \sqsubseteq_L G$.

Cette solution, si elle résout le problème de l'ajout de variables et de la superposition, peut s'avérer insuffisante dans certains cas. Par exemple, des difficultés subsistent lors de la suppression de variables auxiliaires devenues inutiles à la suite d'une série de transformations. La solution générale au problème consiste à distinguer les variables externes des variables internes, de la même façon qu'au paragraphe 4.1.3 on distinguait une partie visible et une partie cachée de l'état d'un programme.

Dans le cas des programmes F et G précédents, si l'on considère que le problème à résoudre est l'incrémentation de la variable x, la variable fin devient une variable auxiliaire cachée (utilisable par des raffinements à venir) et seules les propriétés portant sur x sont pertinentes. On a bien alors $F \sqsubseteq_L G$. Si, par contre, on considère que le problème posé est à la fois d'incrémenter la variable x *et* de détecter la fin de ce processus, la variable fin devient variable externe et intervient dans la spécification du problème. On n'a plus alors $F \sqsubseteq_L G$, mais ce n'est pas critique étant donné que le programme F ne résout pas le problème posé.

4.2.3 Variantes de la relation $\sqsubseteq_L$

En dehors des aspects de visibilité des variables que nous venons d'évoquer, différentes variations de la relation $\sqsubseteq_L$ peuvent être envisagées. Les différentes relations obtenues diffèrent principalement dans le choix des types de propriétés conservées lors d'un raffinement. De la même manière que l'on peut se restreindre à ne considérer qu'une partie des variables d'un programme, on peut se limiter lors d'un raffinement à la conservation d'un sous-ensemble des opérateurs de la logique.

On voit immédiatement que si l'on se limite aux propriétés de type *invariant*, *unless* et *ensures*, on obtient une relation de raffinement équivalente à la relation $\sqsubseteq_L$. En effet, les autres propriétés de la logique se déduisent de ces trois propriétés élémentaires.

La première variation réelle consiste à ne plus imposer à la relation de raffinement de conserver les propriétés de type *ensures*. On obtient alors une notion de raffinement de sûreté : seules les propriétés de sûreté sont conservées lors d'une transformation. Cette forme de raffinement peut être utile lorsqu'on ne considère que les aspects de sûreté dans le développement d'un programme. Évidemment, les aspects de vivacité doivent être pris en compte à un autre niveau (informel, par exemple) pour éviter les transformations non réalistes. En particulier, tout programme Unity se raffine, selon cette relation, en un programme conservant les mêmes sections *declare* et *initially*, mais n'ayant pas d'autre instruction que *skip*.

Une seconde variation, plus intéressante, consiste à remplacer la conservation des propriétés de type *ensures* par la conservation des propriétés de type *leads-to*. En effet, on peut considérer que les propriétés de type *ensures* ne sont qu'un outil permettant la preuve de propriétés de type *leads-to*, et que seules ces dernières sont pertinentes au niveau d'une spécification. On peut donc considérer comme suffisant que les propriétés de type *leads-to* se conservent, même si leur preuve nécessite des propriétés de type *ensures* différentes.

On définit donc la relation de raffinement $\sqsubseteq_{L'}$ comme la conservation des seules propriétés de type *invariant*, *unless* et *leads-to*. On obtient ainsi, selon la sémantique choisie, différentes relations de raffinement. Dans ce chapitre, nous utiliserons les relations $\sqsubseteq_{L'_{\mathrm{CM}}}$, $\sqsubseteq_{L'_{\mathrm{S}}}$ et $\sqsubseteq_{L'_{\mathcal{O}_1}}$ [3]. La notation $\sqsubseteq_{L'}$ désigne l'une quelconque de ces relations.

La relation d'équivalence qui correspond à la relation de raffinement $\sqsubseteq_{L'}$ est plus intuitive que celle qui correspond à la relation $\sqsubseteq_L$. Reprenons, par exemple, les programmes F et G du paragraphe 2.1.4 :

Program F	**Program** G
Assign	**Assign**
x:=0 **if** x > 0	x:=0
⫾	⫾
x:=0 **if** x < 0	*skip*
End F	**End** G

On a vu que ces programmes diffèrent au niveau de leur propriétés de type *ensures* alors que leurs comportements semblent similaires. En fait, on a les relations suivantes :

$$F \sqsubseteq_L G \qquad G \not\sqsubseteq_L F \qquad F \sqsubseteq_{L'} G \qquad G \sqsubseteq_{L'} F$$

pour la sémantique de K.M. Chandy et J. Misra et pour celle de B.A. Sanders. Les programmes F et G sont donc équivalents par rapport au relations de raffinement $\sqsubseteq_{L'_{\mathrm{CM}}}$ et $\sqsubseteq_{L'_{\mathrm{S}}}$.

Comme les propriétés invariantes se déduisent des propriétés de type *unless* et des conditions initiales, on peut souvent utiliser une condition suffisante de raffinement au sens de la relation $\sqsubseteq_{L'}$. En effet, si un programme G n'admet que des états initiaux admis par F, la conservation des propriétés de type *unless*

3. L'opérateur *ensures* n'étant pas défini en sémantique opérationnelle, les relations $\sqsubseteq_{L_{\mathcal{O}_1}}$ et $\sqsubseteq_{L'_{\mathcal{O}_1}}$ sont exactement les mêmes.

suffit à montrer la conservation des invariants. Par conséquent, on a la condition suffisante :

$$\dfrac{\left.\begin{array}{l} p \texttt{ unless } q \texttt{ in } F \;\Rightarrow\; p \texttt{ unless } q \texttt{ in } G \\ p \;\mapsto\; q \texttt{ in } F \;\Rightarrow\; p \;\mapsto\; q \texttt{ in } G \end{array}\right\} \text{pour tous prédicats } p \text{ et } q \text{ portant sur les variables visibles} \\ G.init \;\Rightarrow\; F.init}{F \sqsubseteq_{L'} G} \tag{4.2}$$

Il faut noter que la réciproque est fausse. En effet, un invariant nécessite toujours une clause d'initialisation, mais à une clause d'initialisation ne correspond pas nécessairement un invariant. Par exemple, si F et G sont les programmes suivants :

Program F	**Program** G
Declare	**Declare**
x : Int;	x : Int;
Initially	**Initially**
x = 0;	x = 1;
Assign	**Assign**
x := 0	x := 0
⫿	⫿
x := 1	x := 1
End F	**End** G

On a bien $F \sqsubseteq_{L'} G$ mais on n'a pas $G.init \Rightarrow F.init$.

4.2.4 Raffinement de sémantiques opérationnelles

On a défini, au paragraphe 4.1.3 une relation de raffinement s'appuyant sur les séquences d'états et applicable aux systèmes de transitions. Cette relation s'applique de façon naturelle aux programmes et aux spécifications UNITY vus comme des caractéristiques d'un ensemble de séquences d'états sous les hypothèses d'une sémantique donnée.

Si l'on considère la sémantique opérationnelle $\mathcal{O}_1$ définie au paragraphe 2.1.9, on dispose alors d'une relation de raffinement, notée $\sqsubseteq_{\mathcal{O}_1}$, similaire à celles présentées au paragraphe 4.1.3. En particulier, dans le cas où toutes les variables des programmes sont considérées visibles, cette relation devient :

$$F \sqsubseteq_{\mathcal{O}_1} G \;\equiv\; \Gamma.\mathcal{O}_1.G \subset \Gamma.\mathcal{O}_1.F$$

4.2.5 Relation entre les différentes formes de raffinement

Nous avons défini, dans les paragraphes précédents, différentes formes de raffinement de programmes et de spécifications en UNITY. Il s'agit des relations $\sqsubseteq_{L_{\mathrm{CM}}}$, $\sqsubseteq_{L_{\mathrm{S}}}$, $\sqsubseteq_{L_{\mathcal{O}_1}}$, $\sqsubseteq_{L'_{\mathrm{CM}}}$, $\sqsubseteq_{L'_{\mathrm{S}}}$, $\sqsubseteq_{L'_{\mathcal{O}_1}}$ et $\sqsubseteq_{\mathcal{O}_1}$.

Relation entre $\sqsubseteq_{L_{\mathcal{O}_1}}$, $\sqsubseteq_{L'_{\mathcal{O}_1}}$ et $\sqsubseteq_{L'_{\mathrm{S}}}$

On a déjà indiqué que, étant donné que l'opérateur *ensures* n'est pas défini en sémantique opérationnelle, les relations de raffinement $\sqsubseteq_{L_{\mathcal{O}_1}}$ et $\sqsubseteq_{L'_{\mathcal{O}_1}}$ sont trivialement équivalentes. D'autre part, il est clair que si deux sémantiques des opérateurs logiques coïncident, les relations de raffinement correspondantes sont les mêmes. Or, nous avons vu au paragraphe 2.2.2 que les opérateurs définis par B.A. Sanders (sauf l'opérateur $ensures_{\mathrm{S}}$) et les opérateurs définis sur les séquences de la sémantique $\mathcal{O}_1$ coïncident. On en déduit l'équivalence des relations de raffinement $\sqsubseteq_{L'_{\mathcal{O}_1}}$ et $\sqsubseteq_{L'_{\mathrm{S}}}$. Il reste donc à comparer les relations $\sqsubseteq_{L_{\mathrm{CM}}}$, $\sqsubseteq_{L_{\mathrm{S}}}$, $\sqsubseteq_{L'_{\mathrm{CM}}}$ et $\sqsubseteq_{L'_{\mathrm{S}}}$, d'une part entre elles, et d'autre part avec la relation $\sqsubseteq_{\mathcal{O}_1}$.

Relation entre $\sqsubseteq_L$ et $\sqsubseteq_{L'}$

Un premier résultat évident est que, pour deux programmes F et G :

$$\frac{F \sqsubseteq_{L_{\mathrm{CM}}} G}{F \sqsubseteq_{L'_{\mathrm{CM}}} G} \qquad\qquad \frac{F \sqsubseteq_{L_{\mathrm{S}}} G}{F \sqsubseteq_{L'_{\mathrm{S}}} G} \tag{4.3}$$

En effet, si toutes les propriétés de type *ensures* se conservent, les propriétés de type *leads-to*, qui s'en déduisent, se conservent également. Évidemment, la réciproque est fausse comme nous l'avons vu à travers le premier exemple du paragraphe 4.2.3.

Relation entre $\sqsubseteq_{\mathcal{O}_1}$ et $\sqsubseteq_{L_{\mathcal{O}_1}}$

Un autre résultat évident est que, si les séquences d'états se conservent, les propriétés logiques définies sur ces séquences se conservent également. On a donc trivialement, pour deux programmes F et G :

$$\frac{F \sqsubseteq_{\mathcal{O}_1} G}{F \sqsubseteq_{L_{\mathcal{O}_1}} G}$$

D'après l'équivalence entre les relations $\sqsubseteq_{L'_{\mathrm{S}}}$ et $\sqsubseteq_{L_{\mathcal{O}_1}}$, on en déduit, pour deux programmes F et G :

$$\frac{F \sqsubseteq_{\mathcal{O}_1} G}{F \sqsubseteq_{L'_{\mathrm{S}}} G} \tag{4.4}$$

Relation entre $\sqsubseteq_{L_{\mathrm{CM}}}$ et $\sqsubseteq_{L_{\mathrm{S}}}$ et entre $\sqsubseteq_{L'_{\mathrm{CM}}}$ et $\sqsubseteq_{L'_{\mathrm{S}}}$

Un résultat plus intéressant est le lien entre les relations $\sqsubseteq_{L_{\mathrm{CM}}}$ et $\sqsubseteq_{L'_{\mathrm{CM}}}$ et les relations $\sqsubseteq_{L_{\mathrm{S}}}$ et $\sqsubseteq_{L'_{\mathrm{S}}}$. En effet, même si l'on sait déjà que les propriétés de K.M. Chandy et J. Misra sont "plus fortes" que celles définies par B.A. Sanders, on ne peut rien en déduire a priori sur les relations de raffinement correspondantes[4]. Néanmoins, on peut démontrer le résultat suivant :

$$\frac{F \sqsubseteq_{L_{\mathrm{CM}}} G}{F \sqsubseteq_{L_{\mathrm{S}}} G} \qquad\qquad \frac{F \sqsubseteq_{L'_{\mathrm{CM}}} G}{F \sqsubseteq_{L'_{\mathrm{S}}} G} \tag{4.5}$$

En effet, si une transformation conserve toutes les propriétés Unity au sens de K.M. Chandy et J. Misra, elle conserve nécessairement l'invariance du plus

4. De la même manière que de $A \Rightarrow A'$ et $B \Rightarrow B'$ on ne peut déduire ni $\dfrac{A \Rightarrow B}{A' \Rightarrow B'}$ ni $\dfrac{A' \Rightarrow B'}{A \Rightarrow B}$.

fort invariant du programme. Dès lors, il est facile de montrer qu'elle conserve toutes les propriétés au sens de B.A. Sanders :

– conservation de l'invariance du plus fort invariant : $\texttt{invariant}_{\mathrm{S}}\ F.SI\ \texttt{in}\ G$

$$\begin{array}{ll} & \quad \{\text{par définition}\} \\ & \texttt{invariant}\ F.SI\ \texttt{in}\ F \\ \Rightarrow & \quad \{F \sqsubseteq_{L_{\mathrm{CM}}} G\} \\ & \texttt{invariant}\ F.SI\ \texttt{in}\ G \\ \Rightarrow & \quad \{\text{relation entre } \textit{invariant} \text{ et } \textit{invariant}_{\mathrm{S}}\} \\ & \texttt{invariant}_{\mathrm{S}}\ F.SI\ \texttt{in}\ G \end{array}$$

Notons que $\texttt{invariant}_{\mathrm{S}}\ F.SI\ \texttt{in}\ G$ peut encore s'écrire $G.SI \Rightarrow F.SI$.

– conservation des propriétés de type *invariant*$_{\mathrm{S}}$:

$$\begin{array}{ll} & \texttt{invariant}_{\mathrm{S}}\ p\ \texttt{in}\ F \\ \equiv & \quad \{\text{par définition}\} \\ & F.SI \Rightarrow p \\ \Rightarrow & \quad \{G.SI \Rightarrow F.SI\} \\ & G.SI \Rightarrow p \\ \equiv & \quad \{\text{par définition}\} \\ & \texttt{invariant}_{\mathrm{S}}\ p\ \texttt{in}\ G \end{array}$$

– conservation des propriétés de type *unless* :

$$\begin{array}{ll} & p\ \texttt{unless}_{\mathrm{S}}\ q\ \texttt{in}\ F \\ \equiv & \quad \{\text{par définition}\} \\ & \langle \forall s : s \in F :: \{F.SI \wedge p \wedge \neg q\}\, s\, \{p \vee q\}\rangle \\ \equiv & \quad \{\texttt{invariant}\ F.SI\ \texttt{in}\ F\} \\ & \langle \forall s : s \in F :: \{F.SI \wedge p \wedge \neg q\}\, s\, \{F.SI \wedge p \vee q\}\rangle \\ \equiv & \quad \{\text{par définition}\} \\ & F.SI \wedge p\ \texttt{unless}\ q\ \texttt{in}\ F \\ \Rightarrow & \quad \{F \sqsubseteq_{L_{\mathrm{CM}}} G\} \\ & F.SI \wedge p\ \texttt{unless}\ q\ \texttt{in}\ G \\ \Rightarrow & \quad \{\text{relation entre } \textit{unless} \text{ et } \textit{unless}_{\mathrm{S}}\} \\ & F.SI \wedge p\ \texttt{unless}_{\mathrm{S}}\ q\ \texttt{in}\ G \\ \Rightarrow & \quad \{\texttt{invariant}_{\mathrm{S}}\ F.SI\ \texttt{in}\ G, \text{ théorème de substitution}\} \\ & p\ \texttt{unless}_{\mathrm{S}}\ q\ \texttt{in}\ G \end{array}$$

– conservation des propriétés de type *ensures* : démonstration similaire à celle de la conservation des propriétés de type *unless*.

Ceci montre le résultat pour la relation de raffinement $\sqsubseteq_L$. Pour la relation $\sqsubseteq_{L'}$, la démonstration est similaire en utilisant le fait que :

$$p\ \mapsto_{\mathrm{S}}\ q\ \texttt{in}\ F\ \equiv\ F.SI \wedge p\ \mapsto\ q\ \texttt{in}\ F$$

On a donc montré les deux relations (4.5).

Il faut noter que les relations inverses ne sont pas valides. Considérons, par exemple, les deux programmes F et G suivants :

Program F	**Program** G
Declare	**Declare**
x, y : Int;	x, y : Int;
Initially	**Initially**
$(x = 0) \wedge (y = 0)$;	$(x = 0) \wedge (y = 0)$;
Assign	**Assign**
skip	$y := 1$ **if** $x = 1$
End F	**End** G

Les programmes F et G ont exactement le même ensemble d'exécutions possibles, c'est-à-dire :

$$\mathcal{O}_1.F = \mathcal{O}_1.G = \{\sigma \mid \langle \forall i :: \sigma_i.x = \sigma_i.y = 0 \rangle\}$$

Par conséquent, on a $F \sqsubseteq_{\mathcal{O}_1} G$ et donc, d'après (4.4), $F \sqsubseteq_{L'_S} G$. Comme le programme F ne vérifie pas de propriétés de type *ensures* autres que les tautologies, on a également $F \sqsubseteq_{L_S} G$.

Par contre, on a trivialement $\texttt{invariant}\ y = 0\ \texttt{in}\ F$, mais on n'a pas $\texttt{invariant}\ y = 0\ \texttt{in}\ G$ puisque le triplet $\{y = 0\}\, y := 1\ \texttt{if}\ x = 1\, \{y = 0\}$ n'est pas valide. Par conséquent, on n'a pas $F \sqsubseteq_{L_{CM}} G$ ni $F \sqsubseteq_{L'_{CM}} G$. Ceci montre que les réciproques des formules (4.5) ne sont pas valides.

Absence d'autres relations

Une observation très intéressante, due à R.T. Udink et J.N. Kok, est qu'aucune implication (autre que la relation (4.4)) n'existe entre les relations de raffinement de propriétés et la relation de raffinement de la sémantique opérationnelle $\mathcal{O}_1$. Plus précisément, on peut montrer par des contre-exemples la non validité de chacune des relations suivantes (le symbole $\not\models$ est utilisé pour rappeler que les relations ne sont pas valides) :

$$\not\models \frac{F \sqsubseteq_{\mathcal{O}_1} G}{F \sqsubseteq_{L_{CM}} G} \quad (4.6) \qquad \not\models \frac{F \sqsubseteq_{\mathcal{O}_1} G}{F \sqsubseteq_{L'_{CM}} G} \quad (4.7) \qquad \not\models \frac{F \sqsubseteq_{\mathcal{O}_1} G}{F \sqsubseteq_{L_S} G} \quad (4.8)$$

$$\not\models \frac{F \sqsubseteq_{L_{CM}} G}{F \sqsubseteq_{\mathcal{O}_1} G} \quad (4.9) \qquad \not\models \frac{F \sqsubseteq_{L'_{CM}} G}{F \sqsubseteq_{\mathcal{O}_1} G} \quad (4.10)$$

$$\not\models \frac{F \sqsubseteq_{L_S} G}{F \sqsubseteq_{\mathcal{O}_1} G} \quad (4.11) \qquad \not\models \frac{F \sqsubseteq_{L'_S} G}{F \sqsubseteq_{\mathcal{O}_1} G} \quad (4.12)$$

On a vu que les relations de raffinement de propriétés au sens de K.M. Chandy et J. Misra étaient plus fortes que leurs pendants au sens de B.A. Sanders (prop. (4.5)). Par conséquent, la non validité de la relation (4.8) entraîne la non validité de la relation (4.6), et la non validité de la relation (4.9) entraîne la non validité de la relation (4.11). D'autre part, on sait également que les relations de raffinement qui conservent les propriétés de type *ensures* sont plus fortes que leur pendants fondées sur les propriétés de type *leads-to* (prop. (4.3)). Donc, la non validité de la relation (4.9) entraîne la non validité de la relation (4.10), et la non validité de la relation (4.11) entraîne la non validité de la relation (4.12). Il ne reste donc à montrer, sur des contre-exemples, que la non validité des relations (4.7), (4.8) et (4.9).

Non validité de la relation (4.7)

Pour montrer la non validité de la relation (4.7), nous pouvons reprendre les programmes F et G suivants :

```
Program F                      Program G
Declare                        Declare
  x, y : Int;                    x, y : Int;
Initially                      Initially
  (x = 0) ∧ (y = 0);             (x = 0) ∧ (y = 0);
Assign                         Assign
  skip                           y := 1  if x = 1
End F                          End G
```

On a vu précédemment que ces programmes satisfaisaient à la relation $F \sqsubseteq_{\mathcal{O}_1} G$ mais pas à la relation $F \sqsubseteq_{L'_{\mathrm{CM}}} G$. Par conséquent, ils constituent un contre-exemple à la proposition (4.7).

Non validité de la relation (4.8)

Pour montrer la non validité de la proposition (4.8), nous pouvons réutiliser les programmes F et G suivants :

```
Program F                      Program G
Assign                         Assign
  x := 0  if x > 0               x := 0
[]                             []
  x := 0  if x < 0               skip
End F                          End G
```

En effet, on a vu au paragraphe 2.1.9 que ces deux programmes avaient la même sémantique opérationnelle, mais qu'ils n'avaient pas les mêmes propriétés de type *ensures*. Comme $F.SI \equiv G.SI \equiv true$, les deux programmes n'ont pas non plus les mêmes propriétés de type $ensures_{\mathrm{S}}$. On a donc, pour ces deux programmes, $G \sqsubseteq_{\mathcal{O}_1} F$ et $G \not\sqsubseteq_{L_{\mathrm{S}}} F$.

Non validité de la relation (4.9)

La non validité la plus surprenante est celle de la relation (4.9). Elle montre que la version "la plus forte" de la logique Unity est insuffisante pour caractériser les séquences d'états des programmes. On peut invalider la proposition à partir des deux programmes suivants :

```
Program F                      Program G
Declare                        Declare
  b : Bool;                      b : Bool;
Initially                      Initially
  b;                             ¬b;
Assign                         Assign
  b := ¬b                        b := ¬b
End F                          End G
```

Les seules propriétés, au sens de K.M. Chandy et J. Misra, vérifiées par le programme F sont b `ensures` $\neg b$, $\neg b$ `ensures` b et toutes leurs conséquences logiques. Ces propriétés sont également vérifiées par G. On a donc $F \sqsubseteq_{L_{\mathrm{CM}}} G$ et $G \sqsubseteq_{L_{\mathrm{CM}}} F$.

Or, $\langle \forall \sigma : \sigma \in \Gamma.\mathcal{O}_1.F :: \sigma_0.b \rangle$ et $\langle \forall \sigma : \sigma \in \Gamma.\mathcal{O}_1.G :: \neg\sigma_0.b \rangle$. Par conséquent, $\Gamma.\mathcal{O}_1.F \cap \Gamma.\mathcal{O}_1.G = \emptyset$ et donc $F \not\sqsubseteq_{\mathcal{O}_1} G$ et $G \not\sqsubseteq_{\mathcal{O}_1} F$.

R.T. Udink et J.N. Kok exhibent un contre-exemple beaucoup plus compliqué qui montre que le problème subsiste pour des programmes Unity avec un espace d'états plus grand et des propriétés plus nombreuses que les programmes F et G précédents. Ils construisent deux programmes vérifiant les mêmes propriétés au sens de K.M. Chandy et J. Misra alors que certaines exécutions de l'un violent la contrainte d'équité faible de l'autre.

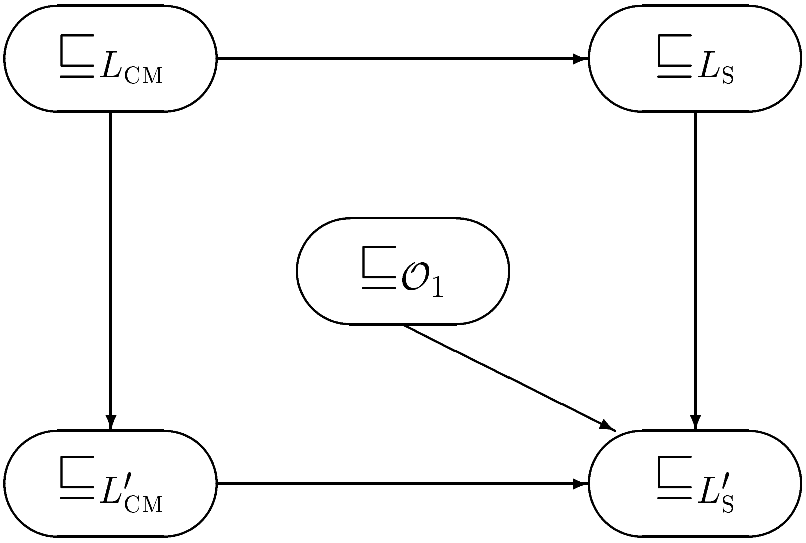

Fig. 4.1 – *Relations entre les formes de raffinement.*

Le diagramme de la figure 4.1 résume les relations existant entre les différentes formes de raffinement de comportement de programmes ou de spécifications Unity présentées dans ce chapitre. Un lien orienté de la relation de raffinement $\sqsubseteq_1$ vers la relation de raffinement $\sqsubseteq_2$ signifie que la relation $\sqsubseteq_1$ est "plus forte" que la relation $\sqsubseteq_2$, c'est-à-dire que pour tous les programmes F et G, on a :

$$\frac{F \sqsubseteq_1 G}{F \sqsubseteq_2 G}$$

Les contre-exemples présentés montrent que les relations qui n'apparaissent pas sur le diagramme et qui ne peuvent pas se déduire par la transitivité des relations du diagramme ne sont pas valides.

4.2.6 Raffinement d'instruction en UNITY

Nous avons présenté au paragraphe 2.3.3 les aspects de la thèse de R.T. Udink liés aux extensions du langage Unity et à la compositionnalité. En fait, ses résultats s'inscrivent dans le cadre plus général du raffinement. En plus des extensions qu'il propose pour le langage (interférence, observabilité, procédures), R.T. Udink présente une forme de raffinement d'instruction Unity et plusieurs théorèmes reliant ce raffinement d'instruction au raffinement de comportement.

Il considère les instructions Unity sous une forme légèrement généralisée (en particulier, l'hypothèse de déterminisme est supprimée) et utilise, comme base de sa notion de raffinement d'instruction, la relation $\sqsubseteq$ que nous avons définie au paragraphe 4.1.2 :

$$s \sqsubseteq s' \;\equiv\; \text{pour tout prédicat } p,\; wp(s,p) \Rightarrow wp(s',p)$$

Ses théorèmes relient alors l'utilisation de cette relation de raffinement avec une notion de raffinement de comportement, proche de $\sqsubseteq_{L'_S}$, mais prenant en considération les interférences possibles entre programmes telles qu'il les a définies (cf § 2.3.3). Parmi ces théorèmes, on retrouve par exemple une version du théorème de renforcement des gardes. Ce théorème est détaillé au paragraphe suivant.

Les "transformeurs" que nous avons présentés au paragraphe 2.3.3 et qui sont utilisés pour représenter les interférences entre les programmes, sont en fait définis à partir de la relation de raffinement $\sqsubseteq$: un programme G peut se composer avec un programme F si et seulement si toutes les instructions de G raffinent les transformeurs spécifiés dans F, et réciproquement.

Nous ne détaillons pas plus cette relation de raffinement d'instruction Unity. Outre le fait que cela nous obligerait à introduire une quantité importante de notations, la notion de raffinement d'instruction n'est pas utilisée dans les chapitres suivants. En effet, les extensions du formalisme Unity présentées aux chapitres 5 et 7 (l'observation et le produit) n'utilisent que des notions de raffinement de comportement.

4.3 Exemples de raffinements en UNITY

Nous présentons dans ce paragraphe deux exemples de transformation de programmes Unity fréquemment utilisées dans le contexte de la répartition de programme : la superposition et le renforcement des gardes. La superposition sert notamment à développer des programmes en couches hiérarchisées, et le renforcement des gardes permet de passer de conditions globales à des conditions locales.

Nous précisons quelles formes de raffinement de comportement sont garanties par ces transformations. En particulier, nous montrons que le théorème de renforcement des gardes constitue un raffinement opérationnel.

4.3.1 La superposition

Comme nous l'avons déjà mentionné aux paragraphes 2.1.6 et 4.2.2, la transformation de programmes UNITY qui consiste à superposer aux instructions existantes de nouvelles affectations *ne modifiant pas les anciennes variables du programme* conserve toutes les propriétés de ce programme. Nous avons vu, qu'en supposant que les variables introduites sont cachées, cette transformation est naturellement une relation de raffinement pour toutes les formes de raffinement de comportement définies dans ce chapitre.

4.3.2 Le renforcement des gardes

L'idée intuitive du renforcement des gardes est de remplacer la garde d'une instruction d'un programme par un prédicat plus fort, c'est-à-dire un prédicat qui implique logiquement cette garde. La nouvelle garde étant plus forte, l'instruction modifiée ne peut changer l'état du programme qu'à partir des états où l'ancienne instruction le pouvait, et dans ce cas la modification est la même. Il est évident qu'une telle transformation conserve les propriétés de sûreté d'un programme.

Dans leur livre sur UNITY, K.M. Chandy et J. Misra suggèrent des conditions sous lesquelles cette transformation préserve également les propriétés de vivacité. On considère un programme UNITY R en distinguant une instruction τ de la forme :

$$\tau : \bar{v} := \bar{e} \texttt{ if } p$$

On note $R \setminus \tau$ le programme ayant les mêmes instructions que R sauf l'instruction τ, et on note R' le transformé de R par renforcement de la garde de l'instruction τ. On a donc :

$$R'.assign = (R \setminus \tau).assign \cup \{\tau' : \bar{v} := \bar{e} \texttt{ if } p \wedge q\}$$

Si les propriétés suivantes sont vérifiées :

$$p \;\mapsto\; q \texttt{ in } R \setminus \tau \qquad \wedge \qquad q \texttt{ unless } \neg p \texttt{ in } R \setminus \tau$$

alors on a la relation $R \sqsubseteq R'$.

Il faut remarquer que la propriété *leads-to* et la propriété *unless* de l'hypothèse s'appliquent à $R \setminus \tau$ et non à R complet. En effet, si c'était le cas, la propriété *leads-to* pourrait provenir de l'instruction modifiée et donc ne plus être satisfaite après le renforcement de la garde. Le théorème ne serait donc pas valide. Par exemple, un programme UNITY dont l'unique instruction est $q := p \texttt{ if } p$ vérifie les deux propriétés $p \;\mapsto\; q$ et $q \texttt{ unless } \neg p$. Par contre, le programme modifié, dont l'unique instruction est alors $q := p \texttt{ if } p \wedge q$ ne vérifie plus la propriété $p \;\mapsto\; q$.

Dans cet énoncé du théorème de renforcement des gardes, nous n'avons pas précisé le type de raffinement choisi ni la sémantique des opérateurs *leads-to* et *unless*.

A.K. Singh montre ce théorème dans les conditions suivantes. Pour les programmes R et R' précédents, on a :

$$\frac{p \ \mapsto_{\mathrm{S}} \ q \ \mathtt{in}\ (R \setminus \tau)[\tau] \ \wedge \ q \ \mathtt{unless}_{\mathrm{S}} \ \neg p \ \mathtt{in}\ (R \setminus \tau)[\tau]}{R \sqsubseteq_{L'_{\mathrm{S}}} R'}$$

La syntaxe "$\mathtt{in}\ (R\setminus\tau)[\tau]$" signifie que les propriétés spécifiées sont des propriétés des instructions du programme $R \setminus \tau$ avec le plus fort invariant du programme $(R \setminus \tau) [\!] \tau$, c'est-à-dire de R. Cette précaution est nécessaire étant donné que la logique UNITY de B.A. Sanders n'est pas compositionnelle (cf § 2.2.3).

La notion de raffinement utilisée par A.K. Singh n'est pas tout à fait la relation $\sqsubseteq_{L'_{\mathrm{S}}}$. Bizarrement, il n'impose que la conservation des propriétés de type *unless* et *leads-to*, et pas celle des invariants. Néanmoins, puisque dans le cas de cette transformation $R.init \equiv R'.init$, la propriété (4.2) s'applique, ce qui permet de déduire le théorème pour la relation $\sqsubseteq_{L'_{\mathrm{S}}}$.

La démonstration de A.K. Singh, que nous ne reprenons pas ici, s'adapte facilement si la sémantique des propriétés est celle de K.M. Chandy et J. Misra. Dans ces conditions, on en déduit une relation de raffinement de la forme $\sqsubseteq_{L'_{\mathrm{CM}}}$. Il est également facile de voir que le renforcement des gardes ne conserve pas, en général, les propriétés de type *ensures*. Considérons les deux programmes F et G suivants :

Program `F`	**Program** `G`
Declare	**Declare**
`x : Int;`	`x : Int;`
`b : Bool;`	`b : Bool;`
Initially	**Initially**
$(\mathtt{x} = 0) \wedge \neg\mathtt{b}$;	$(\mathtt{x} = 0) \wedge \neg\mathtt{b}$;
Assign	**Assign**
$\mathtt{x} := 1$	$\mathtt{x} := 1$ $\boxed{\textbf{if}\ \mathtt{b}}$
$[\!]$	$[\!]$
$\mathtt{b} := true$	$\mathtt{b} := true$
End `F`	**End** `G`

Le théorème de renforcement des gardes s'applique avec $p \equiv true$ et $q \equiv b$. En effet, le programme G se déduit bien du programme F par cette transformation, et on a bien :

$$true \ \mapsto \ b \ \mathtt{in}\ F \qquad \wedge \qquad b \ \mathtt{unless}\ false \ \mathtt{in}\ F$$

pour toutes les sémantiques des opérateurs. On en déduit $F \sqsubseteq_{L'_{\mathrm{CM}}} G$ et $F \sqsubseteq_{L'_{\mathrm{S}}} G$. Par contre, les propriétés $x \neq 1$ $\mathtt{ensures}$ $x = 1$ et $x \neq 1$ $\mathtt{ensures}_{\mathrm{S}}$ $x = 1$ ne sont valides que dans F. On n'a donc pas $F \sqsubseteq_{L_{\mathrm{CM}}} G$ ni $F \sqsubseteq_{L_{\mathrm{S}}} G$[5].

La dernière relation de raffinement à considérer est $\sqsubseteq_{\mathcal{O}_1}$. Il est naturel dans ce cas d'associer aux opérateurs de la logique leur sémantique fondée sur les séquences d'états. Malheureusement, cette sémantique n'est pas compositionnelle et il n'y a pas de moyen simple de contourner la difficulté, comme dans le cas de la sémantique de B.A. Sanders. On a vu également que l'on ne pouvait pas faire les deux hypothèses *leads-to* et *unless* sur le programme R complet. Que

5. Le principe de la preuve présentée par A.K. Singh repose sur le fait qu'aux propriétés de type *ensures* de F correspondent des propriétés de type *leads-to* dans G. En effet, on a bien, dans l'exemple, $x \neq 1 \ \mapsto \ x = 1$ $\mathtt{in}$ G et $x \neq 1 \ \mapsto_{\mathrm{S}} \ x = 1$ $\mathtt{in}$ G.

les hypothèses se fassent sur R ou sur $R \setminus \tau$, le théorème n'est donc pas valide dans le cas de la relation $\sqsubseteq_{\mathcal{O}_1}$.

Par contre, on peut montrer, pour les programme R et R' précédents, le résultat suivant :

$$\frac{p \;\mapsto_{\mathcal{O}_1}\; q \;\texttt{in}\; R' \;\wedge\; q \;\texttt{unless}_{\mathcal{O}_1}\; \neg p \;\texttt{in}\; R'}{R \sqsubseteq_{\mathcal{O}_1} R'} \tag{4.13}$$

En d'autres termes, les hypothèses sont faites sur le programme transformé. Il suffit donc de démontrer les deux propriétés sur le nouveau programme pour garantir le raffinement.
Démonstration :
La preuve se construit en deux étapes. On montre d'abord que les exécutions de $\mathcal{O}_1.R'$ peuvent être obtenues à partir des instructions de R. On montre ensuite qu'à condition d'ajouter du bégaiement, ces exécutions satisfont aux contraintes d'équité de R.

- à une exécution quelconque $\sigma' \in \mathcal{O}_1.R'$, on associe l'exécution σ'' construite par récurrence de la façon suivante :
  - $\sigma''_0 = \sigma'_0$;
  - si $\sigma'_i \xrightarrow{s} \sigma'_{i+1}$ avec $s \neq \tau'$, on choisit $\sigma''_{i+1} = \sigma'_{i+1}$ et on a $\sigma''_i \xrightarrow{s} \sigma''_{i+1}$;
  - si $\sigma'_i \xrightarrow{\tau'} \sigma'_{i+1}$ avec $\sigma'_i.(p \Rightarrow q)$, on choisit $\sigma''_{i+1} = \sigma'_{i+1}$ et on a $\sigma''_i \xrightarrow{\tau} \sigma''_{i+1}$ (les instructions τ et τ' sont équivalentes dans l'état σ'_i) ;
  - si $\sigma'_i \xrightarrow{\tau'} \sigma'_{i+1}$ avec $\sigma'_i.(p \wedge \neg q)$, on choisit $\sigma''_{i+1} = \sigma'_{i+1}$ et on a $\sigma''_i \xrightarrow{skip} \sigma''_{i+1}$ (les instructions *skip* et τ' sont équivalentes dans l'état σ'_i).

  Par construction, on obtient une séquence σ'' dont toutes les transitions sont étiquetées par des instructions du programme R et telle que $\sigma'' = \sigma'$.

- la séquence $\sigma' \in \mathcal{O}_1.R'$ satisfait la contrainte d'équité faible de R' (toutes les instructions de R' sont exécutées une infinité de fois dans σ'). Par conséquent, la séquence σ'' contient une infinité d'exécutions des instructions autres que τ.

  Par contre, il est possible que σ'' ne contienne qu'un nombre fini d'exécutions de l'instruction τ (voir le contre-exemple qui suit). Deux cas sont possibles :

  - σ' vérifie :
  $$\langle \exists i_0 :: \langle \forall i : i \geqslant i_0 :: \sigma'_i.p \rangle\rangle$$
  (le prédicat p reste continûment vrai à partir d'un certain rang). Dans ce cas, des propriétés $p \;\mapsto_\sigma\; q \;\texttt{in}\; \sigma'$ et $q \;\texttt{unless}_\sigma\; \neg p \;\texttt{in}\; \sigma'$, on déduit :
  $$\langle \exists j_0 : j_0 \geqslant i_0 :: \langle \forall j : j \geqslant j_0 :: \sigma'_j.q \rangle\rangle$$
  (le prédicat q reste continûment vrai à partir d'un certain rang), et donc :
  $$\langle \exists j_0 : j_0 \geqslant i_0 :: \langle \forall j : j \geqslant j_0 :: \sigma'_j.(p \Rightarrow q) \rangle\rangle$$

L'instruction τ' étant exécutée une infinité de fois dans la séquence σ' à partir d'états satisfaisant $p \Rightarrow q$, l'instruction τ est exécutée une infinité de fois dans la séquence σ''. Par conséquent, l'exécution σ'' satisfait la contrainte d'équité de R, donc $\sigma'' \in \mathcal{O}_1.R$ et donc $\sigma' \in \mathcal{O}_1.R$.

– σ' vérifie :

$$\langle \forall k : \langle \exists i : i \geqslant k :: \neg \sigma'_i.p \rangle \rangle$$

(le prédicat p est faux infiniment souvent dans σ'). Il existe donc une infinité d'indices i pour lesquels σ'_i (et donc σ''_i) ne vérifie pas p. On peut alors utiliser ces états pour exécuter l'instruction τ sans qu'elle ait d'effet. On construit la séquence σ par récurrence de la façon suivante :

- $\sigma_0 = \sigma''_0 = \sigma'_0$;
- si $\sigma''.p$, alors $\sigma_{i+1} = \sigma''_{i+1}$;
- si $\neg \sigma''_i.p$, alors $\sigma_{i+1} = \sigma_i$ et on a $\sigma_i \xrightarrow{\tau} \sigma_{i+1}$.

Par construction :

- l'instruction τ est exécutée une infinité de fois dans σ ;
- $\sigma \in \mathcal{O}_1.R$;
- $\natural\sigma = \natural\sigma'' = \natural\sigma'$.

Par conséquent, $\sigma' \in \Gamma.\mathcal{O}_1.R$.

Dans les deux cas, on a montré que $\sigma' \in \Gamma.\mathcal{O}_1.R$.

On a donc bien :

$$\mathcal{O}_1.R' \subset \Gamma.\mathcal{O}_1.R$$

D'après la formule (4.1), on en déduit $R \sqsubseteq_{\mathcal{O}_1} R'$.

Le contre-exemple suivant montre qu'il est bien nécessaire de s'abstraire du bégaiement dans la relation de raffinement $\sqsubseteq_{\mathcal{O}_1}$ pour que le résultat précédent soit valide. Considérons les deux programmes F et G suivants :

Program F
Declare
  p, q, r : Bool;
Initially
  ¬p ∧ ¬q ∧ ¬r;
Assign
  p := *true*
⫾
  q := p
⫾
  p := *false* **if** q
⫾
  r := *true* **if** p
End F
	Program G
	Declare
	  p, q, r : Bool;
	Initially
	  ¬p ∧ ¬q ∧ ¬r;
	Assign
	  p := *true*
	⫾
	  q := p
	⫾
	  p := *false* **if** q
	⫾
	  r := *true* **if** p $\boxed{\wedge\ \mathrm{q}}$
	End G

Les trois premières instructions des deux programmes constituent une implantation des deux conditions :

$$p \;\mapsto\; q \qquad \wedge \qquad q \texttt{ unless } \neg p$$

qui sont valides à la fois dans F et dans G. La dernière instruction est l'instruction gardée qui est modifiée lors de la transformation. Le passage du programme F au programme G rentre donc bien dans le cadre du théorème.

On suppose que les instructions sont étiquetées dans l'ordre du texte des programmes : s_1, s_2, s_3 et s_4 pour le programme F, s'_1, s'_2, s'_3 et s'_4 pour le programme G.

Considérons à présent l'exécution du programme G obtenue en exécutant de façon cyclique les instructions de G dans l'ordre suivant : $s'_1 \rightarrow \mathit{skip} \rightarrow s'_4 \rightarrow s'_2 \rightarrow s'_3 \rightarrow s'_2$. On obtient dans ce cas la séquence d'états $\sigma' \in \mathcal{O}_1.G$ suivante (les constantes *true* et *false* sont abrégées respectivement en T et F) :

$$\begin{array}{l}(p,q,r) : (F,F,F) \xrightarrow{s'_1} (T,F,F) \xrightarrow{\mathit{skip}} (T,F,F) \xrightarrow{s'_4} (T,F,F) \xrightarrow{s'_2} (T,T,F) \\ \qquad\qquad\qquad \xrightarrow{s'_3} (F,T,F) \xrightarrow{s'_2} (F,F,F) \cdots\end{array}$$

Cette séquence n'est pas un élément de $\mathcal{O}_1.F$. En effet, la seule façon, dans F, de passer de l'état (F,F,F) à l'état (T,F,F) est d'exécuter l'instruction s_1 ; la seule façon de ne pas modifier l'état (T,F,F) est d'exécuter soit l'instruction *skip*, soit l'instruction s_1, soit l'instruction s_3 ; la seule façon de passer de l'état (T,F,F) à l'état (T,T,F) est d'exécuter l'instruction s_2 ; la seule façon de passer de l'état (T,T,F) à l'état (F,T,F) est d'exécuter l'instruction s_3 ; enfin, la seule façon de passer de l'état (F,T,F) à l'état (F,F,F) est d'exécuter l'instruction s_2. En résumé, la seule façon de construire avec le programme F la séquence σ' est de répéter cycliquement la séquence d'instruction : $s_1 \rightarrow \mathit{skip}|s_1|s_3 \rightarrow s_2 \rightarrow s_3 \rightarrow s_2$. Or, une telle exécution ne satisfait pas la contrainte d'équité du modèle UNITY : l'instruction s_4 n'est jamais exécutée. Par conséquent, $\sigma' \notin \mathcal{O}_1.F$ et donc $\mathcal{O}_1.G \not\subset \mathcal{O}_1.F$.

Néanmoins, on voit que l'on peut insérer des exécutions de l'instruction s_4 à partir, par exemple, de l'état (F,F,F). L'instruction n'a alors aucun effet et n'introduit dans la séquence que du bégaiement supplémentaire. On peut donc retrouver la séquence σ' dans $\Gamma.\mathcal{O}_1.F$.

4.3.3 Exemple d'application

Nous allons montrer, à travers un exemple très simple, comment les deux relations de raffinement présentées au paragraphe précédent peuvent être utilisées pour transformer un programme sans remettre en cause sa correction.

On considère le programme UNITY suivant :

```
Program A
Declare
   x, y : Int;
   l : ListOfInt;
Initially
   (x = 1) ∧ (y = 1);
   (l = nil);
Assign
   x := x + 1
  ||
   l := append (l, ⟨x⟩ 6)
[]
 ⟨ y := head (l)
  ||
   l := tail(l) ⟩ if l ≠ nil
End A
```

Ce programme représente une communication asynchrone entre deux processus. Un premier processus gère un compteur croissant x. Il communique chaque évolution de son compteur à un autre processus. Ce second processus utilise les valeurs reçues pour faire évoluer une variable y qui, par conséquent, est également un compteur croissant. La communication entre les deux processus est réalisée par une variable auxiliaire de type séquence, commune aux deux processus, et qui représente un canal FIFO fiable et non borné.

Les variables externes du programme sont les variables x et y. Seules ces variables apparaissent dans la spécification. La variable l est une variable auxiliaire cachée.

On peut démontrer, sur le programme A, la spécification S suivante :

- $\forall k :: x = k$ $\texttt{unless}_S$ $x > k$: la variable x ne peut que croître (idem pour la variable y) ;

- $\forall k :: x = k$ $\texttt{unless}_S$ $x = k + 1$: la variable x ne peut que croître par pas de un (idem pour la variable y). Cette spécification enrichit en fait la précédente qui en est conséquence logique ;

- $\forall k :: x = k \ \mapsto_S \ x > k$: la variable x finit effectivement par croître ;

- $\forall k :: x = k \ \mapsto_S \ y = k$: toute valeur prise par la variable x est prise ensuite par la variable y ;

- $\texttt{invariant}_S$ $y \leqslant x$: la variable y a toujours une valeur inférieure à celle de la variable x ;

On transforme maintenant le programme A en lui ajoutant une variable

6. La notation $\langle x \rangle$ représente la liste réduite à l'unique élément x.

entière b. On obtient le programme B suivant :

```
Program B
Declare
   x, y, b : Int;
   l : ListOfInt;
Initially
   (x = 1) ∧ (y = 1);
   (l = nil) ∧ (b = 0);
Assign
   x := x + 1
  ||
   l := append (l, ⟨x⟩)
  ||
   b := x
[]
 ⟨ y := head (l)
  ||
   l := tail (l)
  ||
   b := 0 ⟩ if l ≠ nil
End B
```

Ce programme dérive du programme A par superposition : l'ajout de la nouvelle variable b ne modifie par les anciennes variables x, y et l du programme. On en déduit $A \sqsubseteq_{L'_{\mathrm{S}}} B$.

On transforme ensuite le programme par un premier renforcement de garde. On obtient le programme C suivant :

```
Program C
Declare
   x, y, b : Int;
   l : ListOfInt;
Initially
   (x = 1) ∧ (y = 1);
   (l = nil) ∧ (b = 0);
Assign
 ⟨ x := x + 1
  ||
   l := append (l, ⟨x⟩)
  ||
   b := x
[]
 ⟨ y := head (l)
  ||
   l := tail (l)
  ||
   b := 0 ⟩ if  l ≠ nil ∧ b ≠ 0
End C
```

Ce programme, privé de l'instruction dont la garde est renforcée, vérifie trivialement la conditions $x > 0 \wedge l \neq \mathit{nil} \mapsto b \neq 0$ (on a en fait $x > 0 \mapsto b \neq 0$) au sens de K.M. Chandy et J. Misra. Comme $B.\mathit{SI} \Rightarrow x > 0$, on a bien $l \neq \mathit{nil} \mapsto_{\mathrm{S}} b \neq 0$ avec le plus fort invariant du programme B. De la même manière, on a $x > 0 \wedge b \neq 0$ $\texttt{unless}$ false et donc $b \neq 0$ $\texttt{unless}_{\mathrm{S}}$ $l = \mathit{nil}$ avec le plus fort invariant de B. Le théorème de renforcement des gardes s'applique donc. On en déduit $B \sqsubseteq_{L'_{\mathrm{S}}} C$.

On montre alors facilement, sur ce programme C, que :

$$\texttt{invariant}_{\mathrm{S}}\ b \neq 0 \Rightarrow l \neq \mathit{nil}$$

et donc que :

$$\texttt{invariant}_S\ l \neq nil \wedge b \neq 0 \ \equiv\ b \neq 0$$

On peut donc, par substitution, remplacer la garde $l \neq nil \wedge b \neq 0$ dans C par $b \neq 0$[7]. On obtient le programme D suivant, équivalent à C. En particulier, $C \sqsubseteq_{L'_S} D$.

```
Program D
Declare
   x, y, b : Int;
   l : ListOfInt;
Initially
   (x = 1) ∧ (y = 1);
   (l = nil) ∧ (b = 0);
Assign
 ⟨ x := x + 1
 ||
   l := append (l, ⟨x⟩)
 ||
   b := x
[]
 ⟨ y := head (l)
 ||
   l := tail (l)
 ||
   b := 0 ⟩ if   b ≠ 0
End D
```

Un troisième renforcement transforme le programme D pour donner le programme E suivant :

```
Program E
Declare
   x, y, b : Int;
   l : ListOfInt;
Initially
   (x = 1) ∧ (y = 1);
   (l = nil) ∧ (b = 0);
Assign
 ⟨ x := x + 1
 ||
   l := append (l, ⟨x⟩)
 ||
   b := x ⟩ if   b = 0
[]
 ⟨ y := head (l)
 ||
   l := tail (l)
 ||
   b := 0 ⟩ if   b ≠ 0
End E
```

On a bien $true \mapsto b = 0$ (puisque $b \neq 0 \mapsto b = 0$) et $b = 0$ **unless** *false* dans D privé de sa deuxième instruction, au sens de K.M. Chandy et J. Misra (donc au sens de B.A. Sanders avec le plus fort invariant de D). On en déduit $D \sqsubseteq_{L'_S} E$.

On peut alors démontrer :

$$\texttt{invariant}_S\ 0 \leqslant |l| \leqslant 1 \texttt{ in } E$$
$$\texttt{invariant}_S\ b = 0 \equiv l = nil \texttt{ in } E$$
$$\texttt{invariant}_S\ b \neq 0 \Rightarrow head(l) = b \texttt{ in } E$$

7. Il s'agit en fait d'un cas particulier du renforcement de la garde.

Ce dernier invariant permet de remplacer $y := head(l)$ par $y := b$ dans la deuxième instruction du programme E[8]. On obtient le programme F :

```
Program F
Declare
   x, y, b : Int;
   l : ListOfInt;
Initially
   (x = 1) ∧ (y = 1);
   (l = nil) ∧ (b = 0);
Assign
 ⟨ x := x + 1
 ||
   l := append (l, ⟨x⟩)
 ||
   b := x ⟩ if  b = 0
⟦
 ⟨ y := b
 ||
   l := tail (l)
 ||
   b := 0 ⟩ if  b ≠ 0
End F
```

et la relation $E \sqsubseteq_{L'_S} F$.

Enfin, un dernier raffinement permet de supprimer dans le programme F la variable l devenue inutile :

```
Program G
Declare
   x, y, b : Int;
Initially
   (x = 1) ∧ (y = 1);
   (b = 0);
Assign
 ⟨ x := x + 1
 ||
   b := x ⟩ if  b = 0
⟦
 ⟨ y := b
 ||
   b := 0 ⟩ if  b ≠ 0
End G
```

et d'assurer la relation $F \sqsubseteq_{L'_S} G$.

Le programme G implante la communication entre les deux processus au moyen d'un buffer à une case b. Globalement, on est passé d'une communication par canal non borné à une communication par buffer à une case. La relation $A \sqsubseteq_{L'_S} G$, déduite par transitivité du raffinement, permet d'assurer que le programme G satisfait la spécification S.

On peut remarquer que le théorème de renforcement des gardes ne nous assure pas la relation $A \sqsubseteq_{L_S} G$. Par exemple, la propriété

$$\forall k :: x = k \ \texttt{ensures}_S \ x = k + 1$$

est vérifiée par le programme A et pas par le programme G.

Enfin, on peut également noter qu'au cours du raffinement, certaines propriétés supplémentaires ont fait leur apparition. Par exemple, le programme G satisfait la propriété $\texttt{invariant}_S \ 1 \leqslant x - y \leqslant 2$ que le programme A ne vérifiait pas.

8. Il s'agit d'une forme de raffinement de donnée que nous n'avons pas détaillée dans cette partie.

4.4 Bibliographie

Notre présentation, dans le cadre général, du raffinement d'instruction s'appuie sur le livre [MGRV94]. Une approche plus ciblée sur les systèmes réactifs est décrite dans [Bac92]. Le raffinement de comportement que nous présentons est défini dans [AL91]. Cet article définit également les fonctions de raffinement. Il montre en outre que, sous certaines hypothèses, l'ajout de variables auxiliaires permet d'assurer l'existence d'une fonction de raffinement entre deux programmes liés par la relation de raffinement.

Le raffinement de programmes UNITY, considéré comme la conservation d'un ensemble de propriétés logiques, est informellement présenté dans le livre [CM88]. Cette idée est formalisée de différentes manières dans [UK93, Udi95] et dans [Sin93]. Dans les deux cas, les définitions reposent sur l'idée qu'un *programme* qui vérifie une spécification logique se raffine en un *programme* qui vérifie (au moins) la même spécification. Dans [Sin93], une forme de raffinement de point fixe, adaptée aux programmes UNITY qui se terminent, est également définie. La présentation proposée dans ce chapitre diffère de ces définitions par l'utilisation d'une forme de fermeture de spécification qui permet de considérer la relation de raffinement à la fois entre programmes et entre spécifications. Cette étude gère également de manière plus précise le problème des parties visibles et des parties cachées de l'état d'un programme.

Les différentes variations des relations de raffinement de propriétés sont définies dans [UK93]. Cet article étudie également les liens entre le raffinement de comportement opérationnel et les diverses relations de raffinement de propriétés. Le contre-exemple qui montre la non équivalence entre les relations $\sqsubseteq_{\mathcal{O}_1}$ et $\sqsubseteq_{L'_{\mathrm{S}}}$ repose, dans cet article, sur le non respect de la contrainte d'équité faible. Nous présentons un contre-exemple différent, fondé sur un programme UNITY n'admettant pas d'autre invariant que *true*. Cet article considère, en plus des relations que nous définissons, une forme de raffinement opérationnel possédant d'intéressantes propriétés de composition. Cette relation de raffinement n'est pas décrite dans ce chapitre. Par contre, les liens entre les différentes relations de raffinement de propriétés sont simplement cités dans l'article. C'est pourquoi nous avons redémontré ici les relations entre $\sqsubseteq_{L_{\mathrm{CM}}}$ et $\sqsubseteq_{L_{\mathrm{S}}}$ et entre $\sqsubseteq_{L'_{\mathrm{CM}}}$ et $\sqsubseteq_{L'_{\mathrm{S}}}$. Nous avons également présenté un contre-exemple pour chacun des autres liens entre les relations de raffinement.

Le raffinement d'instruction en UNITY que nous n'avons pas détaillé, constitue le cœur de la thèse de R.T. Udink [Udi95]. Il y est associé à différentes formes de raffinement de propriétés.

La superposition de programmes UNITY est définie dans le livre [CM88]. Le théorème de renforcement des gardes y apparaît également, sans preuve, sous la forme d'un exercice. Ce théorème est démontré dans [Sin93] pour la logique de B.A. Sanders et une relation de raffinement voisine de $\sqsubseteq_{L'_{\mathrm{S}}}$. Il est également démontré dans [Udi95] pour une autre forme de raffinement, proche également de $\sqsubseteq_{L'_{\mathrm{S}}}$. Nous ajoutons ici la démonstration de ce théorème, sous une forme légèrement modifiée, pour la sémantique opérationnelle des opérateurs logiques et la relation $\sqsubseteq_{\mathcal{O}_1}$. La forme de cette preuve souligne la nécessité, dans certains cas, de raisonner directement sur les séquences d'états d'un programme. Ici, nous y sommes contraints à cause des différences sémantiques entre les relations de raffinement de propriétés et la relation de raffinement opérationnelle. Nous retrouverons une preuve similaire au chapitre 7, due au fait que l'observation est une contrainte appliquée aux exécutions et non aux transitions d'états.

Chapitre 5

L'observation

Le formalisme UNITY est, par définition, très général. Ses concepteurs montrent les possibilités d'expression de UNITY, à la fois théoriques et pratiques, pour un vaste domaine d'applications. Néanmoins, dans le cas particulier des programmes répartis, l'approche proposée reste particulièrement lourde et complexe. Cette difficulté provient essentiellement de l'absence, dans le langage et dans la logique, de moyens spécifiques pour la description de la communication, qui représente une part importante des systèmes répartis.

Dans ce chapitre, nous proposons une approche abstraite de la communication dans le formalisme UNITY. Cette approche repose sur une notion d'observation. La notion d'observation décrit la vision, partielle et retardée, qu'un processus peut avoir d'un autre processus dans une application répartie. Elle constitue une forme de partage de données plus souple que le partage pur et simple des variables et plus abstrait que la description explicite d'une communication par messages. Pour ces raisons, la communication par observation est adaptée à la description et à l'étude des programmes répartis.

L'observation peut prendre deux formes qu'il convient de distinguer. Elle constitue d'abord une sorte d'opérateur temporel exprimant une relation entre des variables ou des expressions d'un programme. Nous parlerons dans ce cas de *relation* d'observation. Cette relation possède des propriétés d'ordre et différents théorèmes permettent de la relier aux opérateurs de la logique UNITY.

D'un autre point de vue, on peut introduire l'observation au niveau du langage de programmation de UNITY. L'observation est alors vue comme une instruction particulière capable de modifier l'état du programme. Nous parlerons dans ce cas de *mécanisme* d'observation. En s'appuyant sur la relation précédente, ce mécanisme permet de décrire de manière abstraite la communication au sein d'un programme réparti.

Nous débutons ce chapitre en rappelant les bases de l'approche généralement suivie pour décrire des programmes répartis en UNITY, et nous précisons ce qui nous semble être source de difficultés dans cette approche. Nous définissons ensuite la relation d'observation et démontrons quelques théorèmes qui lui sont associés. Cette relation est alors utilisée pour définir le mécanisme d'observation. Nous montrons que ce mécanisme conduit à une nouvelle sémantique opérationnelle des programmes UNITY, et qu'il permet de décrire la communication au sein d'un système réparti. Quelques algorithmes répartis classiques sont décrits en utilisant cette forme de communication. Enfin, ce chapitre se termine par un exemple plus complet : un algorithme d'exclusion mutuelle est décrit, spécifié et prouvé à l'aide des observations.

5.1 Les programmes répartis en UNITY

Au chapitre 2, nous avons présenté le formalisme UNITY dans un cadre très général. Lorsque l'on s'intéresse aux programmes réactifs répartis, certains aspects du formalisme ne sont pas utilisés. C'est le cas, par exemple, du prédicat de point fixe et de toutes les notions associées à la terminaison. Par contre, si le formalisme UNITY est bien adapté à l'étude de programmes réactifs, il ne propose rien de spécifique pour les programmes répartis.

Un programme réparti peut être vu comme un ensemble de processus s'exécutant en parallèle et communiquant par émissions et réceptions de messages. Ce type de programme est généralement modélisé, en UNITY, de la façon suivante :

– les instructions d'un processus sont regroupées au sein d'un programme UNITY. Nous appellerons ce type de programme un *processus* UNITY ;

– les différents programmes représentant les processus sont composés par union (cf § 2.1.6) ;

– les seules variables partagées entre des processus distincts sont de type séquence et sont utilisées de façon restreinte : un processus ajoute des valeurs en queue de séquence (opération *append*), un autre processus enlève des valeurs en tête de séquence (opérations *head* et *tail*), et aucune autre opération de modification n'est autorisée sur ces séquences [1].

Cette représentation des programmes répartis permet donc de retrouver le concept habituel d'échange de messages à partir de la notion de variables partagées en UNITY. En effet, les valeurs insérées dans les séquences représentent les messages. La sémantique des opérations *append*, *head* et *tail* assure alors une communication point-à-point FIFO et fiable entre deux processus partageant une variable de type séquence.

On peut considérer certaines variantes de cette approche comme, par exemple, garder l'instruction *append* par un prédicat portant sur le nombre d'éléments présents dans la séquence (représentation d'un canal de communication borné) ou partager les variables de type séquence entre un plus grand nombre de programmes UNITY (plusieurs instructions *append* et/ou plusieurs instructions *tail* et *head*).

1. Par contre, on autorise généralement un certain nombre d'opérations de consultation comme la comparaison d'égalité à la séquence vide (*nil*) ou entre séquences quelconques, ou encore la longueur (c'est-à-dire le nombre d'éléments) d'une séquence.

Cette représentation en UNITY de la communication entre les différents processus d'une application répartie, est la plus couramment utilisée. En particulier, les exemples de programmes répartis du livre de K.M. Chandy et J. Misra s'appuient sur une description de ce type.

Cette méthode présente l'avantage de proposer une description des systèmes répartis en UNITY qui modélise explicitement les échanges de messages. Elle est donc parfaitement conforme à la fois à la réalité de l'implantation de ces systèmes et aux habitudes des concepteurs d'algorithmes répartis.

Néanmoins, la méthode a aussi ses inconvénients. D'abord, l'utilisation de fonctions telles que *append*, *tail* ou *head* étend implicitement la sémantique du langage UNITY. Au lieu des seules affectations gardées, la sémantique des séquences, en tant que type abstrait, est à prendre en compte. Ensuite, et surtout, cette manipulation restreinte de variables de type séquence n'a pas de représentation propre au niveau de la logique UNITY. Les preuves nécessitent alors une traduction en logique UNITY des propriétés des séquences dans ce cadre particulier d'utilisation. Ceci conduit le plus souvent à un accroissement important de la complexité de ces preuves.

En résumé, la représentation en UNITY de la communication par des variables de type séquence complique à la fois l'écriture des programmes, l'écriture des spécifications et les preuves qui leur sont associées. C'est pour cette raison que nous avons été amenés à définir la relation d'observation et à proposer une représentation de la répartition fondée sur cette relation.

5.2 La relation d'observation

5.2.1 Présentation

Le but de la relation d'observation est de fournir une description de la communication entre processus plus abstraite que les échanges explicites de messages. Ces échanges de messages servent à construire, au niveau d'un processus, une vision des autres processus. Cette vision est partielle et ne fait intervenir qu'une partie de l'état d'un ou plusieurs autres processus. Elle est également retardée par rapport à la réalité. L'idée de l'observation est de représenter directement ce type de vision d'un processus par un autre processus sans détailler explicitement un échange de messages.

Plus précisément, une relation d'observation entre une variable image $`v$ et une variable source $v$ exprime que la suite des valeurs de $`v$ est une suite extraite des valeurs prises par v dans leur ordre chronologique. La variable $`v$ n'est pas tenue de prendre successivement toutes les valeurs de $v$, et le retard entre l'évolution de la variable $v$ et l'évolution de la variable $`v$ n'est pas borné a priori.

La relation d'observation représente en fait une relation entre les historiques des variables : l'historique d'une image est une suite extraite de l'historique de sa source. C'est sur cette idée que repose la définition formelle de l'observation.

5.2.2 Les fonctions horloges

La définition de la relation d'observation s'appuie directement sur la notion de *fonction horloge* présentée dans ce paragraphe. Nous supposons qu'il existe un temps idéal discret et totalement ordonné. Dans ce qui suit, nous prenons comme modèle de temps l'ensemble des entiers naturels. Une fonction horloge est alors définie comme une approximation de ce temps idéal. Cette approximation obéit à des contraintes de sûreté et de vivacité. Concrètement, une fonction horloge est une fonction H de l'ensemble des naturels vers les naturels possédant les propriétés suivantes :

– *sûreté 1* : une horloge ne dépasse jamais le temps idéal :

$$\langle \forall t :: \; H.t \leqslant t \rangle$$

– *sûreté 2* : une horloge est monotone, elle ne recule jamais :

$$\langle \forall t :: \; H.t \leqslant H.(t+1) \rangle$$

– *vivacité* : une horloge finit par avancer :

$$\langle \forall t :: \; \langle \exists t' :: \; H.t' > H.t \rangle \rangle$$

On considère les fonctions horloges comme un type et on note $Horloge(H)$ pour exprimer que la fonction H est une fonction horloge.

Les propriétés suivantes des fonctions horloges sont élémentaires et seront utilisées dans les preuves des théorèmes du paragraphe 5.2.4 :

– une fonction horloge a pour valeur initiale 0 ;

– la fonction identité est une fonction horloge et peut être vue comme l'horloge idéale ;

– la composition de deux fonctions horloges est une fonction horloge.

5.2.3 Définition de la relation d'observation

La relation d'observation est définie à partir d'une sémantique opérationnelle quelconque des programmes UNITY, comme par exemple la sémantique $\mathcal{O}_1$ définie au paragraphe 2.1.9. La relation d'observation entre deux variables ‘v et v d'un programme UNITY F est notée ‘$v \prec v$ `in` F. Pour une sémantique opérationnelle $\mathcal{O}$, elle se définit de la manière suivante :

$$`v \prec v \texttt{ in } F \equiv \langle \forall \sigma : \sigma \in \mathcal{O}.F :: \langle \exists H : Horloge(H) :: \langle \forall t :: \sigma_t.`v = \sigma_{H.t}.v \rangle \rangle \rangle$$

Ceci exprime bien que toute valeur de ‘v est une valeur antérieure de v (propriété de sûreté 1 des fonctions horloges), que ‘v prend les valeurs de v dans l'ordre chronologique (propriété de sûreté 2 des fonctions horloges), et que ‘v prend des valeurs de plus en plus récentes de v (propriété de vivacité des fonctions horloges).

La variable v est appelée *variable observée* ou *variable source* ou encore, en abrégé, *source* de la relation d'observation. La variable $`v$ est appelée *variable image* ou plus simplement *image* de la relation.

Si, pour une relation d'observation donnée, il existe une fonction horloge satisfaisant la condition supplémentaire $\langle \forall t : t > 0 :: H.t < t \rangle$, l'observation est dite *stricte*. Dans une relation d'observation stricte, la variable image ne peut pas prendre en compte immédiatement un changement de valeur de la variable source.

Dans la définition précédente, la relation d'observation porte sur les variables d'un programme. En fait, cette définition se généralise directement à toute expression définie sur les variables d'un programme ou encore à toute expression définie sur l'état d'un programme. Dans ce contexte, la formule

$$`e \prec e \ \ \texttt{in} \ \ F$$

exprime que la valeur de l'expression $`e$ est toujours une valeur antérieure de l'expression $e$, que l'expression $`e$ prend (certaines) des valeurs de l'expression e dans l'ordre chronologique, et que l'expression $`e$ est finalement mise à jour avec une valeur plus récente de l'expression e.

Cette généralisation nous permet notamment de définir l'observation d'un n-uplet de variables. Une telle relation est notée :

$$(`v_1, `v_2, \cdots, `v_n) \prec (v_1, v_2, \cdots, v_n) \ \texttt{in} \ F$$

Elle exprime que les variables $`v_i$ observent *simultanément* les variables v_i, c'est-à-dire avec une même fonction horloge :

$$\begin{gathered}(`v_1, `v_2, \cdots, `v_n) \prec (v_1, v_2, \cdots, v_n) \ \texttt{in} \ F \\ \equiv \\ \langle \forall \sigma : \sigma \in \mathcal{O}.F :: \langle \exists H : Horloge(H) :: \langle \forall t :: \langle \forall i : 1 \leqslant i \leqslant n :: \sigma_t.`v_i = \sigma_{H.t}.v_i \rangle \rangle \rangle \rangle\end{gathered}$$

Cette généralisation de la relation d'observation à des expressions quelconques portant sur les variables d'état permet également de considérer la relation d'observation comme un nouvel opérateur de la logique UNITY. En effet, on peut appliquer la relation d'observation à des expressions *booléennes* des variables, c'est-à-dire à des prédicats d'état. On obtient ainsi un nouvel opérateur temporel $\prec_{\mathcal{O}}$ de la logique UNITY pour chaque sémantique opérationnelle. Lorsqu'il n'y a pas d'ambiguïté sur la sémantique opérationnelle choisie, ou lorsque celle-ci est quelconque et volontairement indéterminée, on note simplement l'opérateur obtenu $\prec$.

Dans le paragraphe suivant, nous présentons des théorèmes reliant ce nouvel opérateur aux autres opérateurs de la logique UNITY.

5.2.4 Propriétés

Nous considérons deux types de propriétés relatives à la relation d'observation : d'une part, des propriétés de la relation d'observation elle-même, et d'autre part, des propriétés reliant l'observation aux opérateurs de la logique UNITY. Ces propriétés sont énoncées pour des expressions quelconques u, v, w, v_i, $`u$, $`v$, $`v_i$, des prédicats $p$ et $`p$ et un seul programme (pas de règles de composition) qui est généralement sous-entendu.

Propriétés de la relation d'observation

- Une image et sa source ont même valeur dans les états initiaux :

$$\frac{`e \prec e \texttt{ in } F}{\langle \forall \sigma : \sigma \in \mathcal{O}.F :: \sigma_0.`e = \sigma_0.e \rangle} \tag{5.1}$$

  Démonstration : la propriété découle du fait que toute fonction horloge H vérifie $H.0 = 0$.

- La relation d'observation est reflexive, antisymétrique et transitive :

$$\frac{}{u \prec u} \qquad \frac{u \prec v \ \wedge \ v \prec u}{\texttt{invariant}_{\mathcal{O}}\ u = v} \qquad \frac{u \prec v \ \wedge \ v \prec w}{u \prec w}$$

  Démonstration :

  - reflexivité : en choisissant l'identité comme fonction horloge H, on a bien $\sigma_t.u = \sigma_{H.t}.u$ pour tous les états de toutes les exécutions ;
  - transitivité : si $u \prec v$ avec une horloge H et $v \prec w$ avec une horloge H', on peut choisir $H' \circ H$ comme horloge pour déduire la relation $u \prec w$;
  - antisymétrie : par récurrence. Soit $\sigma \in \mathcal{O}.F$. Il existe alors deux fonctions horloges H et H' telles que :

$$\langle \forall t :: \sigma_t.u = \sigma_{H.t}.v \wedge \sigma_t.v = \sigma_{H'.t}.u \rangle$$

    On montre qu'à tout instant t, on a les relations suivantes :

    1. $\sigma_t.u = \sigma_t.v$;
    2. $\langle \forall i : H.t \leqslant i \leqslant t :: \sigma_i.v = \sigma_t.v \rangle$;
    3. $\langle \forall i : H'.t \leqslant i \leqslant t :: \sigma_i.u = \sigma_t.u \rangle$.

    Pour $t = 0$, on a $H.t = H'.t = t = 0$. Les trois relations sont vérifiées. Supposons ces trois relations vérifiées pour l'instant t et montrons qu'elles le sont pour l'instant $t + 1$. Trois cas sont possibles :

    - l'expression u change de valeur :

$$\begin{array}{ll}
 & \sigma_{t+1}.u \neq \sigma_t.u \\
\Rightarrow & \quad \{\sigma_t.u = \sigma_t.v\} \\
 & \sigma_{t+1}.u \neq \sigma_t.v \\
\Rightarrow & \quad \{u \prec v \text{ avec } H\} \\
 & \sigma_{H.(t+1)}.v \neq \sigma_t.v \\
\Rightarrow & \quad \{H.t \leqslant H.(t+1) \leqslant t+1 \\
 & \quad\ \ \wedge\ \langle \forall i : H.t \leqslant i \leqslant t :: \sigma_i.v = \sigma_t.v \rangle\} \\
 & H.(t+1) = t+1 \\
\Rightarrow & \quad \{u \prec v \text{ avec } H \Rightarrow \sigma_{t+i}.u = \sigma_{H.(t+1)}.v\} \\
 & \sigma_{t+1}.u = \sigma_{t+1}.v
\end{array}$$

      De $H.(t + 1) = t + 1$, on déduit trivialement :

$$\langle \forall i : H.(t+1) \leqslant i \leqslant t+1 :: \sigma_i.v = \sigma_{t+1}.v \rangle$$

De $\sigma_{t+1}.u = \sigma_{t+1}.v$, on déduit $\sigma_{t+1}.v \neq \sigma_t.v$, et par un raisonnement analogue au précédent, que $H'.(t+1) = t+1$. On en déduit donc trivialement :

$$\langle \forall i : H'.(t+1) \leqslant i \leqslant t+1 :: \sigma_i.u = \sigma_{t+1}.u \rangle$$

– l'expression v change de valeur : le raisonnement est symétrique.
– ni u, ni v ne changent de valeur. On a alors :

$$\sigma_{t+1}.u = \sigma_t.u = \sigma_t.v = \sigma_{t+1}.v$$

De $H.(t+1) \geqslant H.t$ et $\sigma_{t+1}.v = \sigma_t.v$, on déduit la conservation de la propriété 2 au rang $t+1$. De $H'.(t+1) \geqslant H'.t$ et $\sigma_{t+1}.u = \sigma_t.u$, on déduit la conservation de la propriété 3 au rang $t+1$.

Dans les trois cas, on a bien conservation des propriétés 1, 2 et 3 au rang $t+1$. On en déduit l'antisymétrie de la relation d'observation.

– la relation d'observation est compatible avec l'application de fonction :

$$\frac{u \prec v}{f.u \prec f.v} \quad (5.2) \qquad\qquad \frac{f.u \prec f.v \ \wedge \ f \text{ injective}}{u \prec v} \quad (5.3)$$

Démonstration : soit σ une exécution du programme considéré et H une fonction horloge associée à la relation $u \prec v$.

$$\begin{array}{ll}
& u \prec v \\
\Rightarrow & \quad \{\text{par définition}\} \\
& \langle \forall t :: \sigma_t.u = \sigma_{H.t}.v \rangle \\
\Rightarrow & \quad \{x = y \Rightarrow f.x = f.y\} \\
& \langle \forall t :: f.(\sigma_t.u) = f.(\sigma_{H.t}.v) \rangle \\
\Rightarrow & \quad \{f.(\sigma_t.x) = \sigma_t.(f.x)\} \\
& \langle \forall t :: \sigma_t.(f.u) = \sigma_{H.t}.(f.v) \rangle \\
\Rightarrow & \quad \{\text{par définition}\} \\
& f.u \prec f.v
\end{array}$$

Soit maintenant σ une exécution du programme considéré et H une fonction horloge associée à la relation $f.u \prec f.v$.

$$\begin{array}{ll}
& f.u \prec f.v \\
\equiv & \quad \{\text{par définition}\} \\
& \langle \forall t :: \sigma_t.(f.u) = \sigma_{H.t}.(f.v) \rangle \\
\equiv & \quad \{f.(\sigma_t.x) = \sigma_t.(f.x)\} \\
& \langle \forall t :: f.(\sigma_t.u) = f.(\sigma_{H.t}.v) \rangle \\
\equiv & \quad \{f \text{ injective}\} \\
& \langle \forall t :: \sigma_t.u = \sigma_{H.t}.v \rangle \\
\equiv & \quad \{\text{par définition}\} \\
& u \prec v
\end{array}$$

Remarque : on peut également démontrer la deuxième formule à partir de la première en appliquant la fonction f^{-1} aux deux membres de $f.u \prec f.v$.

– les projections d'une relation d'observation sont des relations d'observation :
$$\frac{(\text{‘}v_1, \text{‘}v_2, \cdots, \text{‘}v_n) \prec (v_1, v_2, \cdots, v_n)}{\forall i : 1 \leqslant i \leqslant N :: \text{‘}v_i \prec v_i}$$
Démonstration : découle trivialement de
$$\sigma_t.(v_1, v_2, \cdots, v_n) = \sigma_{t'}.(v_1, v_2, \cdots, v_n) \ \equiv \ \langle \forall i : 1 \leqslant i \leqslant N :: \sigma_t.v_i = \sigma_{t'}.v_i \rangle$$
Par contre, il faut noter que la réciproque n'est pas vraie. En effet, de plusieurs relations d'observation s'appuyant a priori sur des fonctions horloges différentes, il n'est pas possible de déduire l'existence d'une fonction horloge commune aux différentes relations d'observation.

Relation d'observation et logique UNITY

– Observation de prédicats stables :

$$\frac{\text{‘}p \prec p \ \wedge \ \texttt{stable}_{\mathcal{O}}\ p}{\texttt{stable}_{\mathcal{O}}\ \text{‘}p} \quad (5.4) \qquad\qquad \frac{\text{‘}p \prec p \ \wedge \ \texttt{invariant}_{\mathcal{O}}\ p}{\texttt{invariant}_{\mathcal{O}}\ \text{‘}p} \quad (5.5)$$

Ces deux règles expriment que l'image d'un prédicat stable est un prédicat stable, et que l'image d'un invariant est un invariant.

Démonstration : Soit σ une exécution du programme considéré et H une fonction horloge associée à la relation d'observation $\text{‘}p \prec p$. On considère le théorème (5.4) :

$$\begin{array}{ll}
 & \mathit{true} \\
\Rightarrow & \quad \{\sigma_i.\text{‘}p = \sigma_{H.i}.p\} \\
 & \langle \forall i :: \sigma_i.\text{‘}p \Rightarrow \sigma_{H.i}.p \rangle \\
\Rightarrow & \quad \{\texttt{stable}_\sigma\ p\} \\
 & \langle \forall i :: \sigma_i.\text{‘}p \Rightarrow \langle \forall j : j \geqslant H.i :: \sigma_j.p \rangle\rangle \\
\Rightarrow & \quad \{\text{propriété de sûreté 2 des fonctions horloges}\} \\
 & \langle \forall i :: \sigma_i.\text{‘}p \Rightarrow \langle \forall k : k \geqslant i :: \sigma_{H.k}.p \rangle\rangle \\
\Rightarrow & \quad \{\sigma_k.\text{‘}p = \sigma_{H.k}.p\} \\
 & \langle \forall i :: \sigma_i.\text{‘}p \Rightarrow \langle \forall k : k \geqslant i :: \sigma_k.\text{‘}p \rangle\rangle \\
\Rightarrow & \quad \{\text{par définition de } \mathit{stable}\} \\
 & \texttt{stable}_\sigma\ \text{‘}p
\end{array}$$

Le théorème (5.5) se démontre à partir du théorème (5.4) :

$$\begin{array}{ll}
 & \texttt{invariant}_\sigma\ p \\
\Rightarrow & \quad \{\text{par définition}\} \\
 & \sigma_0.p \ \wedge \ \texttt{stable}_\sigma\ p \\
\Rightarrow & \quad \{\text{théorèmes (5.1) et (5.4)}\} \\
 & \sigma_0.\text{‘}p \ \wedge \ \texttt{stable}_\sigma\ \text{‘}p \\
\Rightarrow & \quad \{\text{par définition}\} \\
 & \texttt{invariant}_\sigma\ \text{‘}p
\end{array}$$

– Détection de prédicats stables par observation :

$$\frac{`p \prec p \ \wedge\ \mathtt{stable}_{\mathcal{O}}\ p}{`p\ \mathtt{detects}_{\mathcal{O}}\ p} \tag{5.6}$$

L'image d'une source stable détecte cette source. On rappelle qu'une propriété *detects* se compose de deux parties :

$$`p\ \mathtt{detects}_{\mathcal{O}}\ p \quad \equiv \quad (\mathtt{invariant}_{\mathcal{O}}\ `p \Rightarrow p)\ \wedge\ (p\ \mapsto_{\mathcal{O}}\ `p)$$

Démonstration : Soit σ une exécution du programme considéré et H une fonction horloge associée à la relation d'observation $`p \prec p$. La preuve de la partie invariante est similaire à la preuve de (5.4) :

$$\begin{array}{ll}
 & \mathit{true} \\
\Rightarrow & \quad \{\sigma_i.`p = \sigma_{H.i}.p\} \\
 & \langle \forall i :: \sigma_i.`p \Rightarrow \sigma_{H.i}.p \rangle \\
\Rightarrow & \quad \{\mathtt{stable}_\sigma\ p\} \\
 & \langle \forall i :: \sigma_i.`p \Rightarrow \langle \forall j : j \geqslant H.i :: \sigma_j.p \rangle \rangle \\
\Rightarrow & \quad \{\text{propriété de sûreté 1 des fonctions horloges} : i \geqslant H.i\} \\
 & \langle \forall i :: \sigma_i.`p \Rightarrow \sigma_i.p \rangle \\
\Rightarrow & \quad \{\text{logique propositionnelle}\} \\
 & \langle \forall i :: \sigma_i.(`p \Rightarrow p) \rangle \\
\Rightarrow & \quad \{\text{par définition}\} \\
 & \mathtt{invariant}_\sigma\ `p \Rightarrow p
\end{array}$$

$$\begin{array}{ll}
 & \mathtt{stable}_\sigma\ p \\
\Rightarrow & \quad \{\text{par définition}\} \\
 & \langle \forall i :: \sigma_i.p \Rightarrow \langle \forall j : j \geqslant i :: \sigma_j.p \rangle \rangle \\
\Rightarrow & \quad \{\text{sûreté 1 et vivacité des horloges} : \langle \forall i :: \langle \exists k : k \geqslant i :: H.k \geqslant i \rangle \rangle\} \\
 & \langle \forall i :: \langle \exists k : k \geqslant i :: \sigma_i.p \Rightarrow \langle \forall j : j \geqslant H.k :: \sigma_j.p \rangle \rangle \rangle \\
\Rightarrow & \quad \{\text{logique des prédicats et } j = H.k\} \\
 & \langle \forall i :: \sigma_i.p \Rightarrow \langle \exists k : k \geqslant i :: \sigma_{H.k}.p \rangle \rangle \\
\Rightarrow & \quad \{\sigma_{H.k}.p = \sigma_k.`p\} \\
 & \langle \forall i :: \sigma_i.p \Rightarrow \langle \exists k : k \geqslant i :: \sigma_k.`p \rangle \rangle \\
\Rightarrow & \quad \{\text{par définition}\} \\
 & p\ \mapsto_\sigma\ `p
\end{array}$$

– Vivacité des images :

$$\frac{`p \prec p \ \wedge\ p\ \mathtt{unless}_{\mathcal{O}}\ `p}{p\ \mapsto_{\mathcal{O}}\ `p} \tag{5.7}$$

$$\frac{`p \prec p \ \wedge\ \mathtt{stable}_{\mathcal{O}}\ p}{p\ \mapsto_{\mathcal{O}}\ `p} \tag{5.8}$$

Ces théorèmes expriment que si une source “attend” son image (ou éventuellement attend indéfiniment), cette image finit par la rejoindre.

Démonstration : Soit σ une exécution du programme considéré, i un entier naturel et H une fonction horloge associée à la relation d'observation $`p \prec p$.

$$
\begin{array}{ll}
 & \sigma_i.p \\
\Rightarrow & \{\text{sûreté 1 et vivacité des horloges}\} \\
 & \langle \exists k : k \geqslant i :: H.k \geqslant i \rangle \\
\Rightarrow & \{\text{logique propositionnelle}\} \\
 & \langle \exists k : k \geqslant i \wedge H.k \geqslant i :: \sigma_{H.k}.p \vee \neg \sigma_{H.k}.p \rangle \\
\Rightarrow & \{\sigma_{H.k}.p = \sigma_k.`p\} \\
 & \langle \exists k : k \geqslant i \wedge H.k \geqslant i :: \sigma_k.`p \vee \neg \sigma_{H.k}.p \rangle \\
\Rightarrow & \{i \leqslant H.k \wedge \sigma_i.p \wedge \neg \sigma_{H.k}.p\} \\
 & \langle \exists k : k \geqslant i :: \sigma_k.`p \vee \langle \exists j : j \geqslant i :: \sigma_j.p \wedge \neg \sigma_{j+1}.p \rangle \rangle \\
\Rightarrow & \{\text{logique propositionnelle}\} \\
 & \langle \exists k : k \geqslant i :: \sigma_k.`p \rangle \vee \langle \exists j : j \geqslant i :: \sigma_j.`p \vee (\neg \sigma_j.`p \wedge \sigma_j.p) \wedge \neg \sigma_{j+1}.p \rangle \\
\Rightarrow & \{p \ \mathtt{unless}_{\mathcal{O}} \ `p\} \\
 & \langle \exists k : k \geqslant i :: \sigma_k.`p \rangle \vee \langle \exists j : j \geqslant i :: \sigma_j.`p \vee (\sigma_{j+1}.p \vee \sigma_{j+1}.`p) \wedge \neg \sigma_{j+1}.p \rangle \\
\Rightarrow & \{\text{logique propositionnelle}\} \\
 & \langle \exists k : k \geqslant i :: \sigma_k.`p \rangle \vee \langle \exists j : j \geqslant i :: \sigma_j.`p \vee \sigma_{j+1}.`p \rangle \\
\Rightarrow & \{\text{logique des prédicats}\} \\
 & \langle \exists l : l \geqslant i :: \sigma_l.`p \rangle
\end{array}
$$

Ceci démontre le théorème (5.7). Le théorème (5.8) peut alors être vu comme une conséquence de ce théorème ($\mathtt{stable}\ p \Rightarrow \forall q :: p\ \mathtt{unless}\ q$) ou comme une conséquence du théorème (5.6) sur la propriété *detects*.

– Observation de compteurs :

$$\frac{`u \prec u \ \wedge \ \forall k :: \mathtt{stable}_{\mathcal{O}}\ u \geqslant k}{\forall k :: \mathtt{stable}_{\mathcal{O}}\ `u \geqslant k} \tag{5.9}$$

$$\frac{`u \prec u \ \wedge \ \forall k :: \mathtt{stable}_{\mathcal{O}}\ u \geqslant k}{\mathtt{invariant}_{\mathcal{O}}\ `u \leqslant u} \tag{5.10}$$

$$\frac{`u \prec u \ \wedge \ \forall k :: \mathtt{stable}_{\mathcal{O}}\ u \geqslant k}{\forall l :: u = l \ \mapsto_{\mathcal{O}} \ `u \geqslant l} \tag{5.11}$$

La proposition de logique UNITY $\forall k :: \mathtt{stable}_{\mathcal{O}}\ u \geqslant k$ signifie que u est un compteur croissant. Le théorème (5.9) exprime que l'image d'un compteur croissant est un compteur croissant. Le théorème (5.10) exprime que l'image d'un compteur croissant est toujours inférieure à sa source. Enfin, le théorème (5.11) montre que toute valeur atteinte par un compteur croissant sera atteinte ou dépassée par chacune des images du compteur [2].

2. Les trois théorèmes sont également valables pour des compteurs décroissants en transposant les relations d'ordre. En fait, les résultats sont valables pour n'importe quelle relation d'ordre, même partielle.

Démonstration:

$$\begin{array}{ll}
 & {}^{\text{‘}}u \prec u \ \wedge\ \forall k :: \texttt{stable}_{\mathcal{O}}\ u \geqslant k \\
\Rightarrow & \quad \{\text{théorème d'application de fonction (5.2) avec } f_k : x \mapsto x \geqslant k\} \\
 & ({}^{\text{‘}}u \geqslant k) \prec (u \geqslant k) \ \wedge\ \forall k :: \texttt{stable}_{\mathcal{O}}\ u \geqslant k \\
\Rightarrow & \quad \{\text{théorème de conservation de la stabilité (5.4)}\} \\
 & \forall k :: \texttt{stable}_{\mathcal{O}}\ {}^{\text{‘}}u \geqslant k
\end{array}$$

Pour le théorème (5.10), on considère une exécution σ et une fonction horloge H associée à la relation d'observation ${}^{\text{‘}}u \prec u$.

$$\begin{array}{ll}
 & \forall k :: \texttt{stable}_{\sigma}\ u \geqslant k \\
\Rightarrow & \quad \{\text{par définition}\} \\
 & \langle \forall k :: \langle \forall i :: \sigma_i.u \geqslant k \Rightarrow \langle \forall j : j \geqslant i :: \sigma_j.u \geqslant k \rangle\rangle\rangle \\
\Rightarrow & \quad \{\text{pour } k = \sigma_i.u\} \\
 & \langle \forall i :: \sigma_i.u \geqslant \sigma_i.u \Rightarrow \langle \forall j : j \geqslant i :: \sigma_j.u \geqslant \sigma_i.u \rangle\rangle\rangle \\
\Rightarrow & \{\text{logique des prédicats}\} \\
 & \langle \forall j :: \langle \forall i : i \leqslant j :: \sigma_i.u \leqslant \sigma_j.u \rangle\rangle \\
\Rightarrow & \quad \{\text{pour } i = H.j \text{ (on a bien } H.j \leqslant j)\} \\
 & \langle \forall j :: \sigma_{H.j}.u \leqslant \sigma_j.u \rangle \\
\Rightarrow & \quad \{\sigma_{H.j}.u = \sigma_j.{}^{\text{‘}}u\} \\
 & \langle \forall j :: \sigma_j.{}^{\text{‘}}u \leqslant \sigma_j.u \rangle \\
\Rightarrow & \quad \{\text{par définition}\} \\
 & \texttt{invariant}_{\sigma}\ {}^{\text{‘}}u \leqslant u
\end{array}$$

$$\begin{array}{ll}
 & \texttt{stable}_{\mathcal{O}}\ u \geqslant k \\
\Rightarrow & \quad \{\text{définition de } \mathit{stable}_{\mathcal{O}}\} \\
 & u \geqslant k\ \texttt{unless}_{\mathcal{O}}\ \mathit{false} \\
\Rightarrow & \quad \{\text{affaiblissement rhs } \mathit{unless}_{\mathcal{O}}\} \\
 & u \geqslant k\ \texttt{unless}\ {}^{\text{‘}}u \geqslant k \\
\Rightarrow & \quad \{{}^{\text{‘}}u \prec u, \text{ théorème (5.2) avec } f_k : x \mapsto x \geqslant k\} \\
 & u \geqslant k\ \texttt{unless}\ {}^{\text{‘}}u \geqslant k \ \wedge\ ({}^{\text{‘}}u \geqslant k) \prec (u \geqslant k) \\
\Rightarrow & \quad \{\text{théorème (5.7)}\} \\
 & u \geqslant k \ \mapsto_{\mathcal{O}}\ {}^{\text{‘}}u \geqslant k \\
\Rightarrow & \quad \{\text{renforcement lhs } \mathit{leads\text{-}to}_{\mathcal{O}}\} \\
 & u = k \ \mapsto_{\mathcal{O}}\ {}^{\text{‘}}u \geqslant k
\end{array}$$

– Observation de et par des constantes :

$$\begin{array}{lll}
p \prec \mathit{true} \ \equiv\ \texttt{invariant}_{\mathcal{O}}\ p & \qquad & p \prec \mathit{false} \ \equiv\ \texttt{invariant}_{\mathcal{O}}\ \neg p \\
\mathit{true} \prec p \ \equiv\ \ \mathit{true}\ \mapsto_{\mathcal{O}}\ p & \qquad & \mathit{false} \prec p \ \equiv\ \ \mathit{true}\ \mapsto_{\mathcal{O}}\ \neg p
\end{array}$$

La première formule exprime que les invariants (resp. les négations d'invariants) sont les seules images de la constante *true* (resp. *false*). L'autre formule signifie que les prédicats dont la constante *true* (resp. *false*) est une image sont vrais (resp. faux) infiniment souvent.

Démonstration :

$$
\begin{array}{ll}
 & p \prec true \\
\equiv & \quad \{\text{par définition}\} \\
 & \langle \forall \sigma :: \langle \exists H : Horloge(H) :: \langle \forall i :: \sigma_i.p = \sigma_{H.i}.true \rangle\rangle\rangle \\
\equiv & \quad \{\sigma_{H.i}.true \equiv true\} \\
 & \langle \forall \sigma :: \langle \forall i :: \sigma_i.p \rangle\rangle \\
\equiv & \quad \{\text{par définition}\} \\
 & \mathtt{invariant}_{\mathcal{O}}\ p
\end{array}
$$

$$
\begin{array}{ll}
 & p \prec false \\
\equiv & \quad \{\text{théorèmes (5.2) et (5.3) avec la fonction (injective) : } x \mapsto \neg x\} \\
 & \neg p \prec true \\
\equiv & \quad \{\text{théorème précédent}\} \\
 & \mathtt{invariant}_{\mathcal{O}}\ \neg p
\end{array}
$$

$$
\begin{array}{ll}
 & true \prec p \\
\Rightarrow & \quad \{\text{par définition}\} \\
 & \langle \forall \sigma :: \langle \exists H : Horloge(H) :: \langle \forall i :: \sigma_i.true = \sigma_{H.i}.p \rangle\rangle\rangle \\
\Rightarrow & \quad \{\sigma_i.true \equiv true\} \\
 & \langle \forall \sigma :: \langle \exists H : Horloge(H) :: \langle \forall i :: \sigma_{H.i}.p \rangle\rangle\rangle \\
\Rightarrow & \quad \{\text{vivacité des fonctions horloges}\} \\
 & \langle \forall \sigma :: \langle \forall i :: \langle \exists k : k \geqslant i :: \sigma_k.p \rangle\rangle\rangle \\
\Rightarrow & \quad \{\text{par définition}\} \\
 & true \ \mapsto_{\mathcal{O}} \ p
\end{array}
$$

$$
\begin{array}{ll}
 & true \ \mapsto_{\mathcal{O}} \ p \\
\Rightarrow & \quad \{\text{par définition}\} \\
 & \langle \forall \sigma :: \langle \forall i :: \langle \exists k : k \geqslant i :: \sigma_k.p \rangle\rangle \\
\Rightarrow & \quad \{H(i) = \max\{k | k \leqslant i \wedge \sigma_k.p\}\ H \text{ est bien définie}\} \\
 & \quad\ \ \text{et vérifie les trois propriétés des fonctions horloges}\} \\
 & \langle \forall \sigma :: \langle \exists H : Horloge(H) :: \langle \forall i :: \sigma_{H.i}.p \rangle\rangle \\
\Rightarrow & \quad \{\sigma_i.true \equiv true\} \\
 & \langle \forall \sigma :: \langle \exists H : Horloge(H) :: \langle \forall i :: \sigma_i.true = \sigma_{H.i}.p \rangle\rangle\rangle \\
\Rightarrow & \quad \{\text{par définition}\} \\
 & true \prec p
\end{array}
$$

$$
\begin{array}{ll}
 & false \prec p \\
\equiv & \quad \{\text{théorèmes (5.2) et (5.3) avec la fonction (injective) : } x \mapsto \neg x\} \\
 & true \prec \neg p \\
\equiv & \quad \{\text{théorème précédent}\} \\
 & true \ \mapsto_{\mathcal{O}} \ \neg p
\end{array}
$$

On remarquera que dans le cadre des programmes réactifs caractérisés par une interaction avec un environnement, les relations entre l'observation et les prédicats de la logique UNITY peuvent être utilisés de deux façons:

- soit pour déduire des propriétés sur les images à partir de celles qui ont été spécifiées sur l'environnement, par exemple (5.4) ;

– soit de déduire des propriétés de l'environnement à partir de celles des images, par exemple (5.6).

5.2.5 Définition logique et définition opérationnelle

Nous avons défini la relation d'observation directement sur les exécutions des programmes. Ceci nous conduit inévitablement à raisonner assez souvent sur les séquences d'états, comme par exemple pour démontrer la plupart des théorèmes précédents. Pour cette même raison, dans la définition du *mécanisme* d'observation, présentée au paragraphe suivant, nous ne pouvons pas utiliser les notions de composition de P. Collette ou de R.T. Udink et nous devons définir une nouvelle sémantique opérationnelle appropriée. Enfin, nous ne pouvons utiliser que la définition opérationnelle de la logique UNITY.

Il n'est pas évident de trouver un équivalent, en logique UNITY[3], à la définition que nous avons choisie de la relation d'observation. Néanmoins, la relation suivante est intéressante. On considère la relation $\mathcal{R}$ définie par :

$$
\begin{array}{c}
`u\ \mathcal{R}\ u \ \texttt{in}\ F \\
\equiv \\
F.\mathit{init} \Rightarrow (`u = u) \\
\wedge \\
\forall k, `k :: \\
(`u = `k \wedge u = k)\ \texttt{unless}\ (`u \neq `k \vee u \neq k) \wedge (`u = `k \vee `u = k \vee `u = u)\ \texttt{in}\ F \\
\wedge \\
`u = `k \ \mapsto\ `u \neq `k \vee u = `k\ \texttt{in}\ F
\end{array}
$$

Cette relation exprime que les expressions u et $`u$ sont égales initialement et que, au cours d'une transition, $`u$ peut rester inchangé ($`u = `k$), peut prendre la valeur précédente de u ($`u = k$) ou peut prendre une nouvelle valeur, en même temps que $u$ ($`u = u$). La propriété de type *unless* n'assure aucune vivacité et $`u$ pourrait ne jamais évoluer. On ajoute donc la contrainte *leads-to* qui oblige l'expression $`u$ à évoluer tant qu'elle n'est pas égale à u.

Néanmoins, la relation obtenue n'est pas équivalente à la relation d'observation. En effet, lorsque l'expression $`u$ évolue, elle doit prendre la même valeur que $u$ ou la valeur précédente, et non une valeur quelconque entre sa valeur courante et celle de $u$. En d'autres termes, si $`u$ évolue, elle doit combler tout son retard d'un coup, à une valeur près. En conséquence, la relation $\mathcal{R}$ n'est pas transitive.

Par contre, la relation $\mathcal{R}$ est "plus forte" que la relation d'observation. On a :

$$
\frac{`u\ \mathcal{R}\ u\ \texttt{in}\ F}{`u \prec u\ \texttt{in}\ F}
$$

pour la logique UNITY opérationnelle, celle de B.A. Sanders, ou celle de K.M. Chandy et J. Misra.

3. Ni même en logique temporelle linéaire.

Démonstration :

- Étant données les relations entre les différentes sémantiques de la logique UNITY, il suffit de travailler avec l'hypothèse d'un *unless* et d'un *leads-to* en sémantique opérationnelle.

- Soit σ une exécution d'un programme F. On a :

  $$\sigma_0.`u = \sigma_0.u$$

  d'après l'hypothèse sur $F.init$. Soit i un indice quelconque. D'après l'hypothèse *unless*, on a :

  $$\langle \forall k, `k :: \\ \sigma_i.`u = `k \wedge \sigma_i.u = k \Rightarrow \sigma_{i+1}.`u = `k \vee \sigma_{i+1}.`u = k \vee \sigma_{i+1}.`u = \sigma_{i+1}.u \rangle$$

  c'est-à-dire :

  $$\sigma_{i+1}.`u = \sigma_i.`u \ \vee \ \sigma_{i+1}.`u = \sigma_i.u \ \vee \ \sigma_{i+1}.`u = \sigma_{i+1}.u$$

  On construit par récurrence la fonction H suivante :

  - $H.0 = 0$;
  - Si $\sigma_{i+1}.`u = \sigma_i.`u$ alors $H.(i+1) = H.i$;
    sinon, si $\sigma_{i+1}.`u = \sigma_i.u$ alors $H.(i+1) = i$;
    sinon, $\sigma_{i+1}.`u = \sigma_{i+1}.u$ et $H.(i+1) = i+1$.

  On vérifie facilement par récurrence que :

  $$\langle \forall i :: \sigma_i.`u = \sigma_{H.i}.u \rangle$$

  Il reste à montrer que la fonction H satisfait bien aux contraintes d'une fonction horloge. Les deux propriétés de sûreté ne posent aucune difficulté. D'après l'hypothèse *leads-to*, on a :

  $$\begin{array}{ll} & \langle \forall\ `k, i :: \sigma_i.`u = `k \Rightarrow \langle \exists j : j \geqslant i :: \sigma_j.`u \neq `k \vee \sigma_j.u = `k \rangle\rangle \\ \Rightarrow & \{`k = \sigma_i.`u\} \\ & \langle \forall i :: \langle \exists j : j \geqslant i :: \sigma_j.`u \neq \sigma_i.`u \vee \sigma_j.`u = \sigma_j.u \rangle\rangle \\ \Rightarrow & \langle \forall i :: \ \ \langle \exists j' : j' \geqslant i :: \sigma_{j'+1}.`u \neq \sigma_{j'}.`u \rangle \\ & \quad \vee \quad \langle \exists j'' : j'' \geqslant i :: \sigma_{j''+1}.`u = \sigma_{j''+1}.u \rangle\rangle \\ \Rightarrow & \quad \{\text{cas 2 ou 3 de la construction de } H\} \\ & \langle \forall i : \langle \exists j''' : j''' \geqslant i :: H.(j'''+1) \geqslant j''' \rangle\rangle \\ \Rightarrow & \langle \forall i : \langle \exists j'''' : j'''' \geqslant i :: H.j'''' \geqslant i \rangle\rangle \end{array}$$

  ce qui montre la propriété de vivacité de la fonction H et donc la relation $`u \prec u$.

La relation $\mathcal{R}$ peut donc fournir un moyen de démontrer une relation d'observation de manière axiomatique dans le cas particulier où l'image se recale sur la valeur courante de la source.

5.3 Le mécanisme d'observation

5.3.1 La section *observe*

Dans la partie précédente, nous avons défini l'observation comme une relation entre variables ou expressions d'un programme UNITY. Cette relation peut être considérée comme un nouvel opérateur de la logique UNITY, dont la sémantique est définie opérationnellement. Nous avons montré quelques théorèmes reliant cet opérateur d'observation aux autres opérateurs de la logique.

Nous allons maintenant montrer comment la relation d'observation peut être utilisée pour définir un *mécanisme* permettant de réaliser une forme abstraite de communication entre différents processus d'un système réparti.

Ce mécanisme d'observation constitue une partie de la description d'un programme. Cette partie est introduite par le mot-clé *observe*. Elle est constituée d'une conjonction de relations d'observation que le programme suppose valides. Il est donc de la responsabilité de l'environnement du programme de réaliser effectivement ces relations. La section *observe* remplace alors une implantation explicite quelconque des relations qu'elle contient, y compris l'initialisation des images en fonction de celle des sources.

Les relations d'observation d'une section *observe* doivent respecter certaines contraintes de façon à ce que leur réalisation par l'environnement du système soit élémentaire, y compris sur une architecture répartie :

- les membres gauches[4] des relations ne sont que des variables ou des n-uplets de variables, et non des expressions quelconques ;

- les variables images ne sont modifiées *que* par le mécanisme d'observation. En particulier, elles ne doivent pas apparaître en partie gauche d'une affectation ;

- une variable ne doit être image que dans une relation d'observation au plus. Elle peut par contre être à la fois image dans une relation et source dans une ou plusieurs autres relations.

Par exemple, $x \prec y \ \wedge \ y \prec z$ constitue une section *observe* valide (sous l'hypothèse que ni x, ni y ne sont affectées par le programme), mais pas $x \prec y \ \wedge \ x \prec z$.

5.3.2 Existence d'une implantation pour une section *observe*

Sous ces hypothèses, on peut facilement exhiber différents modèles UNITY d'une section *observe*. Par exemple, les programmes *Obs* et *ObsRep* suivant

4. Dans l'utilisation que nous ferons du mécanisme d'observation, les membres droits seront aussi des variables ou des n-uplets.

constituent deux implantations correctes de la relation d'observation $`v \prec v$:

```
Program Obs
Declare
   ‘v : T⁵;
Initially
   ‘v = v;
Assign
   ‘v := v
End Obs
```

```
Program ObsRep
Declare
   ‘v : T;
   l : List Of T;
Initially
   (‘v = v) ∧ (l = nil);
Assign
   l := append (l, ⟨v⟩)
⫾
   ⟨ ‘v := head (l)
  ||
     l := tail (l) ⟩ if l ≠ nil
End ObsRep
```

Le programme *Obs* constitue l'implantation la plus simple d'une relation d'observation. Le programme *ObsRep* peut être vu comme une implantation répartie en ce sens que si les variables $`v$ et $v$ sont situées sur deux processus différents, on peut ajouter la première instruction au processus de $v$ et la seconde au processus de $`v$ sans que les processus ne partagent d'autres variables qu'une séquence, ce qui modélise une communication par messages.

Dire que ces deux programmes constituent une implantation de la relation $`v \prec v$ consiste à démontrer que $`v \prec v$ `in` $Obs \| F$ et $`v \prec v$ `in` $ObsRep \| F$ pour tout programme $F$ ne modifiant ni la variable $`v$ ni la variable l.

Démonstration : le programme *Obs* vérifie trivialement :

$$\forall `k :: `v = `k \texttt{ ensures } `v = v$$

dans la logique de K.M. Chandy et J. Misra. D'après la stabilité du prédicat $`v = `k$ dans F (le programme F ne modifie pas la variable $`v$), cette propriété *ensures* est également valide dans $Obs \| F$. On en déduit que le programme $Obs \| F$ vérifie la propriété $`v \; \mathcal{R} \; v$ où $\mathcal{R}$ est la relation définie au paragraphe 5.2.5. Par conséquent, on a $`v \prec v$ `in` $Obs \| F$.

Le programme *ObsRep* constitue un exemple de programme décrit en utilisant explicitement des séquences et dont nous avons dit plus haut que leur preuve était généralement très lourde. Ceci est d'autant plus vrai dans le cas présent étant donné qu'à cause de la sémantique des observations, la preuve doit être faite au niveau des exécutions du programme. C'est pourquoi nous ne détaillons pas ici cette preuve dont nous ne présentons que le principe général. En enrichissant par superposition le programme $ObsRep \| F$ avec les historiques des variables $`v$ et $v$ et en s'appuyant sur le fait que $`v$ et l ne sont pas modifiées dans F, on montre que la concaténation de l'historique de $`v$ et de la séquence $l$ constitue une sous-suite de l'historique de la variable $v$. On en déduit que l'historique de $`v$ est une sous-suite de l'historique de v. On montre en outre facilement que la longueur de cet historique n'est pas bornée ce qui garantit la vivacité de l'observation. On en déduit donc la relation $`v \prec v$ `in` $ObsRep \| F$.

Remarque : ces deux exemples d'implantation réalisent en fait une relation d'observation *stricte*. L'implantation d'une relation d'observation non stricte est plus délicate à décrire en UNITY. La composition par union n'est plus suffisante et une superposition du programme est alors nécessaire.

5. La variable $`v$ est déclarée avec le même type que la variable v qu'on suppose être le type générique T.

5.3.3 La sémantique opérationnelle $\mathcal{O}bs$

Les programmes du paragraphe précédent montrent qu'il est possible de traduire systématiquement un programme avec une section *observe* en un programme UNITY standard ne contenant plus d'observations. Néanmoins, le rôle du mécanisme d'observation étant de fournir une description abstraite de la communication, ce mécanisme perdrait tout son intérêt s'il devait être traduit systématiquement en instructions UNITY. C'est pourquoi il est nécessaire de prendre en compte la section *observe* d'un programme directement au niveau de sa sémantique.

Nous avons vu aux paragraphes 2.3.2 et 2.3.3 que des propositions avaient été faites pour prendre en compte l'influence de l'environnement d'un programme au niveau de sa sémantique opérationnelle. Ces sémantiques sont des extensions de la sémantique $\mathcal{O}_1$, fondées sur un entrelacement des transitions dues au programme et des transitions dues à son environnement. Les transitions de l'environnement sont alors contraintes par un prédicat binaire (cf § 2.3.2) ou par une relation de raffinement vis-à-vis d'un transformeur de prédicats (cf § 2.3.3). Malheureusement, ces extensions ne sont pas adaptées au problème spécifique d'un environnement satisfaisant des relations d'observation. Il y a à cela trois raisons principales :

- telle que la relation d'observation est définie, une image et sa source sont susceptibles d'évoluer lors de la même transition d'état. Cette particularité ne pourrait être maintenue en utilisant des modèles sémantiques reposant sur l'entrelacement des transitions d'un système et de son environnement [6] ;

- dans ces sémantiques, les contraintes imposées à l'environnement se limitent à lui interdire de violer des propriétés de sûreté. Le prédicat binaire utilisé par P. Collette est reflexif et le transformeur de prédicat utilisé par R.T. Udink admet *skip* comme raffinement. Par conséquent, un environnement ne modifiant jamais l'état du système satisfait toutes les contraintes exprimables dans ce contexte. Il est donc impossible par ce moyen d'imposer à un environnement de respecter la vivacité des observations ;

- la relation d'observation est définie sur les exécutions d'un programme et nous avons vu au paragraphe 5.2.5 qu'il n'était pas évident de trouver un équivalent axiomatique défini au niveau des transitions. Nous avons donc besoin, pour exprimer l'observation, de contraindre l'environnement au niveau des exécutions complètes du couple (programme, environnement) et non plus au niveau des transitions de l'environnement.

Pour ces raisons, nous devons donc définir une sémantique opérationnelle des programmes UNITY qui permette aux images d'évoluer simultanément avec leur source, et de façon conforme à la sémantique des observations, y compris en ce qui concerne son aspect de vivacité.

6. Nous reviendrons plus longuement sur l'intérêt de ce parallélisme entre l'évolution d'une image et celle de sa source dans le chapitre 7.

Cette extension de la sémantique opérationnelle $\mathcal{O}_1$ est notée $\mathcal{O}bs$. Elle est spécifique à la relation d'observation et se définit de la façon suivante :

– $F.Img$ est l'ensemble des variables images du programme F, c'est-à-dire les variables qui apparaissent dans le membre gauche d'(exactement) une relation d'observation de $F.observe$;

$$F.Img \;=\; \{x \in F.declare \mid \langle \exists y : y \in F.declare :: x \prec y \ \mathtt{in}^7\ F.observe\rangle\}$$

– $F.NImg$ est l'ensemble des variables qui ne sont images dans aucune relation d'observation de $F.observe$. Il s'agit donc des variables affectées par les instructions du programme.

Une variable du programme F est nécessairement dans une et une seule de ces deux classes :

$$F.Img \cup F.NImg = F.declare \quad \wedge \quad F.Img \cap F.NImg = \emptyset$$

– $\mathcal{O}_{assign}.F$ est l'ensemble des séquences d'états σ qui satisfont les trois contraintes :

  – $\sigma_0.F.init$,
  – $\langle \forall t :: \langle \exists s : s \in F.assign :: \sigma_t \xrightarrow{s} \sigma_{t+1}\rangle\rangle$ and
  – $\langle \forall s : s \in F.assign :: \langle \forall t :: \langle \exists t' : t' > t :: \sigma_{t'} \xrightarrow{s} \sigma_{t'+1}\rangle\rangle\rangle$.

où $\sigma_t \xrightarrow{s} \sigma_{t+1} \;\equiv\; \langle \forall x : x \in F.NImg :: \sigma_{t+1}.x = (s.\sigma_t).x\rangle$;

– $\mathcal{O}_{observe}.F$ est l'ensemble des séquences d'états σ qui satisfont la contrainte :

$$\langle \forall\ x \in F.Img : x \prec y \in F.observe :: \langle \exists H : Horloge(H) :: \langle \forall t :: \sigma_t.x = \sigma_{H.t}.y\rangle\rangle\rangle;$$

– $\mathcal{O}bs.F = \mathcal{O}_{assign}.F \cap \mathcal{O}_{observe}.F$.

D'un côté, l'ensemble $\mathcal{O}_{assign}$ décrit les modifications des variables par les instructions du programme, en respectant l'hypothèse d'équité faible. Ces instructions sont exécutées à partir d'un état complet du programme, y compris les variables images, mais seule l'évolution des variables non images est prise en compte. Les variables images sont libres de prendre n'importe quelle valeur lors d'une transition. De l'autre côté, l'ensemble $\mathcal{O}_{observe}$ contient toutes les séquences d'états satisfaisant les relations d'observation de $F.observe$. La sémantique opérationnelle $\mathcal{O}bs$ est constituée des séquences satisfaisant les deux classes de contraintes.

Nous disposons donc d'un modèle opérationnel dans lequel les relations des sections *observe* des programmes sont implicitement satisfaites. Il faut noter

7. $x \prec y$ **in** $F.observe$ signifie qu'un des termes de la conjonction de relations d'observation $F.observe$ est de la forme $x \prec y$.

qu'étant donné que la *relation* d'observation a été définie pour une sémantique opérationnelle quelconque des programmes UNITY, cette relation et les théorèmes qui la lient aux opérateurs de la logique sont valables pour la sémantique $\mathcal{O}bs$.

Nous avons vu au paragraphe 3.2.2 que les théorèmes classiques de la logique UNITY (c'est-à-dire ceux du livre de K.M. Chandy et J. Misra) sont valables pour n'importe quelle sémantique opérationnelle. Il sont donc utilisables pour les opérateurs définis sur $\mathcal{O}bs$.

5.3.4 Représentation de la communication

Le mécanisme d'observation que nous venons de présenter permet de décrire la communication entre les différents processus d'une application répartie. Cette description est plus abstraite que les échanges explicites de messages et s'avère plus facile à manipuler lors de la modélisation et lors de la preuve de correction d'un algorithme réparti.

Naturellement, ceci suppose que le mécanisme d'observation est le seul moyen de communication utilisé par les processus. Afin de caractériser ce type de programme réparti, nous généralisons la composition par union aux programmes UNITY munis d'une section *observe*, et nous définissons la notion de programme *bien réparti*.

Nous avons vu qu'un programme réparti était représenté en UNITY par une union de processus UNITY :

$$P \;=\; \langle [\!]\, i : P_i \rangle$$

La composition par union s'applique aux programmes munis d'une section *observe* simplement en considérant la conjonction des relations d'observation :

$$\langle [\!]\, i : P_i \rangle.observe = \bigwedge_i P_i.observe$$

la définition des autres sections du programme $\langle [\!]\, i : P_i \rangle$ n'étant pas modifiée.

Comme dans le cas de la composition des sections *declare* et *initially*, l'union n'est possible qu'en l'absence de conflits entre les sections *observe*. Plus précisément, une même variable ne doit pas être image à la fois d'une relation de $P_i.observe$ et d'une relation de $P_j.observe$ pour $i \neq j$:

$$\langle \forall i, j : i \neq j :: P_i.Img \cap P_j.Img = \emptyset \rangle \qquad (5.12)$$

Sous cette hypothèse, la section *observe* du programme $\langle [\!]\, i : P_i \rangle$ est bien consistante.

Il faut ensuite spécifier que les relations d'observation constituent le seul moyen de communication entre différents processus UNITY. Pour cela, il suffit d'interdire le partage de variables autrement que par le biais des relations d'observation. Par convention, on choisit de placer une relation d'observation dans la section *observe* du processus observant. En d'autres termes, les images sont locales et les sources distantes. Un programme *bien réparti* est donc un programme qui ne référence des variables extérieures que dans les membres droits de ses relations d'observation.

Un processus UNITY ne déclare donc pas les variables qui apparaissent en partie droite de ses relations d'observation[8]. Ces variables doivent naturellement être déclarées au niveau d'un autre processus pour que le programme composé soit bien défini. On ne retrouve donc, dans la section *declare* d'un processus, que ses variables locales. On définit un programme *bien réparti* de la façon suivante :

$$\langle [\![i :: P_i \rangle \text{ est } \textit{bien réparti} \equiv \langle \forall i, j : i \neq j :: P_i.\mathit{declare} \cap P_j.\mathit{declare} = \emptyset \rangle$$

Il faut noter qu'un programme bien réparti satisfait nécessairement la contrainte (5.12). Par conséquent, sa section *observe* est toujours consistante.

Dans la suite de ce chapitre et dans les deux chapitres suivants, nous ne considérerons plus que des programmes communiquant par observation et bien répartis.

5.4 Exemples d'algorithmes communiquant par observation

Dans cette partie, nous présentons différents exemples d'algorithmes qui utilisent le mécanisme d'observation pour réaliser les communications. Nous ne décrivons que les programmes, sans spécifications ni preuves de correction, excepté une preuve de raffinement au paragraphe 5.4.3. Le but de cette partie est d'illustrer le comportement intuitif du mécanisme d'observation. Un exemple de preuve détaillée de correction fera l'objet de la partie suivante.

5.4.1 Le protocole du bit alterné

Nous considérons un système composé d'un émetteur *Sender* et d'un récepteur *Receiver*. L'émetteur envoie successivement tous les messages contenus dans une file d'attente In, le récepteur ajoute les messages reçus à une file Out.

L'émetteur et le récepteur utilisent le protocole du bit alterné : chaque message est acquitté par un bit opposé au bit d'acquittement du message précédent. Un message non acquitté peut être réémis et un même acquittement peut être envoyé plusieurs fois. Par conséquent, ce protocole est tolérant à la perte et à la duplication de messages, à la fois dans le sens du transfert des données que dans le sens des acquittements.

L'émetteur gère un bit b qu'il associe à chaque message envoyé. Il attend un acquittement de ce même bit avant d'envoyer le message suivant. Le récepteur gère un bit d'acquittement b_ack et ne prend en compte que les messages dont

8. Sauf dans le cas, improbable, où le processus choisit d'observer une de ses variables locales.

le bit est opposé à b_ack. Avant chaque acquittement, il inverse la valeur de b_ack.

```
Program Sender
Declare
   In : List Of T;
   msg : Int;
   b : Bool;
   'b_ack : Bool;
Observe
   'b_ack ≺ b_ack
Initially
   b;
Assign
   msg, In, b:= head(In), tail(In), ¬b
               if (b = 'b_ack) ∧ (In ≠ nil)
End Sender
```

```
Program Receiver
Declare
   Out : List Of T;
   b_ack : Bool;
   'msg : Int;
   'b : Bool;
Observe
   ('msg, 'b) ≺ (msg, b)
Initially
   b_ack;
Assign
   Out, b_ack:= append(Out,⟨'msg⟩), ¬b_ack
               if b_ack = ¬'b
End Receiver
```

Toutes les communications sont réalisées par observation (le système est bien réparti). On utilise une observation simultanée de msg et de b pour associer le bit de synchronisation et le message correspondant.

5.4.2 Construction d'un arbre de recouvrement

Nous abordons ici le problème de la construction d'une structure d'arbre sur un réseau connexe de sites. Quel que soit le nombre de sites qui commencent une construction, l'état du système, à la terminaison de ce type d'algorithme, doit être le suivant :

1. un arbre unique, contenant tous les sites, est construit ;
2. la racine sait qu'elle est racine, les autres processus connaissent leur père ;
3. chaque site connaît l'ensemble de ses fils dans l'arbre ;
4. la racine de l'arbre détecte la terminaison de la construction de l'arbre.

Dans la suite du paragraphe, nous décrivons en UNITY un algorithme communiquant par observation qui répond aux contraintes 1 et 2. Nous enrichissons ensuite cet algorithme par deux superpositions successives afin de répondre aux contraintes 3 et 4.

Nous supposons que les sites ont tous une identité propre, et que ces identités sont totalement ordonnées. Initialement, un site connaît son identité ainsi que celle des sites auxquels il est connecté (ses voisins).

Lorsqu'un site veut construire un arbre, il commence un processus de construction ayant pour “couleur” son numéro de site. Les sites se rattachent à cet arbre en choisissant pour père le premier site qu'ils voient avec cette couleur. Un site déjà rattaché à un arbre n'accepte de se rattacher à un autre arbre que si la couleur de ce nouvel arbre est supérieure à celle de l'arbre dans lequel il est déjà. Ainsi, l'ordre total entre les couleurs permet d'interrompre la construction d'un arbre au profit de la construction d'un arbre plus prioritaire. Parmi les différents sites qui démarrent un processus de construction, seul le site de numéro le plus élevé parviendra effectivement à construire un arbre dont il sera la racine. Les autres se rattacheront à cet arbre.

On considère un système composé de N sites. Une variable var d'un site j est représentée par $j.var$. Le comportement des sites est le suivant :

```
Program Site (myself : Int, voisins : Set Of Int)
Declare
  pere, couleur : Int;
  ‘couleur : Array 1..N of Int;
  veut_construire : Bool;
Observe
  ⟨∀j : j ∈ voisins :: ‘couleur[j] ⊰ j.couleur⟩
Initially
  (pere = 0) ∧ (couleur = 0);
Assign
  couleur  :=  myself if (couleur = 0) ∧ veut_construire
‖
  ⟨ ‖j : j ∈ voisins ::
     pere, couleur  :=  j, ‘couleur[j] if ‘couleur[j] > couleur
  ⟩
End Site
```

Le système atteint un point-fixe dans lequel tous les sites ont la même couleur. L'un d'eux a pour père 0 (il est la racine) et tous les autres sites ont chacun un père et un seul parmi leurs voisins. Il existe donc bien une structure d'arbre unique entre les sites.

Pour répondre au point 3 de la spécification du problème, il faut que chaque site connaisse l'ensemble de ses fils. Pour cela, chaque site observe ses voisins, et considère comme ses fils les voisins qui le considèrent comme père. Lorsqu'une construction d'arbre est abandonnée au profit d'une autre, un site réinitialise son ensemble de fils et recommence son enquête auprès de ses voisins.

Ces fonctionnalités sont ajoutées au programme précédent par superposition (cf § 2.1.6). Une variable *fils* de type ensemble d'entiers est ajoutée à chaque site. Elle est initialement vide. Lorsqu'un site se rattache à un nouvel arbre, elle est réinitialisée à vide. De plus, un site ajoute à l'ensemble de ses fils ses voisins, présents dans un arbre de même couleur, et dont il est le père. Le comportement d'un site devient :

```
Program Site (myself : Int, voisins : Set Of Int)
Declare
  pere, couleur : Int;
  ‘pere, ‘couleur : Array 1..N of Int;
  fils : Set Of Int;
  veut_construire : Bool;
Observe
  ⟨∀j : j ∈ voisins :: (‘pere[j], ‘couleur[j]) ⊰ (j.pere, j.couleur)⟩
Initially
  (pere = 0) ∧ (couleur = 0) ∧ (fils = ∅);
Assign
  couleur  :=  myself if (couleur = 0) ∧ veut_construire
‖
  ⟨ ‖j : j ∈ voisins ::
     pere, couleur, fils  :=  j, ‘couleur[j], ∅ if ‘couleur[j] > couleur
  ⟩
‖
  ⟨ ‖j : j ∈ voisins ::
     fils  :=  fils ∪ {j} if (‘pere[j] = myself ∧ ‘couleur[j] = couleur)
  ⟩
End Site
```

Enfin, pour satisfaire la quatrième contrainte, on ajoute au programme, toujours par superposition, un mécanisme de détection de terminaison. Un site considère qu'à son niveau la construction d'un arbre est terminée si tous ses voisins sont dans cet arbre et si tous ses fils considèrent que la construction est terminée à leur niveau. Lorsque la racine de l'arbre détecte la terminaison, la construction de l'arbre est achevée[9].

On ajoute donc aux sites une variable fin et chaque site observe la variable fin de ses voisins. Le programme complet devient donc :

```
Program Site (myself : Int, voisins : Set Of Int)
Declare
  pere, couleur : Int;
  ‘pere, ‘couleur : Array 1..N of Int;
  fils : Set Of Int;
  fin : Int; veut_construire : Bool;
Observe
  ⟨∀j : j ∈ voisins :: (‘pere[j], ‘couleur[j]) ⊰ (j.pere, j.couleur)⟩ ∧
  ⟨∀j : j ∈ voisins :: ‘fin[j] ⊰ j.fin⟩
Initially
  (pere = 0) ∧ (couleur = 0) ∧ (fils = ∅) ∧ (fin = 0);
Assign
  couleur  :=  myself if (couleur = 0) ∧ veut_construire
[]
  ⟨ []j : j ∈ voisins ::
     pere, couleur, fils  :=  j, ‘couleur[j], ∅ if ‘couleur[j] > couleur
  ⟩
[]
  ⟨ []j : j ∈ voisins ::
     fils  :=  fils ∪ {j} if (‘pere[j] = myself ∧ ‘couleur[j] = couleur)
  ⟩
[]
  fin  :=  couleur if ⟨∀j : j ∈ voisins ::
                         ‘couleur[j] = couleur ∧ (‘pere[j] = myself ⇒ ‘fin[j] = couleur)⟩
End Site
```

La racine de l'arbre construit détecte la fin de la construction par la condition $fin = myself$[10].

L'arbre construit par cet algorithme n'a pas de structure particulière. Néanmoins, on peut facilement modifier l'algorithme pour imposer des contraintes plus strictes sur la forme de l'arbre. Par exemple, on peut associer à chaque site sa profondeur dans l'arbre et favoriser, entre deux pères potentiels *de même couleur*, celui dont la profondeur est la plus petite. On construit ainsi un arbre de profondeur minimale.

5.4.3 Exclusion mutuelle par front circulant

Nous nous intéressons dans ce paragraphe à un algorithme d'exclusion mutuelle basé sur un front circulant sur un anneau. Le système considéré est constitué de N sites identiques répartis sur un anneau logique. Chaque site gère un compteur croissant de telle manière qu'il existe toujours un site et un seul dont le compteur soit plus petit que celui de son prédécesseur sur l'anneau. L'unicité de ce front dans les valeurs des compteurs assure la propriété d'exclusion mutuelle. L'équité faible de l'algorithme est obtenue en faisant circuler ce front autour de l'anneau.

9. On peut remarquer que la structure d'arbre construite est elle-même utilisée dans la détection de terminaison.

10. Si l'on souhaite que les autres sites détectent également la fin de la construction, il faut alors que la racine propage à travers l'arbre le résultat de sa détection.

Pour savoir s'il possède le privilège, un site doit connaître la valeur du compteur de son prédécesseur sur l'anneau. Si l'on fait abstraction des communications, on peut considérer que chaque site connaît instantanément la valeur du compteur du site précédent. Par contre, dans un contexte réparti, il est nécessaire que les sites communiquent pour acquérir cette connaissance. Dans ce paragraphe, nous montrons que les théorèmes sur l'observation des compteurs permettent d'établir une relation de raffinement entre le système dans lequel la connaissance des compteurs est immédiate et celui dans lequel les sites utilisent le mécanisme d'observation pour obtenir les valeurs des compteurs.

Dans le cas où la communication n'est pas prise en compte, on peut modéliser le comportement d'un site par le programme UNITY suivant :

```
Program Site (i)
Declare
   req, excl : Bool;
   c : Int;
Initially
   (c = i) ∧ ¬req ∧ ¬excl;
Assign
   req := true
[]
   excl := true if req ∧ ¬excl¹¹ ∧ (c < Site(i − 1 mod N).c)
[]
   excl, req, c := false, false, c + N if excl
[]
   c := c + N if ¬req ∧ (c < Site(i − 1 mod N).c)
End Site
```

Deux des instructions du programme $Site(i)$ manipulent explicitement $Site(i - i \ mod \ N).c$ qui est une variable d'un autre site. Une telle connaissance immédiate est impossible dans un contexte réparti. Nous considérons à présent une description répartie du système dans laquelle un site i utilise le mécanisme d'observation pour connaître la valeur du compteur c du site $i - 1 \ mod \ N$:

```
Program Site' (i)
Declare
   req, excl : Bool;
   c : Int;
Observe
   [ t ⋞ Site'(i − 1 mod N).c ]
Initially
   (c = i) ∧ ¬req ∧ ¬excl;
Assign
   req := true
[]
   excl := true if req ∧ ¬excl ∧ [ c < t ]
[]
   excl, req, c := false, false, c + N if excl
[]
   c := c + N if ¬req ∧ [ c < t ]
End Site
```

La seule différence entre les programmes $Site$ et $Site'$ est que $c < t$ remplace $c < Site(i-1 \ mod \ N).c$ dans les deux gardes concernées. En d'autres termes, le programme se base maintenant sur une *image* du compteur de son prédécesseur pour entrer en section critique.

11. La conjonction $\neg excl$ est utile pour la preuve de raffinement. Si la variable $excl$ est déjà vraie, l'instruction n'a pas d'effet. Par conséquent, $\neg excl$ n'est pas nécessaire dans la garde.

Nous allons à présent démontrer la relation de raffinement suivante :

$$\langle [\![i : 0 \leqslant i < N :: Site(i) \rangle \sqsubseteq_{L_{\mathcal{O}bs}} \langle [\![i : 0 \leqslant i < N :: Site'(i) \rangle \tag{5.13}$$

D'après cette relation, toutes les propriétés prouvées sur le programme (plus simple) dans lequel les aspects liés à la communication sont ignorés restent valides dans le programme bien réparti communiquant par observation.

La preuve de cette relation de raffinement repose sur le théorème (4.13) de renforcement des gardes démontré au paragraphe 4.3.2[12]. La preuve n'est détaillée que pour la modification de la garde de la deuxième instruction. Le cas de la quatrième instruction est similaire.

Le programme $Site''$ est obtenu en renforçant la garde de la deuxième instruction du programme $Site$ par la conjonction $c < t$. Dans la suite, nous abrégeons $Site(i).var$ en var_i et $i-1 \ mod \ N$ en $i-1$. L'instruction pour le site i devient donc :

$$excl_i := true \ \texttt{if} \ req_i \wedge \neg excl_i \wedge c_i < c_{i-1} \wedge c_i < t_i$$

On note S_i le système dans lequel cette transformation est appliquée aux sites entre 0 et i, les autres sites restant inchangés :

$$S_i \ = \ \langle [\![j : 0 \leqslant j \leqslant i :: Site''(j) \rangle \ [\![\ \langle [\![j : i < j < N :: Site(j) \rangle$$

Le passage du système S_{i-1} au système S_i entre dans le cadre du théorème (4.13) et constitue un raffinement de programme si les deux conditions suivantes sont établies (les opérateurs de la logique UNITY sont ici implicitement indicés par $\mathcal{O}bs$) :

$$req_i \wedge \neg excl_i \wedge c_i < c_{i-1} \ \mapsto \ c_i < t_i \ \texttt{in} \ S_i \tag{5.14}$$

$$c_i < t_i \ \texttt{unless} \ \neg req_i \vee excl_i \vee c_i \geqslant c_{i-1} \ \texttt{in} \ S_i \tag{5.15}$$

Dans ce cas, on déduit la validité de la relation $S_{i-1} \sqsubseteq_{L_{\mathcal{O}bs}} S_i$ et, par transitivité, la relation :

$$S_0 \sqsubseteq_{L_{\mathcal{O}bs}} S_{N-1}$$

Le système S_0 correspond au système $\langle [\![i : 0 \leqslant i < N :: Site(i) \rangle$, membre gauche de la relation (5.13). Dans le système S_{N-1}, la deuxième instruction de chaque site est gardée par :

$$req_i \wedge \neg excl_i \wedge c_i < c_{i-1} \wedge c_i < t_i$$

c'est-à-dire par :

$$req_i \wedge \neg excl_i \wedge c_i < t_i \wedge (c_i < t_i \Rightarrow c_i < c_{i-1})$$

Comme les variables c sont des compteurs croissants, d'après le théorème (5.10), on a :

$$\texttt{invariant} \ t_i \leqslant c_{i-1} \ \texttt{in} \ S_{N-1}$$

12. Au paragraphe 4.3.2, ce théorème a été démontré pour la sémantique opérationnelle $\mathcal{O}_1$. Néanmoins, la preuve est strictement la même pour la sémantique $\mathcal{O}bs$.

et donc :

$$\texttt{invariant}\ c_i < t_i \Rightarrow c_i < c_{i-1}\ \texttt{in}\ S_{N-1}$$

Par conséquent, la conjonction $c_i < t_i \Rightarrow c_i < c_{i-1}$ est inutile dans la garde et le système S_{N-1} est équivalent au système $\langle [\![i : 0 \leqslant i < N :: Site'(i) \rangle$, membre droit de la relation (5.13). On en déduit donc la validité de la relation de raffinement (5.13).

Il nous reste donc à démontrer les propriétés (5.14) et (5.15). La propriété de sûreté (5.15) découle directement du texte du programme. En effet, la variable t_i est un compteur croissant d'après le théorème (5.9). Par conséquent, le seul moyen de falsifier le membre gauche du *unless* est de faire évoluer la variable c_i. Ceci n'est possible que par la deuxième ou la quatrième instruction du programme $Site(i)$. Or, ces instructions sont gardées soit par $excl_i$, soit par $\neg req_i$. Dans les deux cas, le membre droit du *unless* est donc vrai et la propriété (5.15) est satisfaite.

La propriété de vivacité (5.14) provient intuitivement du fait que la variable c_i ne peut être modifiée avant que le membre droit du *leads-to* ne devienne vrai et de la vivacité de l'observation. En effet, pour les mêmes raisons que précédemment, le programme S_{N-1} satisfait la propriété :

$$\forall l :: req_i \wedge \neg excl_i \wedge c_i < c_{i-1} \wedge c_i = l\ \texttt{unless}\ c_i < t_i \tag{5.16}$$

D'autre part, c_i étant un compteur croissant, le théorème (5.11) donne :

$$\forall k :: c_{i-1} = k \ \mapsto\ t_i \geqslant k \tag{5.17}$$

En appliquant le théorème PSP à (5.16) et5.17), on obtient :

$$\begin{array}{c} \forall k, l :: c_{i-1} = k \wedge req_i \wedge \neg excl_i \wedge c_i < c_{i-1} \wedge c_i = l \\ \mapsto \\ (t_i \geqslant k \wedge req_i \wedge \neg excl_i \wedge c_i < c_{i-1} \wedge c_i = l) \vee c_i < t_i \end{array} \tag{5.18}$$

ce qui permet de dériver la propriété (5.14) :

$$\begin{array}{ll} & req_i \wedge \neg excl_i \wedge c_i < c_{i-1} \\ \Rightarrow & \quad \{\text{logique des prédicats}\} \\ & \langle \exists k, l :: k > l \wedge c_i = l \wedge c_{i-1} = k \wedge req_i \wedge \neg excl_i \wedge c_i < c_{i-1} \rangle \\ \mapsto & \quad \{(5.18),\ \text{théorème de disjonction}\} \\ & \langle \exists k, l :: (\underline{k > l} \wedge \underline{c_i = l} \wedge \underline{t_i \geqslant k} \wedge req_i \wedge \neg excl_i \wedge c_i < c_{i-1}) \vee c_i < t_i \rangle \\ \Rightarrow & \quad \{\text{logique des prédicats}\} \\ & c_i < t_i \end{array}$$

5.4.4 Détection de la terminaison

L'algorithme que nous considérons à présent est un algorithme de détection de terminaison proposé par F. Mattern. L'application dont on veut détecter la terminaison est supposée satisfaire les contraintes suivantes :

– initialement, tous les sites, sauf un, sont passifs ;

– ce site particulier envoie un nombre fini de messages puis devient passif ;

– à chaque réception d'un message, un site passe dans l'état actif, effectue des calculs locaux, envoie éventuellement un nombre fini de messages et retourne dans l'état passif.

L'application est terminée lorsque tous les sites sont passifs (c'est-à-dire en attente d'un message) et qu'il ne reste plus de messages en transit (tous les messages envoyés ont été reçus). L'algorithme fait en outre l'hypothèse que la réaction d'un site à la réception d'un message (calculs locaux et envois de messages) est *atomique*[13].

Le principe de l'algorithme consiste à lancer des *vagues* successives pour récolter le nombre de messages émis et recus par chaque site. La terminaison est détectée dès lors que deux vagues successives (ne se chevauchant pas) obtiennent le même total de messages reçus et le même total de messages émis.

On peut utiliser le mécanisme d'observation pour récolter les valeurs des compteurs d'émission et de réception et pour réaliser une succession de vagues. Le site de détection possède une image $`E[i]$ du compteur d'émissions $E_i$ et une image $`R[i]$ du compteur de réception R_i de chacun des sites i. Il gère en plus un compteur de vagues v que tous les autres sites observent dans une variable $`v_i$. En observant simultanément $E_i$, $R_i$ et $`v_i$, le site de détection peut récolter les valeurs des compteurs de tous les sites correspondant à une vague donnée. Une fois que toutes ces valeurs sont récupérées, le site peut déclancher une nouvelle vague en incrémentant le compteur v. Le comportement du site de détection est modélisé par le programme suivant (nous omettons tous les intervalles de quantification de la forme $1 \leqslant i \leqslant N$ où N est le nombre de sites observés) :

```
Program Detection
Declare
   ‘E, ‘R, ‘‘V : Array of Int;
   SRP : Int;
   v : Int;
   Terminaison : Bool;
Always
   Bool  Fin_de_vague          = ⟨∀i :: ‘‘V[i] = v⟩;
   Bool  Application_demarree  = ⟨∃i :: ‘E[i] > 0⟩;
Initially
   ⟨∀i :: E[i],R[i] = 0,0⟩ ∧
   (v = 0) ∧ (SRP = 0) ∧ ¬Terminaison;
Assign
   v, SRP, Terminaison  :=  v + 1, ⟨+i :: ‘R[i]⟩, (SRP = ⟨+i :: ‘E[i]⟩)
                        if  Application_demarree ∧ Fin_de_vague
End Detection
```

La variable SRP représente la somme des compteurs de réception récoltés lors de la vague précédant la vague courante. L'égalité entre SRP et la somme des compteurs d'émission récoltés par la vague courante est une condition suffisante pour la détection de la terminaison (les sommes des compteurs d'émission et de réception récoltés par les deux vagues sont alors égaux).

13. Sous cette hypothèse, on ne peut observer un site que dans l'état passif. L'algorithme détecte en fait l'absence de messages en transit.

5.5 Algorithme d'exclusion mutuelle à base d'horloges logiques

5.5.1 Présentation

Nous présentons dans cette partie un exemple complet d'un algorithme d'exclusion mutuelle ainsi que sa preuve de correction. Cet algorithme utilise des horloges logiques et s'inspire de l'algorithme de la boulangerie de L. Lamport.

Le programme que nous étudions ici décrit le comportement identique de N sites par un automate à trois états. Un site se trouve initialement dans l'état *thinking* et en dehors de la section critique. Il peut ensuite déposer une requête d'accès et reste alors dans l'état *requesting* jusqu'à ce qu'il obtienne l'accès à la section critique. Finalement, il accède à la section critique et entre dans l'état *mutex*. Le site reste en section critique pendant une durée finie, puis retourne dans l'état *thinking*.

Un tel algorithme d'exclusion mutuelle doit respecter deux contraintes essentielles de correction :

- *sûreté* : il n'y a au plus qu'un site en section critique à un instant donné ;
- *vivacité* : tout site ayant formulé une requête est autorisé à accéder à la section critique après une attente finie.

L'algorithme que nous décrivons ici utilise des horloges logiques pour dater les requêtes. Lorsqu'un site demande l'accès à section critique, sa requête est datée avec l'horloge courante du site. Le site sera autorisé à entrer lorsque sa requête sera la plus ancienne, c'est-à-dire datée avec la plus petite valeur d'horloge.

5.5.2 Utilisation du mécanisme d'observation

Nous considérons un système réactif composé de N sites et nous associons à chaque site $Mutex(s)$ un vecteur d'horloges[14] $C[1..N]$. Dans ce vecteur, la composante $Mutex(s).C[s]$ représente l'horloge du site s. Les autres composantes sont les images que possède le site s des horloges des autres sites. En d'autres termes, $Mutex(s).C[s']$ pour $s \neq s'$ est une image, par la relation d'observation, de l'horloge distante $Mutex(s').C[s']$. Ces relations d'observation entre les sites, représentées dans la figure 5.1, s'expriment par la formule suivante, déduite de la concaténation des sections *observe* des différents sites :

$$\forall s, s' : 1 \leqslant s, s' \leqslant N \wedge s \neq s' :: Mutex(s).C[s'] \prec Mutex(s').C[s']$$

Le comportement d'un site vis-à-vis de son horloge est le suivant :

- initialement, toutes les horloges sont initialisées avec le numéro de leur site ;

14. Il s'agit de variables du programme représentant des horloges logiques, à ne pas confondre avec les fonctions horloges définies au paragraphe 5.2.2.

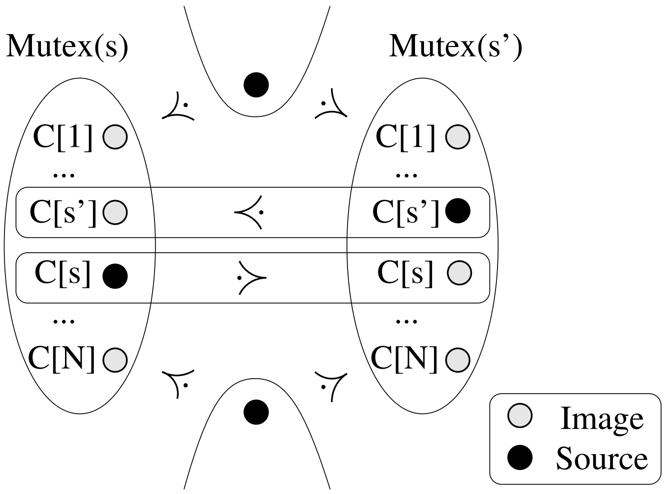

FIG. 5.1 – *Relations d'observation entre les sites.*

- lorsqu'un site $Mutex(s)$ est dans l'état *thinking*, il ne doit pas empêcher les autres sites d'accéder à la section critique. Pour cela, un tel site incrémente son horloge $C[s]$ infiniment souvent ;
- lorsqu'un site commence une requête d'accès, il cesse d'incrémenter son horloge ;
- un site accède à la section critique lorsque son horloge (qu'il a préalablement arrêtée lors de sa requête d'accès) devient plus petite que toutes les images des horloges des autres sites.

Le programme UNITY suivant décrit ce comportement d'un site :

```
Program Mutex(s)
Declare
   C : Array 1..N of Int;
   state : {thinking, requesting, mutex};
Always
   Bool Minimum = ⟨∀s' : 1 ⩽ s' ⩽ N  ∧  (s' ≠ s) :: C[s] < C[s']⟩;
Observe
   ⟨∀s' : 1 ⩽ s' ⩽ N  ∧  (s' ≠ s) :: C[s'] ≺ Mutex(s').C[s']⟩
Initially
   (C[s] = s) ∧ (state = thinking);
Assign
    { incrémentation de l'horloge }
   C[s] := C[s] + N if state = thinking
[]
    { dépot d'une requête et attente }
   state := requesting if state = thinking
[]
    { entrée en section critique }
   state := mutex if (state = requesting) ∧ Minimum
[]
    { sortie et incrémentation de l'horloge }
   state, C[s] := thinking, C[s] + N if state = mutex
End Mutex
```

Le système réparti global est donc décrit par l'union des différents processus UNITY :

$$SR \;=\; \langle [\!] \; s \;:\; 1 \leqslant s \leqslant N \;::\; Mutex(s) \rangle$$

Les seules références, dans le programme $Mutex(s)$, à des variables extérieures, sont les $Mutex(s').C[s']$ que l'on retrouve en partie droite des relations de la section *observe*. Par conséquent, le programme SR est bien réparti.

La section observe de ce programme, consistante puisque le programme est bien réparti, est constituée de la conjonction des relations d'observation des programmes :

$$\forall s, s' : 1 \leqslant s, s' \leqslant N \wedge s \neq s' :: Mutex(s).C[s'] \prec Mutex(s').C[s']$$

On retrouve bien le schéma de communication présenté précédemment.

Dans la spécification et la preuve qui suivent, nous adopterons les abréviations suivantes :

- on abrège $Mutex(s).var$ en $s.var$;
- on abrège $Mutex(s).state = mutex$, $Mutex(s).state = requesting$ et $Mutex(s).state = thinking$ respectivement en $s.mutex$, $s.requesting$ et $s.thinking$;
- on utilise également $s.Minimum$ pour désigner la variable $Minimum$ du processus $Mutex(s)$, telle qu'elle est définie dans $Mutex(s).always$.

5.5.3 Preuve de correction

Les spécifications de sûreté et de vivacité d'un programme d'exclusion mutuelle, présentées informellement au paragraphe 5.5.1, se traduisent, en logique UNITY, par :

- sûreté :

$$\begin{array}{c}\texttt{invariant}_{\mathcal{O}bs} \\ \langle \forall s, s' : 1 \leqslant s, s' \leqslant N :: s.mutex \wedge s'.mutex \ \Rightarrow \ (s = s') \rangle \\ \texttt{in } SR\end{array} \qquad (5.19)$$

- vivacité :

$$\forall s \ : 1 \leqslant s \leqslant N :: \ s.resquesting \ \mapsto_{\mathcal{O}bs} \ s.mutex \ \texttt{in } SR \qquad (5.20)$$

Dans la preuve qui suit, nous écrirons les opérateurs de la logique UNITY sans indice particulier. Ces opérateurs sont considérés définis opérationnellement sur la sémantique $\mathcal{O}bs$. Toutes les propriétés logiques énoncées portent sur le programme SR et nous les écrirons sans expliciter "`in` SR". Nous omettrons également le terme $1 \leqslant \cdots \leqslant N$ dans tous les prédicats : la quantification des variables représentant les sites s'effectue implicitement sur l'intervalle $1..N$.

Propriétés des horloges

Dans le programme SR, les horloges des sites sont implantées par des variables entières initialisées par le numéro du site. Ces horloges sont incrémentées par pas de N, N étant le nombre de sites. Cette implantation d'horloges constitue une réalisation simplifiée des horloges logiques de L. Lamport. Elle garantit les propriétés suivantes :

– les domaines de valeur des horloges sont disjoints :

$$\texttt{invariant}\ \langle \forall s, s' :\ s \neq s'\ :: Dom(s.C[s]) \cap Dom(s'.C[s']) = \emptyset \rangle \quad (5.21)$$

En effet, on peut facilement vérifier l'invariant :

$$\texttt{invariant}\ \langle \forall s :: s - 1\ mod\ N = s.C[s] - 1 \rangle$$

– les horloges sont des compteurs croissants :

$$\forall s, k ::\ \texttt{stable}\ \ C[s] \geqslant k \quad (5.22)$$

Le caractère croissant des horloges locales, incrémentées de N en N, est évident. Les autres composantes des vecteurs sont des images, par la relation d'observation, de compteurs croissants. En vertu du théorème (5.9), il s'agit donc également de compteurs croissants.

– les valeurs des horloges sont totalement ordonnées :

$$\texttt{invariant}\ \langle \forall s, s' :\ s \neq s'\ ::\ (s.C[s] \leqslant s'.C[s']) \vee (s.C[s] \geqslant s'.C[s']) \rangle \quad (5.23)$$

En effet, les horloges des sites sont des variables entières.

Propriété d'exclusion mutuelle

Lemme 1 *Si l'horloge locale d'un site est inférieure aux images qu'a ce site des horloges distantes, elle est également inférieure à toutes les horloges distantes.*

Démonstration : on considère un site $Mutex(s)$ dont l'horloge locale $C[s]$ est inférieure aux images des autres horloges :

$$\begin{array}{ll} & \langle \forall s' :\ s' \neq s\ :: s.C[s] < s.C[s'] \rangle \\ \Rightarrow & \quad \{\text{les horloges sont des compteurs croissants (5.22), théorème (5.10)}\} \\ & \langle \forall s' :\ s' \neq s\ :: s.C[s] < s.C[s'] \rangle \wedge \langle \forall s' :\ s' \neq s\ ::\ s.C[s'] \leqslant s'.C[s'] \rangle \\ \Rightarrow & \\ & \langle \forall s, s' :\ s' \neq s\ :: s.C[s] < s'.C[s'] \rangle \end{array}$$

Donc $C[s]$ est aussi un minimum global.

Lorsqu'un site entre en section critique, son horloge est inférieure aux images des autres horloges. D'après le lemme précédent, elle est aussi inférieure aux horloges réelles des autres sites. Tant que ce site est en section critique, son horloge n'est pas modifiée. Les horloges étant des compteurs croissants, l'horloge du site en section critique reste donc minimale tant que ce site reste en section critique. On a donc :

$$\forall s :: \texttt{invariant}\ s.mutex \Rightarrow s.Minimum$$

Par conséquent :

$$
\begin{array}{ll}
& s_1 \neq s_2 \wedge s_1.mutex \wedge s_2.mutex \\
\Rightarrow & \{\text{d'après l'invariant précédent}\} \\
& s_1 \neq s_2 \wedge s_1.Minimum \wedge s_2.Minimum \\
\Rightarrow & \{\text{définition de } Minimum\} \\
& s_1 \neq s_2 \wedge \langle \forall s : s \neq s_1 :: s_1.C[s_1] < s_1.C[s] \rangle \wedge \langle \forall s' : s' \neq s_2 :: s_2.C[s_2] < s_2.C[s'] \rangle \\
\Rightarrow & \{\text{d'après le lemme 1}\} \\
& s_1 \neq s_2 \wedge \langle \forall s : s \neq s_1 :: s_1.C[s_1] < s.C[s] \rangle \wedge \langle \forall s' : s' \neq s_2 :: s_2.C[s_2] < s'.C[s'] \rangle \\
\Rightarrow & \{s = s_2,\ s' = s_1\} \\
& s_1.C[s_1] < s_2.C[s_2] \wedge s_2.C[s_2] < s_1.C[s_1] \\
\Rightarrow & \\
& \mathit{false}
\end{array}
$$

On a donc bien la propriété d'exclusion mutuelle.

Propriété de vivacité

Les horloges des sites étant totalement ordonnées (5.23) et disposant de domaines de valeurs disjoints (5.21), il existe à tout instant un unique site qui possède la plus petite horloge. À un instant donné, soit s_0 ce site. On a donc :

$$\langle \forall s : s \neq s_0 :: s_0.C[s_0] < s.C[s] \rangle$$

Dans les trois lemmes suivants, nous montrons que l'horloge de s_0 finit par croître. On en déduit ensuite que le minimum global des horloges finit par augmenter, ce qui nous permet d'établir la propriété de vivacité.

Lemme 2 *Si s_0 est dans l'état mutex, il finit par incrémenter son horloge :*

$$\forall k :: \ s_0.mutex \wedge s_0.C[s_0] = k \quad \mapsto \quad s_0.C[s_0] > k \tag{5.24}$$

Démonstration : cette propriété se déduit directement de l'hypothèse d'équité faible du modèle UNITY : l'instruction “sortie et incrémentation de l'horloge” est finalement exécutée.

Lemme 3 *Si s_0 est dans l'état requesting, il finit par incrémenter son horloge :*

$$\forall k :: \ s_0.requesting \wedge s_0.C[s_0] = k \quad \mapsto \quad s_0.C[s_0] > k \tag{5.25}$$

Démonstration : le programme SR vérifie :

$$\forall k :: s_0.requesting \wedge s_0.C[s_0] = k \ \ \mathtt{unless} \ \ s_0.mutex \tag{5.26}$$

Si l'horloge d'un site est supérieure à une valeur l, d'après le théorème (5.8), ses images finiront par dépasser la valeur l :

$$\forall s :: \frac{s_0.C[s] \prec s.C[s],\ \forall m :: \mathtt{stable}\ \ s.C[s] > m}{\forall l :: s.C[s] > l \ \mapsto \ s_0.C[s] > l}$$

D'après le théorème PSP [15] :

$$\begin{gathered}\forall s,k,l : s \neq s_0 :: \\ s_0.requesting \wedge s_0.C[s_0] = k \wedge s.C[s] > l \\ \mapsto \\ s_0.mutex \vee (s_0.requesting \wedge s_0.C[s_0] = k \wedge s_0.C[s] > l)\end{gathered}$$

On en déduit, en choisissant $k = l = s_0$:

$$\begin{gathered}\forall s : s \neq s_0 :: \\ s_0.requesting \;\mapsto\; s_0.mutex \vee (s_0.requesting \wedge s_0.C[s_0] < s_0.C[s])\end{gathered} \tag{5.27}$$

Comme les horloges et leurs images sont des compteurs croissants, de (5.26), on déduit :

$$\forall s : s \neq s_0 :: s_0.requesting \wedge s_0.C[s_0] < s_0.C[s] \;\; \texttt{unless} \;\; s_0.mutex \tag{5.28}$$

En appliquant le théorème "completion" [16] aux propriétés (5.27) et (5.28), on déduit :

$$\begin{gathered}s_0.requesting \\ \mapsto \\ s_0.mutex \vee (s_0.requesting \wedge \langle \forall s : s \neq s_0 :: s_0.C[s_0] < s_0.C[s] \rangle)\end{gathered} \tag{5.29}$$

Comme le programme SR vérifie :

$$s_0.requesting \wedge \langle \forall s : s \neq s_0 :: s_0.C[s_0] < s_0.C[s] \rangle \;\; \texttt{unless} \;\; s_0.mutex$$

et que l'instruction "entrée en section critique" est finalement exécutée, on a :

$$s_0.requesting \wedge \langle \forall s : s \neq s_0 :: s_0.C[s_0] < s_0.C[s] \rangle \;\;\mapsto\;\; s_0.mutex$$

En appliquant le théorème "cancellation" [17] à la propriété (5.29), on obtient finalement :

$$s_0.requesting \;\mapsto\; s_0.mutex \tag{5.30}$$

D'après le lemme 2, on en déduit :

$$\forall k :: \; s_0.requesting \wedge s_0.C[s_0] = k \;\;\mapsto\;\; s_0.C[s_0] > k$$

Lemme 4 *Si s_0 est dans l'état thinking, il finit par incrémenter son horloge.*

Démonstration : De l'état *thinking*, un site peut soit rester dans cet état, soit aller vers l'état *requesting*. On a donc :

$$\begin{gathered}\forall k :: s_0.thinking \wedge s_0.C[s_0] = k \\ \mapsto \\ (s_0.thinking \wedge s_0.C[s_0] > k) \vee (s_0.requesting \wedge s_0.C[s_0] = k)\end{gathered}$$

15. Théorème PSP : $\dfrac{p \;\mapsto\; q \;,\; r \;\; \texttt{unless} \;\; b}{p \wedge r \;\;\mapsto\;\; (q \wedge r) \vee b}$

16. Théorème "completion" : $\dfrac{\langle \forall i \;::\; p_i \;\;\mapsto\;\; q_i \vee b \rangle \;,\; \langle \forall i \;::\; q_i \;\; \texttt{unless} \;\; b \rangle}{\langle \forall i \;::\; p_i \rangle \;\;\mapsto\;\; \langle \forall i \;::\; q_i \rangle \vee b}$, pour tout ensemble fini de prédicats p_i, q_i.

17. Théorème "cancellation" : $\dfrac{p \;\;\mapsto\;\; q \vee b \;,\; b \;\;\mapsto\;\; r}{p \;\;\mapsto\;\; q \vee r}$. Ici, on a $r \equiv q \equiv s_0.mutex$.

D'après le lemme 3 et le théorème "cancellation", on en déduit :

$$\forall k :: \; s_0.thinking \wedge s_0.C[s_0] = k \quad \mapsto \quad s_0.C[s_0] > k \tag{5.31}$$

Lemme 5 *Le minimum global des horloges croît à l'infini.*

Démonstration : en appliquant la disjonction finie du *leads-to* aux trois lemmes 2, 3 et 4, on obtient :

$$\forall k :: \; s_0.C[s_0] = k \;\mapsto\; s_0.C[s_0] > k$$

Comme $s_0.C[s_0]$ était la plus petite horloge, le minimum global des horloges finit par croître :

$$\forall k :: \; \langle \min s :: s.C[s] \rangle = k \;\mapsto\; \langle \min s :: s.C[s] \rangle > k$$

Par récurrence, on en déduit :

$$\forall k, l :: \; \langle \min s :: s.C[s] \rangle = k \;\mapsto\; \langle \min s :: s.C[s] \rangle > l$$

En utilisant la disjonction infinie du *leads-to*, on en déduit finalement :

$$\forall l :: \; \mathtt{true} \quad \mapsto \quad \langle \min s :: s.C[s] \rangle > l \tag{5.32}$$

Puisque le minimum global des horloges croît à l'infini, c'est que toutes les horloges croissent à l'infini. Comme un site en attente d'accès bloque son horloge, on en déduit qu'il finit par accéder à la section critique, ce qui termine la preuve de la propriété de vivacité.

5.5.4 Topologie d'observation

Dans l'algorithme que nous avons considéré, chaque site observe tous les autres sites. Par conséquent, le réseau de communication sous-jacent est complètement maillé. On peut modifier cette topologie d'observation sans remettre en question les propriétés de l'algorithme.

Par exemple, une structure d'anneau peut remplacer le réseau maillé. Dans ce cas, chaque site n'observe que son prédécesseur sur l'anneau. Cet anneau d'observation est décrit par la relation :

$$\forall s, s' \; : \; s \neq s' \; :: \; s.C[s'] \prec (pred.s).C[s']$$

Un site $Mutex(s)$ voit directement son prédécesseur. Par contre, il ne voit les autres sites que par l'intermédiaire des images qu'en possède son prédécesseur. Comme la relation d'observation est transitive, on déduit de cette relation la relation suivante :

$$\forall s, s' :: \; s.C[s'] \prec s'.C[s']$$

qui correspond à l'hypothèse de communication de la preuve précédente. L'algorithme reste donc correct si les communications entre les sites se font autour d'un anneau.

5.6 Bibliographie

L'approche traditionnelle des programmes répartis en UNITY est largement utilisée dans le livre [CM88]. La relation d'observation ainsi que quelques théorèmes sont décrits dans le rapport [CFM+96b]. La définition est reprise avec quelques théorèmes supplémentaires dans [CFM+96a]. Nous avons repris ici la plupart des théorèmes de ces deux papiers dont nous avons détaillé les preuves. Le mécanisme d'observation est succinctement décrit dans ces deux références. Il est détaillé dans [Cha97] où l'on trouve également la présentation de la sémantique $\mathcal{O}bs$ et la définition des programmes bien répartis. Parmi les exemples de programmes communiquant par observation, le protocole du bit alterné est traité de manière plus complète et démontré dans [CFM+96b]. Il est également repris, sous un angle différent dans [CFM+97]. L'algorithme d'exclusion mutuelle par front circulant s'inspire des algorithmes proposés par E.W. Dijkstra dans [Dij74]. La preuve de la relation de raffinement est reprise dans [CFM+97]. L'algorithme de détection de la terminaison de F. Mattern, ausi connu sous le nom d'algorithme des quatre compteurs, est présenté dans [Mat87]. La notion de vague est présentée notamment dans [Ray92]. La description de l'algorithme d'exclusion mutuelle à base d'horloge ainsi que sa preuve de correction sont publiés dans [CFM+96a]. L'algorithme choisi s'inspire de l'algorithme présenté par L. Lamport dans [Lam78].

Chapitre 6

Observation et causalité

Nous avons vu que le mécanisme d'observation peut être utilisé pour représenter la communication au sein d'un système réparti. Elle constitue alors une description plus abstraite et donc plus simple à manipuler que la description plus classique qui s'appuie sur des séquences de messages.

D'autre part, la simple affectation de variables peut être considérée comme une forme extrêmement simplifiée de communication. L'objectif de ce chapitre est d'examiner de plus près les liens qui unissent le mécanisme d'observation et l'affectation simple du modèle Unity. Plus précisément, nous cherchons dans quelle mesure la communication par observation diffère de la communication par simple affectation.

Nous allons voir que, contrairement à ce qu'on pourrait penser a priori, l'observation et l'affectation Unity ont de nombreux points communs. Nous verrons également qu'il subsiste une différence importante dans leur comportement vis-à-vis de la notion de causalité et nous montrerons qu'il s'agit là de la différence majeure.

Une première partie présente les similitudes qui existent entre le mécanisme d'observation et l'affectation dans le modèle Unity. En particulier, on montre que dans le cas d'une communication globale et unilatérale, l'observation et l'affectation permettent au système d'accéder aux mêmes états. Dans une deuxième partie, la différence de comportement vis-à-vis de la causalité entre l'observation et l'affectation est mise en évidence, d'abord à travers un exemple, puis de façon générale. Cette seconde partie se termine en montrant que, s'il n'y a pas d'influence de la causalité, l'observation et l'affectation permettent à un système d'accéder aux mêmes états.

6.1 Similitudes entre observation et affectation

6.1.1 Retard, perte et vivacité

Le mécanisme d'observation, tel qu'il a été défini, permet de réaliser entre différents processus une communication dont les principales caractéristiques sont :

- retard non borné entre l'évolution de la source et celle de l'image ;

- perte de valeurs de la source, due aux “sauts” possibles des fonctions horloges ;
- rafraîchissement infiniment fréquent de l’image par rapport à la source, garanti par la propriété de vivacité des fonctions horloges.

Au contraire, l’affectation correspond plutôt, a priori, à une communication instantanée de la valeur courante de la source, sans possibilité de perte.

Néanmoins, cette intuition est celle que l’on a lorsque l’on pense à l’affectation en elle-même. Or, il ne faut pas perdre de vue que l’affectation est utilisée ici dans le modèle d’exécution de UNITY dont une des caractéristiques est le grand non déterminisme.

Supposons donc qu’une communication soit réalisée par une simple affectation non gardée de l’image par la source. Dans ce cas, le non déterminisme de UNITY a trois conséquences :

- la source peut évoluer entre deux exécutions de l’affectation ;
- une exécution de l’affectation peut remplacer une valeur précédente de l’image avant que cette valeur ne soit utilisée ;
- si l’affectation est exécutée dans un état i, la valeur obtenue sera utilisée dans un état $j > i$. À ce moment-là, elle ne correspondra plus à la valeur courante de la source, mais à une valeur passée.

Les deux premiers points montrent que l’aspect de la perte de valeurs est déjà (doublement) présent dans l’affectation, et le dernier point montre que les valeurs obtenue par affectation sont utilisées en tant que valeurs passées de la source. De plus l’équité faible de UNITY permet un retard non borné de l’image sur la source, mais oblige à rafraîchir l’image infiniment souvent.

On voit donc que les propriétés qui paraissent être les caractéristiques d’une communication par observation se retrouvent intégralement dans une communication par affectation.

6.1.2 Cas de la communication globale et unilatérale

Considérons, à titre d’exemple, le cas d’un processus qui accède à des données d’un autre processus. On suppose que l’accès aux différentes données se fait globalement, c’est-à-dire qu’il nécessite *une seule* relation d’observation ou *une seule* affectation (Fig. 6.1).

On modélise ces deux processus à l’aide de deux programmes UNITY F et G bien répartis. L’ensemble des données du processus G qui sont vues par F est représenté par la variable x. La variable $`x$ de F est la vision que ce processus a de la valeur de x dans G. Cette variable peut être mise à jour soit par observation, soit par affectation. On obtient les deux composantes de communication suivantes :

Program C
Observe
  $`x \prec x$
End C
	Program C'
	Assign
	  $`x := x$
	End C'

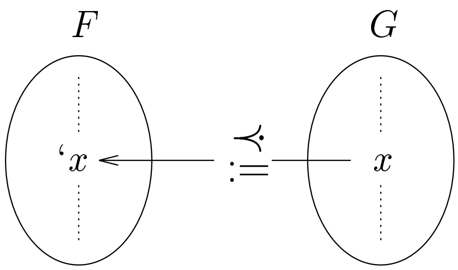

Fig. 6.1 – *Communication globale et unilatérale.*

Nous allons comparer les deux systèmes obtenus en termes d'invariants, c'est-à-dire d'accessibilité des états.

Étant donné, comme nous l'avons vu au paragraphe 5.3.2, que l'affectation constitue une implantation possible de l'observation, on a la relation :

$$F\llbracket C\rrbracket G \sqsubseteq_{\mathcal{O}bs} F\llbracket C'\rrbracket G$$

On en déduit que tout invariant de $F\llbracket C\rrbracket G$ est un invariant de $F\llbracket C'\rrbracket G$:

$$\frac{\texttt{invariant } P \texttt{ in } F\llbracket C\rrbracket G}{\texttt{invariant } P \texttt{ in } F\llbracket C'\rrbracket G}$$

Nous allons montrer que, réciproquement, les invariants de $F\llbracket C'\rrbracket G$ sont des invariants de $F\llbracket C\rrbracket G$, c'est à dire :

$$\frac{\texttt{invariant } P \texttt{ in } F\llbracket C'\rrbracket G}{\texttt{invariant } P \texttt{ in } F\llbracket C\rrbracket G} \tag{6.1}$$

Ainsi, quel que soit le moyen de communication choisi, observation ou affectation, le système possède le même ensemble d'états accessibles.

6.1.3 Démonstration

La démonstration utilise le fait que, en dehors de la relation de communication décrite par C ou C', les processus F et G sont totalement indépendants et qu'aucune hypothèse n'est faite sur leur vitesses d'exécution relatives. Le principe de la preuve est de montrer qu'un état du système $F\llbracket C\rrbracket G$ peut être atteint par $F\llbracket C'\rrbracket G$ à condition que le processus F soit "suffisamment rapide". En effet, si F est suffisamment rapide, il n'a besoin que d'accéder à la valeur courante de la variable x.

Notations

– On utilise les deux projections Π_F et Π_G qui permettent de restreindre un état σ_k du système aux seules variables de F ou de G. On note donc les états sous la forme d'un couple (a, b) où a (resp. b) correspond à la partie de l'état accessible par F (resp. G) :

$$\sigma_k = (a_k, b_k) \text{ avec } a_k = \Pi_F.\sigma_k \text{ et } b_k = \Pi_G.\sigma_k$$

- On note également $\sigma_k \setminus v$ pour décrire la projection de l'état σ_k sur les variables autres que v.

- Lorsque l'on insère, dans une séquence d'exécution σ, un nouvel état derrière l'état σ_k, cet état est noté $\sigma'_k = (a'_k, b'_k)$. Si l'on insère deux états, ils sont notés σ'_k et σ''_k, etc.

- Lorsqu'un état $\sigma_k = (a_k, b_k)$ d'une séquence σ est modifié par une manipulation de cette séquence, le nouvel état obtenu est noté $\bar{\sigma}_k = (\bar{a}_k, \bar{b}_k)$. S'il l'état obtenu est de nouveau modifié, il devient $\bar{\bar{\sigma}}_k$, etc.

- Une transition entre deux états successifs est notée $\sigma_k \longrightarrow \sigma_{k+1}$. Sur la flèche apparaît l'instruction qui engendre cette transition. S_F (resp. S_G) représente une instruction générique du processus F (resp. G). $S_F|S_G$ représente une instruction d'un des deux processus. Sous la flèche apparaissent les éventuelles modifications de variables dues au mécanisme d'observation, sous la forme $v \leftarrow e$. Par exemple, une transition peut être de la forme :
$$\sigma_k \xrightarrow[\text{`}x \leftarrow \sigma_{H.(k+1)}.x]{S_F|S_G} \sigma_{k+1}$$

Principe

Nous allons définir, par récurrence sur n, un procédé de modification d'une exécution σ de $F\|C\|G$ qui permet de remplacer les n premières mises à jour [1] de la variable ‘x dues à l'observation par des exécutions de l'instruction ‘$x := x$. La propriété fondamentale de ce procédé est que :

- les états qui suivent les mises à jour de ‘x par l'observation sont inchangés ;

- la partie modifiée de la séquence σ constitue toujours un préfixe d'une exécution possible du programme $F\|C'\|G$ dans la sémantique $\mathcal{O}bs$.

Par conséquent, un état de σ qui suit une mise à jour de ‘x est accessible par le programme $F\|C'\|G$. Donc pour un état quelconque σ_k de σ, on considère le dernier état qui précède σ_k et qui suit directement une mise à jour de ‘x. Cet état est accessible par le programme $F\|C'\|G$ et, puisque les instructions de $F\|C\|G$ sont des instructions de $F\|C'\|G$, l'état σ_k est également accessible. Par conséquent, l'exécution σ satisfait aux invariants du programme $F\|C'\|G$, ce qui démonte la propriété (6.1).

On choisit donc une exécution σ dans $\mathcal{O}bs.(F\|C\|G)$ et une fonction horloge H associée dans cette exécution à la relation ‘$x \prec\!\cdot\, x$.

Cas de la première mise à jour de ‘x

Cette mise à jour a lieu dans l'état dont l'indice est le plus petit entier k tel que $H.k > 0$. On note i cet indice et j représente $H.i$. La transition entre

1. En fait, on va considérer les modifications des valeurs prises par une fonction horloge associée à la relation ‘$x \prec\!\cdot\, x$. Ces modifications ne correspondent pas nécessairement à des modifications de la valeur de ‘x, mais toutes les modifications de ‘x sont prises en compte.

les états σ_{i-1} et σ_i affecte donc à ‘x la valeur $\sigma_j.x$, tout en exécutant une instruction d'un des deux processus :

$$\sigma_{i-1} \xrightarrow[\text{‘}x\leftarrow\sigma_{H.j}.x]{S_F|S_G} \sigma_i$$

D'après la définition de la relation d'observation, on a nécessairement $j \leqslant i$. On considère séparément les trois cas $j = i$, $j = i-1$ et $j < i-1$:

Cas $j = i$:
On remplace la transition $\sigma_{i-1} \xrightarrow[\text{‘}x\leftarrow\sigma_{H.j}.x]{S_F|S_G} \sigma_i$ par :

$$\sigma_{i-1} = (a_{i-1}, b_{i-1}) \xrightarrow{S_F|S_G} \bar{\sigma}_i = (\bar{a}_i, \bar{b}_i) \xrightarrow{\text{‘}x:=x} \sigma'_i = (a'_i, b'_i)$$

Les transitions $\xrightarrow[\text{‘}x\leftarrow\sigma_{H.j}.x]{S_F|S_G}$ et $\xrightarrow{S_F|S_G}$ ont le même effet sur $\sigma_{i-1} \setminus \text{‘}x$. Par conséquent, $\bar{b}_i = b_i$ et $(\bar{a}_i \setminus \text{‘}x) = (a_i \setminus \text{‘}x)$. D'autre part, l'exécution de l'affectation ‘$x := x$ est sans effet sur les variables autres que ‘x. On en déduit $b'_i = \bar{b}_i = b_i$ et $(a'_i \setminus \text{‘}x) = (\bar{a}_i \setminus \text{‘}x) = (a_i \setminus \text{‘}x)$.

De $\bar{b}_i = b_i$, on déduit que $\bar{\sigma}_i.x = \sigma_i.x$. Par conséquent, l'exécution dans $\bar{\sigma}_i$ de l'affectation ‘$x := x$ est équivalente à ‘$x \leftarrow \sigma_i.x$ et donc à ‘$x \leftarrow \sigma_j.x$. Donc $a'_i.\text{‘}x = \sigma_j.x = a_i.\text{‘}x$, et comme $(a'_i \setminus \text{‘}x) = (a_i \setminus \text{‘}x)$, on en déduit $a'_i = a_i$ et donc $\sigma'_i = \sigma_i$.

Comme le mécanisme d'observation n'y intervient plus, le segment $\sigma_0\sigma_1 \cdots \sigma_{i-1}\bar{\sigma}_i\sigma_i$ est bien préfixe d'une exécution de $F\|C'\|G$.

Cas $j = i-1$:
On distingue deux cas selon que l'instruction exécutée est une instruction de F ou une instruction de G. On remplace donc :

$$\sigma_{i-1} \xrightarrow[\text{‘}x\leftarrow\sigma_{H.j}.x]{S_F} \sigma_i \text{ par } \sigma_{i-1} = (a_{i-1}, b_{i-1}) \xrightarrow{S_F} \bar{\sigma}_i = (\bar{a}_i, \bar{b}_i) \xrightarrow{\text{‘}x:=x} \sigma'_i = (a'_i, b'_i)$$

et

$$\sigma_{i-1} \xrightarrow[\text{‘}x\leftarrow\sigma_{H.j}.x]{S_G} \sigma_i \text{ par } \sigma_{i-1} = (a_{i-1}, b_{i-1}) \xrightarrow{\text{‘}x:=x} \bar{\sigma}_i = (\bar{a}_i, \bar{b}_i) \xrightarrow{S_G} \sigma'_i = (a'_i, b'_i)$$

Par un raisonnement similaire au précédent, on montre dans les deux cas que $\sigma'_i = \sigma_i$ et que $\sigma_0\sigma_1 \cdots \sigma_{i-1}\bar{\sigma}_i\sigma_i$ est un préfixe d'une exécution de $F\|C'\|G$.

Cas $j < i-1$:
Les différentes transitions entre j et $i-1$ sont dues à des instructions de F ou de G, sans influence du mécanisme d'observation puisque l'on s'intéresse au cas de la première mise à jour de ‘x. On a donc un segment de la forme :

$$\sigma_j \xrightarrow{S_F|S_G} \sigma_{j+1} \xrightarrow{S_F|S_G} \cdots \xrightarrow{S_F|S_G} \sigma_{i-2} \xrightarrow{S_F|S_G} \sigma_{i-1} \xrightarrow[\text{‘}x\leftarrow\sigma_j.x]{S_F|S_G} \sigma_i$$

On commence par dissocier l'effet de la dernière instruction du segment de celui du mécanisme d'observation. On obtient le segment :

$$\begin{array}{l}\sigma_j = (a_j, b_j) \xrightarrow{S_F | S_G} \cdots \\ \sigma_{i-1} = (a_{i-1}, b_{i-1}) \xrightarrow{S_F | S_G} \bar{\sigma}_i = (\bar{a}_i, \bar{b}_i) \xrightarrow[\text{‘}x \leftarrow \sigma_j.x]{skip} \sigma'_i = (a'_i, b'_i)\end{array}$$

de la séparation des variables de F de celles de G, on déduit :

$$\bar{b}_i = b_i = b'_i \qquad (\bar{a}_i \setminus \text{‘}x) = (a_i \setminus \text{‘}x) \qquad a'_i = a_i$$

Le segment est donc en fait :

$$\begin{array}{l}\sigma_j = (a_j, b_j) \xrightarrow{S_F | S_G} \cdots \\ \sigma_{i-1} = (a_{i-1}, b_{i-1}) \xrightarrow{S_F | S_G} \bar{\sigma}_i = (\alpha, b_i) \xrightarrow[\text{‘}x \leftarrow \sigma_j.x]{skip} \sigma_i = (a_i, b_i)\end{array}$$

avec $(\alpha \setminus \text{‘}x) = (a_i \setminus \text{‘}x)$. Cette étape n'a donc pas modifié l'état final du segment : $\sigma'_i = \sigma_i$.

L'étape suivante consiste à permuter les exécutions des instructions de F et de G de manière à ce que les exécutions des instructions de F précèdent celles des instructions de G. Ceci correspond en fait à “accélérer” l'exécution du processus F.

Le système étant bien réparti, les instructions de F n'utilisent pas les variables de G et réciproquement. On en déduit que l'exécution de S_F puis de S_G équivaut à celle de S_G puis de S_F. Par conséquent, si :

$$(a, b) \xrightarrow{S_F} (\bar{a}, b) \xrightarrow{S_G} (\bar{a}, \bar{b})$$

alors :

$$(a, b) \xrightarrow{S_G} (a, \bar{b}) \xrightarrow{S_F} (\bar{a}, \bar{b})$$

On peut remarquer que les différentes projections sur G des états du système, c'est-à-dire b et $\bar{b}$, se retrouvent dans les états obtenus après la permutation. En particulier, toute valeur de la variable x avant la permutation est récupérable dans un état après la permutation. Nous utiliserons cette propriété pour prolonger le procédé que nous définissons aux autres mises à jour de ‘x.

En permutant les exécutions de S_F et de S_G dans le segment considéré, on obtient :

$$\begin{array}{c}\sigma_j = (a_j, b_j) \xrightarrow{S_F} \bar{\sigma}_{j+1} = (\bar{a}_{j+1}, \bar{b}_{j+1}) \xrightarrow{S_F} \cdots \\ \bar{\sigma}_k = (\bar{a}_k, \bar{b}_k) \xrightarrow{S_G} \bar{\sigma}_{k+1} = (\bar{a}_{k+1}, \bar{b}_{k+1}) \xrightarrow{S_G} \cdots \\ (\bar{\alpha}, \bar{b}_i) \xrightarrow[\text{‘}x \leftarrow \sigma_j.x]{skip} \bar{\sigma}_i = (\bar{a}_i, \bar{b}_i)\end{array}$$

Étant donnée la commutativité des instructions S_F et S_G, on a :

- $\bar{\alpha} = \alpha$ et par conséquent $\bar{a}_i = a_i$ et $\bar{b}_i = b_i$;
- $\langle \forall m : j \leqslant m \leqslant i :: \langle \exists m' : j \leqslant m' \leqslant i :: \sigma_m.x = \bar{\sigma}_{m'}.x \rangle\rangle$
  En d'autres termes, cette étape ne fait pas perdre de valeurs de x. Comme $b_i = \bar{b}_{i+1} = \bar{b}_{i+2} = \cdots = \bar{b}_k$, on peut affiner la formule précédente en :

$$\langle \forall m : j \leqslant m \leqslant i :: \langle \exists m' : k \leqslant m' \leqslant i :: \sigma_m.x = \bar{\sigma}_{m'}.x \rangle\rangle \qquad (6.2)$$

De $b_j = \bar{b}_k$, on déduit $\sigma_j.x = \bar{\sigma}_k.x$. On insère donc à l'indice k une exécution de l'instruction $`x := x$. On obtient le segment suivant :

$$\begin{aligned}
&\sigma_j = (a_j, b_j) \xrightarrow{S_F} \bar{\sigma}_{j+1} = (\bar{a}_{j+1}, b_{j+1}) \xrightarrow{S_F} \cdots \\
&\bar{\sigma}_k = (\bar{a}_k, b_j) \xrightarrow{`x:=x} \bar{\bar{\sigma}}_k = (\bar{\bar{a}}_k, b_j) \xrightarrow{S_G} \bar{\bar{\sigma}}_{k+1} = (\bar{\bar{a}}_{k+1}, \bar{b}_{k+1}) \xrightarrow{S_G} \cdots \\
&\qquad\qquad\qquad (\bar{\bar{\alpha}}, \bar{b}_i) \xrightarrow[`x \leftarrow \sigma_j.x]{skip} \bar{\bar{\sigma}}_i = (\bar{\bar{a}}_i, \bar{\bar{b}}_i)
\end{aligned}$$

On a $(\bar{\bar{a}}_k \setminus `x) = (\bar{a}_k \setminus `x)$. Comme les exécutions des instructions de G ne dépendent pas de la valeur de $`x$, on en déduit $\langle \forall m : k \leqslant m \leqslant i :: \bar{\bar{b}}_m = \bar{b}_m \rangle$. Comme $\bar{\bar{a}}_k = \bar{\bar{a}}_{k+1} = \cdots = \bar{\bar{\alpha}}$, on a aussi $(\bar{\bar{\alpha}} \setminus `x) = (\bar{\alpha} \setminus `x) = (a_i \setminus `x)$. Or, $\bar{\alpha}.`x = \bar{\bar{a}}_k.`x = \bar{a}_k.x = a_j.x$. Donc $\bar{\bar{a}}_i = \bar{\bar{\alpha}} = a_i$. Le segment est en fait :

$$\begin{aligned}
&\sigma_j = (a_j, b_j) \xrightarrow{S_F} \bar{\sigma}_{j+1} = (\bar{a}_{j+1}, b_{j+1}) \xrightarrow{S_F} \cdots \\
&\bar{\sigma}_k = (\bar{a}_k, b_j) \xrightarrow{`x:=x} \bar{\bar{\sigma}}_k = (\bar{\bar{a}}_k, b_j) \xrightarrow{S_G} \bar{\bar{\sigma}}_{k+1} = (\bar{\bar{a}}_{k+1}, \bar{b}_{k+1}) \xrightarrow{S_G} \cdots \\
&\qquad\qquad\qquad (a_i, b_i) \xrightarrow[`x \leftarrow \sigma_j.x]{skip} \sigma_i = (a_i, b_i)
\end{aligned}$$

On peut donc supprimer la dernière transition $\xrightarrow[`x \leftarrow \sigma_j.x]{skip}$. On obtient finalement le segment :

$$\begin{aligned}
&\sigma_j = (a_j, b_j) \xrightarrow{S_F} \bar{\sigma}_{j+1} = (\bar{a}_{j+1}, b_{j+1}) \xrightarrow{S_F} \cdots \\
&\bar{\sigma}_k = (\bar{a}_k, b_j) \xrightarrow{`x:=x} \bar{\bar{\sigma}}_k = (\bar{\bar{a}}_k, b_j) \xrightarrow{S_G} \bar{\bar{\sigma}}_{k+1} = (\bar{\bar{a}}_{k+1}, \bar{b}_{k+1}) \xrightarrow{S_G} \cdots \\
&\sigma_i = (a_i, b_i)
\end{aligned}$$

qui mène à l'état σ_i en utilisant une exécution de $`x := x$ à la place du mécanisme d'observation et qui constitue donc un préfixe d'une exécution de $F[\![C']\!]G$.

Cas des autres mises à jour de $`x$

Hypothèse de récurrence : on suppose que l'on peut utiliser le procédé précédent jusqu'à l'indice $i-1$ pour remplacer dans σ les mises à jour de $`x$ dues au mécanisme d'observation par des exécutions de l'instruction $`x := x$, sans modifier l'état qui suit la mise à jour et en obtenant un préfixe d'une exécution de $F[\![C']\!]G$.

On considère la transition $\sigma_{i-1} \longrightarrow \sigma_i$. Si cette transition ne correspond pas à une mise à jour de la variable $`x$ (c'est-à-dire si $H.(i-1) = H.i$), le procédé précédemment utilisé est valable jusqu'à l'indice i.

Si, au contraire, $H.(i-1) \neq H.i$, on note $j = H.i$. D'après la formule (6.2), la valeur $\sigma_j.x$ est accessible dans un état $\sigma_{j'}$ de σ *qui suit* toutes les exécutions des affectations $`x := x$ déjà introduites. On considère donc le segment constitué des états dont l'indice est compris entre j' et i. Dans ce segment :

– il n'y a que des exécutions d'instructions de F ou de G ;

– la seule influence du mécanisme d'observation a lieu lors de la transition $\sigma_{i-1} \longrightarrow \sigma_i$.

Par conséquent, le cas est strictement identique au cas de la première mise à jour et le même procédé s'applique.

Par conséquent, on dispose bien d'un procédé permettant de transformer tout préfixe d'une exécution de $F[\![C]\!]G$ en un préfixe d'une exécution de $F[\![C']\!]G$ sans modifier les états qui suivent les mises à jour de la variable $`x$.

6.2 Différences entre observation et affectation

6.2.1 Exemple

Malgré les similitudes présentées au paragraphe précédent, nous allons voir que les communications par des affectations et par le mécanisme d'observation diffèrent sur un point important.

Nous commençons par montrer, sur un exemple, que la propriété d'identité des invariants vue au paragraphe précédent n'est pas valide en général.

On considère les trois processus P_0, P_1, P_2 suivants :

```
Program P0
Declare
   ‘x, ‘y : Int;
   p, q, r : Bool;
Initially
   ¬p ∧ ¬q ∧ ¬r;
Assign
   p := (‘x = 10)
[]
   q := (‘y = 6) if p
[]
   r := (‘y = 7) if q
End P0
```

```
Program P1
Declare
   x, z : Int;
Initially
   x = 0;
Assign
   x := x + 1 if x < z
End P1
```

```
Program P2
Declare
   y : Int;
Initially
   y = 0;
Assign
   y := y + 1
End P2
```

Les deux processus P_1 et P_2 gèrent chacun un compteur croissant, respectivement x et y. Le processus P_0 récupère les valeurs de ces compteurs, et positionne le booléen r s'il voit successivement le compteur x à 10, puis le compteur y à 6 puis à 7.

Le processus P_1 utilise une image z du compteur y pour garantir que son compteur x reste inférieur à y. On intègre donc la relation $\prec y$ à la composante de communication. On considère les deux composantes de communication C et C' selon que P_0 obtient les valeurs des compteurs par le mécanisme d'observation ou par des affectations :

```
Program C
Observe
   z ≺ y    ∧
   ‘x ≺ x   ∧
   ‘y ≺ y
End C
```

```
Program C’
Observe
   z ≺ y
Assign
   ‘x := x
[]
   ‘y := y
End C’
```

On voit facilement que, pour que $`x$ atteigne la valeur 10, il faut que $x$ et $y$ aient déjà atteint ou dépassé cette valeur. Dans ce cas, si la communication se fait par des affectations, $`y$ ne peut plus prendre que des valeurs au moins égales à 10. Donc, même si la variable $`y$ possède déjà l'une des deux valeurs 6 ou 7, elle ne pourra pas prendre l'autre valeur ensuite. Par conséquent, dans le cas de la communication par des affectations, le booléen r ne peut être positionné :

$$\texttt{invariant } \neg r \texttt{ in } P_0 \| P_1 \| P_2 \| C'$$

Par contre, dans le cas d'une communication par le mécanisme d'observation, rien n'empêche $`y$ de prendre successivement les valeurs 6 et 7, même alors que y a dépassé la valeur 10. L'invariant précédent n'est donc pas valide dans $P_0 \| P_1 \| P_2 \| C$.

Intuitivement, la différence de comportement entre le système communiquant par affectations et celui qui utilise le mécanisme d'observation provient du fait que l'observation permet de mémoriser *plusieurs* valeurs du passé de la source alors que lorsqu'on utilise une affectation, on ne dispose que d'*une* valeur du passé de la source.

6.2.2 Observation, affectation et causalité

Nous considérons une application répartie composée de $n + 1$ processus $P_0, P_1, \cdots, P_n$:

$$P \;=\; \langle \| i : 0 \leqslant i \leqslant n :: P_i \rangle$$

Nous nous intéressons plus particulièrement à la façon dont le processus P_0 perçoit son environnement $\langle \| i : 1 \leqslant i \leqslant n :: P_i \rangle$. Nous considérons les deux cas de communication, par le mécanisme d'observation ou par des affectations (Fig. 6.2).

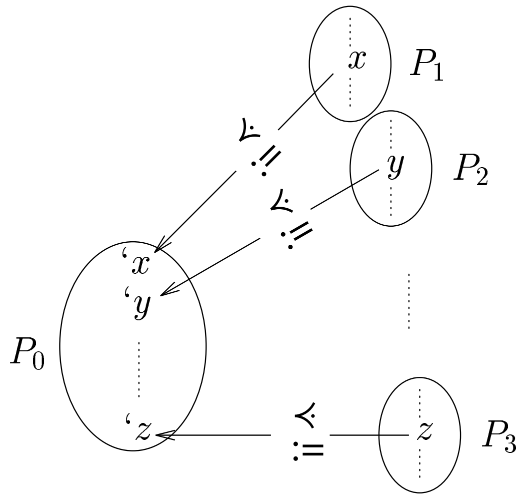

Fig. 6.2 – *Perception par P_0 de son environnement.*

Le processus P_0, à travers les valeurs qu'il possède des variables de son environnement, perçoit les événements de cet environnement que sont les changements de valeurs des variables.

Nous avons vu que la perception des événements liés à une même variable était assez semblable, que la communication s'effectue par des affectations ou par le mécanisme d'observation : les événements sont vus dans l'ordre chronologique, avec un retard non borné et des possibilités de pertes.

La question que nous considérons dans ce paragraphe est celle de la perception de liens existant entre des événements liés à des variables différentes. En particulier, nous nous intéressons à la façon dont le processus P_0 perçoit les liens de causalité qui peuvent exister entre les événements de changement de valeur des variables de son environnement.

On considère x et y, deux variables de l'environnement de P_0 supposées situées sur des processus différents (autres que P_0). Le processus P_0 dispose de deux copies de ces variables, respectivement $`x$ et $`y$.

Cas d'une communication par observation

Les variables $`x$ et $`y$ sont mises à jour par le mécanisme d'observation. On suppose donc que la section *observe* du système comporte les deux relations $`x \prec x$ et $`y \prec y$.

Soit σ une exécution du système considéré. On s'intéresse, dans σ, à l'événement e_1 (resp. e_2) résultant d'un changement de valeur de la variable x (resp. y). On a donc deux indices i_1 et i_2 tels que $\sigma_{i_1}.x \neq \sigma_{i_1+1}.x$ et $\sigma_{i_2}.y \neq \sigma_{i_2+1}.y$.

On choisit un indice i_3 tel que $i_3 > i_1$ et $i_3 > i_2$ et on considère différentes extensions possibles du préfixe $\sigma_0\sigma_1 \cdots \sigma_{i_3}$. On suppose que, dans ce préfixe, les images $`x$ et $`y$ n'ont encore jamais été mises à jour, c'est-à-dire :

$$\langle \forall i : 0 \leqslant i \leqslant i_3 :: \sigma_i.`x = \sigma_0.x \ \wedge\ \sigma_i.`y = \sigma_0.y \rangle$$

On définit les fonctions H_1, H_2, H'_1 et H'_2, de $\mathbb{N}$ dans $\mathbb{N}$, de la façon suivante :

$$\begin{array}{ll}
\langle \forall t : 0 \leqslant t \leqslant i_3 :: H_1.t = 0 \rangle & \langle \forall t : 0 \leqslant t \leqslant i_3 + 2 :: H_2.t = 0 \rangle \\
H_1.(i_3+1) = i_1 & H_2.(i_3+3) = i_2 \\
H_1.(i_3+2) = i_1 + 1 & H_2.(i_3+4) = i_2 + 1 \\
\langle \forall t : t > i_3 + 2 :: H_1.t = t \rangle & \langle \forall t : t > i_3 + 4 :: H_2.t = t \rangle \\
\\
\langle \forall t : 0 \leqslant t \leqslant i_3 + 2 :: H'_1.t = 0 \rangle & \langle \forall t : 0 \leqslant t \leqslant i_3 :: H'_2.t = 0 \rangle \\
H'_1.(i_3+3) = i_1 & H'_2.(i_3+1) = i_2 \\
H'_1.(i_3+4) = i_1 + 1 & H'_2.(i_3+2) = i_2 + 1 \\
\langle \forall t : t > i_3 + 4 :: H'_1.t = t \rangle & \langle \forall t : t > i_3 + 2 :: H'_2.t = t \rangle
\end{array}$$

On voit facilement que les quatre fonctions H_1, H_2, H'_1 et H'_2 satisfont aux trois contraintes des fonctions horloges définies au paragraphe 5.2.2. Par conséquent, en associant :

$$\begin{array}{rl}
 & H_1 \text{ à la relation } `x \prec x \text{ et } H_2 \text{ à la relation } `y \prec y \\
\text{ou} & H'_1 \text{ à la relation } `x \prec x \text{ et } H'_2 \text{ à la relation } `y \prec y \\
\text{ou} & H_1 \text{ à la relation } `x \prec x \text{ et } H'_2 \text{ à la relation } `y \prec y
\end{array}$$

nous obtenons trois extensions, valides pour la sémantique $\mathcal{O}bs$, du préfixe $\sigma_0\sigma_1 \cdots \sigma_{i_3}$.

Dans ces trois extensions, le processus P_0 perçoit les événements e_1 et e_2. Cependant, dans le premier cas, il perçoit e_1 avant e_2, dans le deuxième cas, il perçoit e_2 avant e_1, et dans le dernier cas, il perçoit les deux événements simultanément.

Ceci montre que le lien éventuel de causalité entre les événements e_1 et e_2 n'est pas perçu par le processus P_0. En d'autres termes, la communication par le mécanisme d'observation ne transmet pas les liens de causalité.

Cas d'une communication par affectations

Nous allons montrer, sur un exemple, que si le processus P_0 voit son environnement à travers une communication par affectations, il est capable de connaître certaines relations de causalité de cet environnemnt.

Pour cela, nous reprenons le système $P_0 \| P_1 \| P_2 \| C'$ du paragraphe 6.2.1. La communication $z \prec\cdot\ y$ introduit un lien de causalité entre les événements de P_1 et ceux de P_2. On considère l'événement e_1 qui représente le passage, pour la variable x, de la valeur 9 à la valeur 10, et l'événement e_2 qui représente le passage, pour la variable y, de la valeur 6 à la valeur 7.

Toute exécution du système possède une et une seule occurrence de chacun des événements e_1 et e_2. Les variables x, y et z sont des compteurs croissants et satisfont l'invariant $x \leqslant y \leqslant z$. Par conséquent, l'événement e_2 précède l'événement e_1 dans toute exécution du système, c'est-à-dire que e_2 précède causalement e_1.

Si, dans une exécution du système, le processus P_0 perçoit l'événement e_1, c'est-à-dire que $`x$ passe de la valeur 9 à la valeur 10, alors le processus $P_0$ ne pourra pas, par la suite, percevoir l'événement $e_1$. En effet, comme nous l'avons vu précédemment, à partir du moment où $`x$, et donc x, a pris la valeur 10, tous les états σ_i qui suivent satisfont $\sigma_i.y \geqslant 10$. Par conséquent, aucune exécution de l'instruction $`y := y$ ne peut faire passer $`y$ de la valeur 6 à la valeur 7. On en déduit que, si le processus P_0 perçoit les événements e_1 et e_2, il percevra nécessairement e_2 avant e_1. En d'autres termes, le lien de causalité entre e_1 et e_2 lui apparaît.

6.2.3 Existence d'autres différences

Nous venons de voir que les communications par observation et par affectations diffèrent dans leur transfert de la causalité. Dans ce paragraphe, nous allons montrer que, du point de vue de l'accessibilité des états, ce comportement vis-à-vis de la causalité constitue la seule différence entre la communication par observation et la communication par affectations.

Pour cela, nous allons montrer que, s'il n'y a pas de liens de causalité entre les variables sources, le système communiquant par le mécanisme d'observation et le système communiquant par des affectations ont les mêmes invariants.

On considère le processus P_0 et son environnement. Pour représenter simplement le fait que dans l'environnemnt $\langle \| i : 1 \leqslant i \leqslant n :: P_i \rangle$ il n'y a pas de liens de causalité entre les variables $x, y, \cdots, z$, nous allons supposer que le processus P_0 voit son environnement à travers un filtre qui lui masque les liens de causalité entre ces variables (Fig. 6.3).

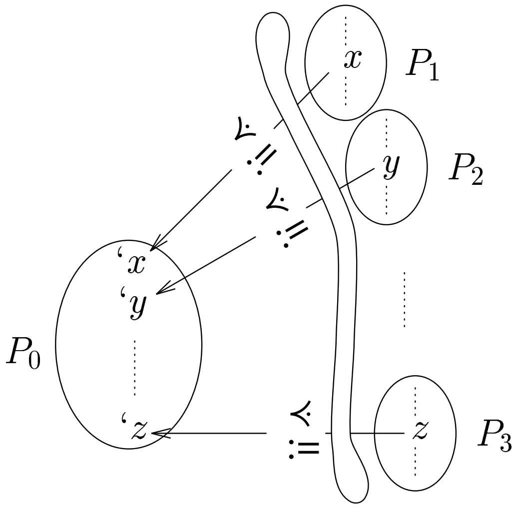

Fig. 6.3 – *Non perception par P_0 de la causalité.*

Le rôle de ce filtre est de brouiller la perception que P_0 a de son environnement, tout en ne modifiant pas le comportement de cet environnement. Pour cela, il doit être capable, à partir d'une exécution de $\langle [\!] i : 1 \leqslant i \leqslant n :: P_i \rangle$, d'engendrer toutes les exécutions équivalentes aux liens de causalité des sources près. Autrement dit, on doit retrouver, en sortie du filtre, tous les entrelacements possibles des histoires des variables sources correspondant à cette exécution.

Nous allons utiliser le mécanisme d'observation pour simuler un tel filtre. En effet, nous avons vu précédemment que la communication par observation ne transmet pas les liens de causalité. Par conséquent, si P_0 ne voit plus qu'une image par observation de son environnement, les liens de causalité lui seront bien masqués. En fait, ce procédé ne correspond pas tout à fait au filtre tel que nous l'avons spécifié. En effet, le passage aux images induit en plus une perte de valeurs. Néanmoins, puisque les moyens de communication que nous comparons possèdent tous deux la possibilité de perte de valeurs, cet aspect n'a pas d'influence [2].

On renomme donc, dans les processus $P_1, P_2, \cdots, P_n$, les variables $x, y, \cdots, z$ en $xx, yy, \cdots, zz$ qui sont supposées être de nouvelles variables, et on ajoute au système le filtre FC suivant :

```
Program FC
Observe
   x ≺· xx    ∧
   y ≺· yy    ∧
   ...
   z ≺· zz
End FC
```

2. On pourrait, en utilisant la superposition et des séquences de messages, implanter un filtre sans pertes. Néanmoins, celui-ci présenterait l'inconvénient d'être beaucoup plus complexe.

On considère donc les systèmes :

$$P_0 \| C \| FC \| \langle \| i : 1 \leqslant i \leqslant n :: P_i \rangle \qquad \text{et} \qquad P_0 \| C' \| FC \| \langle \| i : 1 \leqslant i \leqslant n :: P_i \rangle$$

où C correspond à une communication de $\langle \| i : 1 \leqslant i \leqslant n :: P_i \rangle$ vers P_0 par observation et C' correspond à la même communication, mais par affectations. On a alors la propriété :

$$\begin{array}{c} \texttt{invariant}\ P\ \texttt{in}\ P_0 \| C \| FC \| \langle \| i : 1 \leqslant i \leqslant n :: P_i \rangle \\ \equiv \\ \texttt{invariant}\ P\ \texttt{in}\ P_0 \| C' \| FC \| \langle \| i : 1 \leqslant i \leqslant n :: P_i \rangle \end{array} \tag{6.3}$$

Démonstration : Il suffit de montrer que :

$$\frac{\texttt{invariant}\ P\ \texttt{in}\ P_0 \| C' \| FC \| \langle \| i : 1 \leqslant i \leqslant n :: P_i \rangle}{\texttt{invariant}\ P\ \texttt{in}\ P_0 \| C \| FC \| \langle \| i : 1 \leqslant i \leqslant n :: P_i \rangle}$$

la propriété inverse découlant directement du fait que l'affectation est une implantation particulière du mécanisme d'observation.

Soit σ une exécution de $\mathcal{O}bs.(P_0 \| C \| FC \| \langle \| i : 1 \leqslant i \leqslant n :: P_i \rangle)$. Nous appliquons la même méthode que pour la démonstration du paragraphe 6.1.3 : nous montrons que les états qui suivent les mises à jour des images par le mécanisme d'observation sont accessibles par un préfixe d'une exécution de $P_0 \| C' \| FC \| \langle \| i : 1 \leqslant i \leqslant n :: P_i \rangle$.

On considère la relation $`x \prec x$, les autres relations étant traitées indépendamment et de la même manière, et on s'intéresse à la première mise à jour de $`x$ dans σ. Avec les mêmes notations qu'au paragraphe 6.1.3, on considère un segment de la forme :

$$\sigma_i \xrightarrow{S} \cdots \xrightarrow{S} \sigma_{i-1} \xrightarrow[`x \leftarrow \sigma_j.x]{S} \sigma_i$$

Comme précédemment, on transforme ce segment en :

$$\sigma_i \xrightarrow{S} \cdots \xrightarrow{S} \sigma_{i-1} \xrightarrow{S} \bar{\sigma}_i \xrightarrow[`x \leftarrow \sigma_j.x]{skip} \sigma_i$$

Soit K une fonction horloge associée à la relation $x \prec xx$ pour l'exécution σ. On définit la fonction $\bar{K}$ par :

- $\langle \forall t : 0 \leqslant t \leqslant j :: \bar{K}.t = K.t \rangle$;
- $\langle \forall t : j < t \leqslant i :: \bar{K}.t = K.j \rangle$;

Sur l'intervalle $[0..i]$, la fonction K satisfait aux contraintes de sûreté des fonctions horloges. Par conséquent, en choisissant les mêmes instructions que pour σ, il existe une exécution σ^0 de $P_0 \| C' \| FC \| \langle \| i : 1 \leqslant i \leqslant n :: P_i \rangle$ pour laquelle $\langle \forall t : 0 \leqslant t \leqslant i :: \sigma^0_t.x = \sigma^0_{\bar{K}.t}.xx \rangle$ et dont les $i+1$ premiers états ne diffèrent de ceux de σ que par la valeur de x. En exécutant à la suite de ce préfixe l'instruction $`x := x$, on affecte à $`x$ la valeur $\sigma^0_i.x$, c'est-à-dire $\sigma^0_{\bar{K}.i}.xx$, c'est-à-dire $\sigma_{K.j}.xx$, c'est-à-dire $\sigma_j.x$. On obtient donc nécessairement l'état σ_i.

On traite les autres mises à jour de ‘x en modifiant la fonction horloge K de la même manière, ce qui conclue la démonstration.

En conclusion, on voit donc finalement que :

- le mécanisme d'observation diffère d'autres moyens de communication plus élaborés par son non transfert des liens de causalité. Le rôle de l'observation se limite à la stricte communication des valeurs des variables sources ;
- dans tous les cas où la causalité n'intervient pas (cas d'une source unique, absence/masquage des liens de causalité), l'affectation et l'observation réalisent la même forme de communication. En particulier, l'affectation prend en compte le délai de communication non borné et la perte de valeurs.

Il faut noter que, si l'affectation transmet parfois des relations de causalité, elle ne les transmet pas toutes et ne constitue donc pas un moyen général de réaliser une communication causale.

Chapitre 7

Placement de programmes répartis

Le formalisme Unity a été conçu pour dissocier la conception des programmes de l'architecture sur laquelle ils vont s'exécuter. Le formalisme n'est donc lié a aucun type d'architecture a priori, mais une étape dite de placement (*mapping*) est nécessaire pour préciser de quelle manière un programme donné doit s'exécuter sur une architecture donnée.

Jusqu'à présent, nous avons décrit les programmes répartis par des *unions* de processus Unity. Dans ce chapitre, nous nous focalisons sur le placement de ces programmes répartis, c'est-à-dire la manière dont les processus Unity et le mécanisme de communication se retrouvent dans la réalité d'une architecture répartie.

En particulier, nous constatons que la sémantique de l'opérateur d'union ne permet pas de prendre en compte le parallélisme réel susceptible d'exister entre les différents sites. En effet, l'union impose une exclusion mutuelle entre les exécutions d'instructions de sites différents. Cette exclusion mutuelle n'est pas garantie par une architecture répartie asynchrone.

Néanmoins, les caractéristiques particulières des programmes répartis, et notamment l'absence de mémoire commune, ainsi que l'utilisation d'un modèle abstrait de communication fondé sur l'observation, permettent de proposer une extension de l'union qui prend en compte cet aspect de la répartition. Nous définissons donc une nouvelle forme d'union compatible avec le parallélisme réel d'un système réparti. Pour un programme donné, le passage de la description par union à une description s'appuyant sur ce nouvel opérateur constitue une formalisation de l'étape de placement.

Dans le cadre de programmes bien répartis, cette transformation préserve les principales propriétés du programme. Par conséquent, on peut raisonner sur un programme réparti décrit classiquement à l'aide de l'union, puis transposer cette description sur une architecture répartie de façon systématique.

Dans tout le chapitre, nous ne considérerons que des programmes bien répartis dans la sémantique opérationnelle $\mathcal{O}bs$. Les propriétés de la logique Unity ne sont pas indicées, mais correspondent implicitement aux propriétés $prop_{\mathcal{O}bs}$.

7.1 Description du problème

Lorsqu'on décrit des systèmes répartis dans le formalisme Unity à l'aide de l'opérateur d'union, on est amené à considérer des programmes de la forme :

$$P \;=\; \langle [\![i :: P_i \rangle$$

Le placement naturel d'un tel programme sur une architecture répartie consiste à associer chaque processus P_i à un site physique d'exécution. Deux instructions issues de deux processus différents seront donc exécutées par deux processeurs différents.

Or, l'opérateur d'union ne fait que rassembler les différentes instructions des processus P_i au sein de la section *assign* du programme P. De part la définition du modèle Unity, ces instructions se retrouvent donc en exclusion mutuelle deux à deux, y compris dans le cas d'instructions issues de processus P_i différents.

Cette exclusion mutuelle entre instructions de processus différents ne correspond pas à la réalité d'un système réparti. En effet, il est impossible de forcer cette contrainte entre des processus qui ne partagent pas d'horloge : rien n'empêche a priori plusieurs sites indépendants d'exécuter chacun une instruction au même instant. Le non respect de cette contrainte d'exclusion mutuelle par les applications réparties peut introduire une incohérence dans la modélisation du système comme le montre l'exemple suivant.

On considère deux processus représentés par les programmes Unity suivants :

Program F	**Program** G
Declare	**Declare**
x : Int;	y : Int;
Initially	**Initially**
x = 0;	y = 0;
Assign	**Assign**
x := 1	y := 1
End F	**End** G

L'exécution concurrente de ces deux processus est alors représentée par l'union $F[\![G$ qui n'est elle-même qu'une représentation du programme :

Program F [] G
Declare
  x, y : Int;
Initially
  $(x = 0) \wedge (y = 0)$;
Assign
  x := 1
[]
  y := 1
End F [] G (7.1)

Si l'on considère les deux processus F et G comme constituant un système réparti, le comportement :

$$(x, y) : (0, 0) \to (1, 1) \to (1, 1) \to \ldots \tag{7.2}$$

doit être admis comme possible, puisqu'il correspond au cas où les deux processus exécutent leur instruction simultanément. Or, cette séquence d'états est impossible pour le programme $F \| G$ puisque les deux instructions s'excluent mutuellement au sein de ce programme.

Cette non concordance entre la sémantique opérationnelle du système réparti et celle du programme est extrêmement gênante étant donné qu'elle permet de prouver au sein du modèle UNITY des propriétés du programme composé qui ne sont pas respectées par le système des processus. Par exemple :

$$x + y = 0 \;\mapsto\; x + y = 1$$

est démontrable pour le programme $F \| G$, mais est invalidée lors de l'exécution (7.2) des deux processus F et G dans un contexte réparti.

Il apparaît donc que le placement traditionnel des programmes UNITY composés par union ne convient pas pour les architectures réparties. Un tel placement n'est pas capable de prendre en compte tout le parallélisme entre les processus d'une application répartie.

Dans la suite de ce chapitre, nous définissons une autre forme de placement qui prend en compte l'exécution simultanée d'instructions par des sites différents. Cette définition et les résultats qui en découlent s'appliquent à des programmes bien répartis.

7.2 Un opérateur de composition de processus UNITY

Le mécanisme d'observation permet de représenter la communication entre des processus UNITY composés par l'opérateur d'union. On a vu au paragraphe 5.3.4 que cette communication s'effectue alors en parallèle avec les transitions du système et non au cours de transitions supplémentaires. Cette forme de communication est donc compatible avec le parallélisme induit par le placement d'un programme sur une architecture répartie. Néanmoins, le problème présenté au paragraphe précédent demeure : des instructions de calcul réalisées par des processus distincts ne peuvent pas s'exécuter simultanément.

Dans ce paragraphe, nous définissons une nouvelle forme de composition, inspirée de l'union, mais en accord avec les hypothèses d'une exécution répartie. En particulier, cette forme de composition prend en compte l'exécution concurrente d'instructions situées sur des sites différents. Par conséquent, le placement d'un programme ainsi composé sur une architecture répartie ne soulève plus la difficulté précédente.

7.2.1 Principe

Les instructions d'un programme UNITY étant *atomiques* par définition, le seul cas de chevauchement possible entre les exécutions de deux instructions est celui de la simultanéité complète[1]. En effet, deux instructions atomiques

1. Si l'on est amené à considérer un chevauchement partiel entre instructions, c'est que la granularité choisie pour la modélisation n'est pas la bonne.

sont considérées comme sans durée et ne peuvent s'exécuter qu'à deux instants distincts (c'est-à-dire l'une entièrement avant l'autre) ou au même instant. C'est ce dernier cas qui n'est pas pris en compte par l'opérateur d'union.

L'idée de l'extension proposée consiste à représenter la simultanéité possible de plusieurs instructions par l'ajout de nouvelles affectations. Chacune de ces instructions supplémentaires simule l'exécution simultanée de plusieurs instructions.

Par exemple, le programme (7.1) pourrait avoir la section *assign* suivante :

$$\begin{array}{ll} \textbf{Assign} & x := 1 \\ [\!] & y := 1 \\ [\!] & x, y := 1, 1 \end{array}$$

Les exécutions possibles de ce programme dans le modèle UNITY coïncident avec les exécutions possibles des deux processus $F : x := 1$ et $G : y := 1$ en parallèle, y compris la possibilité d'exécution simultanée d'une instruction sur chaque site. Par conséquent, les propriétés prouvées sur ce programme resteront valides lors d'une exécution sur une architecture répartie.

7.2.2 Définition de l'opérateur de produit

Nous formalisons le principe présenté au paragraphe précédent en définissant un nouvel opérateur de composition de programmes, baptisé *produit* et noté $[\![\!]\!]$.

Nous avons supposé, jusqu'à présent, qu'un programme UNITY contenait toujours l'instruction *skip*. À partir de maintenant, nous supposons en outre l'existence, dans tout programme UNITY, de l'instruction $fair := \neg fair$. La variable booléenne *fair* n'est jamais déclarée mais fait partie de l'état de tout programme. Un état d'un programme F est donc une fonction de $F.declare \cup \{fair\}$. La variable *fair* n'est utilisée que dans la définition du produit. Elle ne doit pas être utilisée ailleurs dans le programme ou dans ses spécifications.

Étant donnés deux programmes UNITY (étendu par la section *observe*) F et G tels que $F[\!]G$ soit bien réparti, on définit le produit $F[\![\!]\!]G$ de la façon suivante :

- $$\begin{array}{rcl} (F[\![\!]\!]G).assign & = & (F[\!]G).assign \\ & \cup & \{(s_F \| s_G) \texttt{ if } fair \mid s_F \in F.assign \wedge s_G \in G.assign\} \end{array}$$
- les autres sections de $F[\![\!]\!]G$ sont celles de $F[\!]G$.

Le programme $F[\!]G$ étant bien réparti, les programmes F et G ne partagent pas de variables hors des observations. Les instructions composées $s_F \| s_G$ constituent donc bien des instructions UNITY valides (non contradictoires et déterministes).

La notation "$(s_F \| s_G)$ `if` *fair*" signifie simplement que la conjonction "$\wedge fair$" est ajoutée aux gardes des instructions s_F et s_G. La variable *fair* est utilisée pour casser l'équité faible de UNITY. En effet, les instructions de la forme $s_F \| s_G$ ne représentent pas des instructions réelles des processus, mais seulement l'exécution simultanée de ces instructions. Par conséquent, elles ne nécessitent aucune forme d'équité. Si l'équité faible était maintenue pour ces instructions, cela signifierait que l'on considère que les coïncidences d'exécutions doivent avoir lieu infiniment souvent. Une telle obligation ne serait pas en accord avec le comportement d'un système réparti asynchrone.

Néanmoins, en UNITY, la contrainte d'équité faible s'applique à toutes les instructions de la section *assign*. C'est pourquoi il est nécessaire de supprimer cette contrainte pour les instructions supplémentaires ajoutées par le produit. Grâce à la variable *fair*, ces instructions ne sont jamais tenues d'être exécutées.

La définition du produit $[\![\!]\!]$ permet de prendre en compte les exécutions simultanées d'instructions de processus différents en introduisant implicitement dans la section *assign* les transitions correspondantes. L'intérêt de cette définition réside dans le fait que les exécutions possibles des processus F et G en milieu réparti coïncident avec les exécutions possibles de $F[\![\!]\!]G$ dans le modèle UNITY.

Nous verrons au paragraphe 7.3 que des propriétés de $F[\![\!]\!]G$ peuvent se déduire de celles de $F[\!]G$ et donc qu'il n'est pas nécessaire de considérer explicitement les instructions supplémentaires. Ces instructions, généralement très nombreuses, sont utilement masquées par les théorèmes de raffinement présentés dans ce paragraphe.

7.2.3 Propriétés élémentaires

Ce paragraphe présente quelques propriétés élémentaires de l'opérateur de produit. Ces propriétés facilitent les raisonnements de la partie suivante, et en particulier la démonstration des théorèmes pour un nombre quelconque de processus. Elle sont purement syntaxiques : deux programmes sont égaux s'ils ont le même ensemble d'instructions.

$$\begin{aligned} F[\![\!]\!]G &= G[\![\!]\!]F \\ F[\![\!]\!](G[\![\!]\!]H) &= (F[\![\!]\!]G)[\![\!]\!]H = F[\![\!]\!]G[\![\!]\!]H \\ F[\![\!]\!](G[\!]H) &= (F[\![\!]\!]G)[\!](F[\![\!]\!]H) \end{aligned}$$

Les deux premières propriétés montrent que l'opérateur de produit conserve la commutativité et l'associativité de l'union. Elles reposent sur la commutativité et l'associativité du $\|$ lorsqu'il est défini. La dernière propriété montre que l'opérateur $[\![\!]\!]$ distribue sur l'opérateur $[\!]$.

7.3 La transformation de l'union en produit

7.3.1 Présentation

L'opérateur de produit défini précédemment permet de décrire la composition de plusieurs processus UNITY de manière à prendre en compte les coïncidences d'exécution susceptibles de se produire lors d'une exécution répartie. L'utilisation de cet opérateur facilite donc l'étape de placement sur une architecture répartie. Par contre, il faut a priori considérer explicitement les nouvelles instructions de la forme $s_F \| s_G$ pour démontrer des propriétés sur un programme de la forme $F[\![\!]\!]G$. Or, ces instructions peuvent rapidement devenir très nombreuses si le nombre de processus UNITY s'accroît. En effet, le programme produit contient alors toutes les possibilités de parallélisme entre deux instructions, toutes celles entre trois instructions, etc. Un programme composé de dix processus ayant chacun dix instructions exigerait la prise en compte de vingt-cinq milliards neuf cent trente-sept millions quatre cent vingt-quatre mille cinq cents instructions supplémentaires.

De plus, l'introduction d'opérateurs de produit fige l'architecture sur laquelle le programme s'exécutera en obligeant à préciser quelles instructions s'exécutent sur quels sites. Ceci va à l'encontre de la philosophie Unity qui veut que les considérations d'architecture soient reportées le plus tard possible dans le processus de conception.

Pour ces deux raisons, il semble intéressant de rechercher les moyens de raisonner sur un programme indépendamment de sa répartition précise et de n'introduire d'opérateurs de produit que tardivement et sans expliciter les instructions masquées par ces opérateurs. Ceci peut se faire en caractérisant, pour un programme bien réparti, les propriétés qui restent valides lorsque l'opérateur d'union est remplacé par l'opérateur de produit.

D'après le programme exemple (7.1), on sait déjà que certaines propriétés de la logique Unity de $F [] G$ ne sont pas valides dans $F [][] G$. Néanmoins, si la classe des propriétés maintenues est suffisante, ceci permettra de continuer à réaliser les preuves au niveau de $\langle [] i :: P_i \rangle$ puis de transformer certaines unions en produits en fonction de l'architecture choisie.

7.3.2 Théorème de raffinement

Les théorèmes suivants montrent la conservation de différents types de propriétés de sûreté et de vivacité lors d'un passage de $[]$ à $[][]$. On suppose le programme $\langle [] i :: P_i \rangle$ bien réparti. Le produit $\langle [][] i :: P_i \rangle$ est donc bien défini. P et Q sont deux prédicats d'état quelconques portant sur les variables des processus P_i. Dans ces conditions, on montre :

$$\frac{\texttt{invariant } P \texttt{ in } \langle [] i :: P_i \rangle}{\texttt{invariant } P \texttt{ in } \langle [][] i :: P_i \rangle} \tag{7.3}$$

$$\frac{P \mapsto Q \texttt{ in } \langle [] i :: P_i \rangle \ \wedge\ \langle \exists i :: Q.\mathit{Var} \subset P_i.\mathit{declare} \rangle}{P \mapsto Q \texttt{ in } \langle [][] i :: P_i \rangle} \tag{7.4}$$

$$\frac{P \mapsto Q \texttt{ in } \langle [] i :: P_i \rangle \ \wedge\ \texttt{stable } Q \texttt{ in } \langle [] i :: P_i \rangle}{P \mapsto Q \texttt{ in } \langle [][] i :: P_i \rangle} \tag{7.5}$$

Naturellement, dans le cas où les coïncidences d'exécution n'ont pas lieu, on retrouve pour le programme $\langle [][] i :: P_i \rangle$ les exécutions de $\langle [] i :: P_i \rangle$. On en déduit la relation suivante :

$$\langle [][] i :: P_i \rangle \sqsubseteq_{\mathcal{O}bs} \langle [] i :: P_i \rangle \tag{7.6}$$

qui montre que la transformation de l'union en produit n'introduit pas de propriétés indésirables dans un programme. En particulier, de (7.3) et (7.6), on déduit que les programmes $\langle [] i :: P_i \rangle$ et $\langle [][] i :: P_i \rangle$ ont exactement les mêmes invariants, c'est-à-dire le même ensemble d'états accessibles.

7.3.3 Exemple d'utilisation

On considère l'algorithme d'exclusion mutuelle présenté au chapitre 5. Nous avons modélisé le service d'exclusion mutuelle et son environnement par le programme :

$$SR \ = \ \langle \| \ s \ : \ 1 \leqslant s \leqslant N \ :: \ Mutex(s) \rangle$$

dont nous avons vu qu'il était bien réparti.

Nous avons démontré les propriétés suivantes, propriétés fondamentales de sûreté et de vivacité d'un algorithme d'exclusion mutuelle :

$$\begin{gathered}\texttt{invariant}\\ \langle \forall s, s' : 1 \leqslant s, s' \leqslant N :: s.mutex \wedge s'.mutex \ \Rightarrow \ (s = s') \rangle\\ \texttt{in}\ SR\end{gathered}$$

$$\forall s \ : 1 \leqslant s \leqslant N :: \ s.resquesting \ \mapsto \ s.mutex \ \texttt{in} \ SR$$

Ces propriétés prennent la forme d'un invariant global et d'un ensemble de propriétés de type *leads-to* dont les membres droits sont locaux à un processus $Mutex$. Elles sont donc conservées en vertu des propriétés (7.3) et (7.4). On en déduit que ces propriétés restent valides dans le programme :

$$SR' \ = \ \langle [\![\!]\!] \ s \ : \ 1 \leqslant s \leqslant N \ :: \ Mutex(s) \rangle$$

7.4 Preuves

7.4.1 Lemme fondamental

Soit $P = \langle \| i : 1 \leqslant i \leqslant N :: P_i \rangle$ un programme Unity bien réparti. On note $F = P_{N-1}$, $G = P_N$ et $R = \langle \| i : 1 \leqslant i \leqslant N-2 :: P_i \rangle$. On a donc $P = R \| F \| G$. La preuve des théorèmes (7.3), (7.4) et (7.5) repose alors sur le lemme important suivant qui établit un lien étroit entre chaque séquence d'états de $\mathcal{O}bs.(R \| (F [\![\!]\!] G))$ et une séquence correspondante dans $\mathcal{O}bs.(R \| F \| G)$.

Soit $s_F \| s_G$ `if` *fair* une instruction de $(R \| (F [\![\!]\!] G)).assign$ avec $s_F \in F.assign$ et $s_G \in G.assign$. En conservant la notation $\xrightarrow{s}$ pour décrire une transition due à l'instruction s au cours de laquelle les variables images sont susceptibles d'évoluer librement (cf § 5.3.3), on a :

$$\begin{aligned}&\langle \forall \sigma : \sigma \in \mathcal{O}bs.(R \| (F [\![\!]\!] G)) :: \langle \forall i : \sigma_i \xrightarrow{s_F \| s_G} \sigma_{i+1} ::\\ &\qquad \langle \exists \sigma' : \sigma' \in \mathcal{O}bs.(R \| F \| G) :: \langle \exists j :: \sigma'_j = \sigma_i \xrightarrow{s_F} \sigma'_{j+1} \xrightarrow{s_G} \sigma'_{j+2} = \sigma_{i+1} \rangle \rangle \rangle \rangle\end{aligned}$$

Ce résultat s'interprète de la manière suivante : lors d'une exécution de $R \| (F [\![\!]\!] G)$, les effets sur l'état d'un programme, lorsque $fair \equiv true$, d'une instruction composée introduite par l'opérateur de produit sont identiques à ceux produits par l'exécution successive des deux composantes de cette instruction dans une certaine exécution de $R \| F \| G$ (Fig. 7.1). Naturellement, lorsque $fair \equiv false$, l'instruction n'a pas d'effet sur l'état du programme.

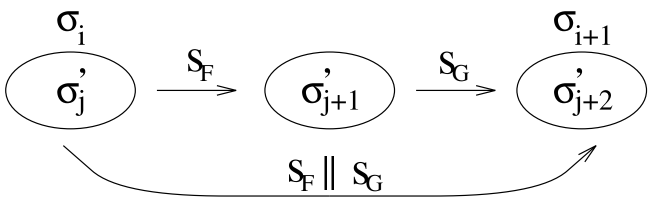

Fig. 7.1 – *Interprétation des instructions composées.*

Démonstration : pour démontrer ce lemme. il faut mettre en évidence l'existence de la sous-séquence $\sigma'_j \xrightarrow{s_F} \sigma'_{j+1} \xrightarrow{s_G} \sigma'_{j+2}$ dans une séquence de $\mathcal{O}bs.(R\|F\|G)$. Ceci se fait en trois étapes :

- La séparation des variables entre les programmes F et G permet de montrer que :
$$\langle \forall \sigma_i :: (s_F \parallel s_G).\sigma_i = s_F.s_G.\sigma_i = s_G.s_F.\sigma_i \rangle$$
Les effets de s_F et de s_G sur un état σ_i sont complètement indépendants et peuvent être considérés parallèlement ou dans n'importe quel ordre. On peut démontrer cette double égalité par exemple en calculant la *weakest precondition* de $s_F \| s_G$, $s_F; s_G$ et $s_G; s_F$.

- On suppose ensuite que $\langle \exists \sigma' : \sigma' \in \mathcal{O}bs.(R\|F\|G) :: \langle \exists j :: \sigma'_j = \sigma_i \rangle \rangle$, c'est-à-dire que l'état σ_i est accessible dans une séquence σ' de $\mathcal{O}bs.(R\|F\|G)$, et on montre que l'on peut choisir l'évolution de σ' pour les indices supérieurs à j de façon à respecter $\sigma'_j \xrightarrow{s_F} \sigma'_{j+1} \xrightarrow{s_G} \sigma'_{j+2} = \sigma_{i+1}$ [2]. Pour cela, on montre que le mécanisme d'observation est capable de faire évoluer les images parallèlement à s_F et à s_G de manière identique à leur évolution dans σ parallèlement à $s_F \parallel s_G$. Si on note ‘$\bar{v}$ le vecteur de *toutes* les variables images de $R\|F\|G$, on a $\sigma'_j.‘\bar{v} = \sigma_i.‘\bar{v}$ (puisque $\sigma'_j = \sigma_i$). On choisit σ' tel que $\sigma'_{j+1}.‘\bar{v} = \sigma'_j.‘\bar{v}$ (les images n'évoluent pas pendant l'exécution de l'instruction s_F) et $\sigma'_{j+2}.‘\bar{v} = \sigma_{i+1}.‘\bar{v}$ (les images évoluent parallèlement à s_G de manière analogue à leur évolution parallèlement à $s_F \parallel s_G$ dans σ). Cette évolution des variables images est conforme à la définition de la relation d'observation. Il existe donc une telle séquence σ' dans $\mathcal{O}bs.(R\|F\|G)$.

- Le raisonnement précédent montre en fait la validité du lemme pour le cas de la *première* occurrence d'une instruction supplémentaire (c'est-à-dire une instruction qui n'était pas dans $(R\|F\|G).assign$). On en déduit que dans ce cas, l'état σ_{i+1} est lui aussi accessible dans une séquence σ' et que le même raisonnement s'applique pour la seconde occurrence d'une instruction supplémentaire. On montre donc par récurrence la validité du lemme pour toutes les occurrences d'instructions supplémentaires.

Ce procédé de construction d'une séquence de $\mathcal{O}bs.(R\|F\|G)$ montre que l'on peut choisir l'exécution σ' telle que les images des relations d'observation

2. Partant d'un état donné, l'exécution d'une instruction donnée peut mener vers différents états selon l'évolution des variables images.

restent constantes pendant l'exécution de l'instruction s_F. On a donc en fait une version plus forte du lemme qui est :

$$\langle \forall \sigma : \sigma \in \mathcal{O}bs.(R \| (F \| \| G)) :: \langle \forall i : \sigma_i \xrightarrow{s_F \| s_G} \sigma_{i+1} :: \\ \langle \exists \sigma' : \sigma' \in \mathcal{O}bs.(R \| F \| G) :: \langle \exists j :: \sigma'_j = \sigma_i \wedge s_F.\sigma'_j = \sigma'_{j+1} \xrightarrow{s_G} \sigma'_{j+2} = \sigma_{i+1} \rangle\rangle\rangle\rangle \tag{7.7}$$

C'est cette version du lemme qui est utilisée dans la preuve qui suit.

7.4.2 Démonstration des théorèmes (7.3), (7.4) et (7.5)

À partir du moment où le lemme (7.7) est établi, on montre aisément les trois théorèmes. En effet, le lemme montre que l'exécution simultanée de deux instructions issues de deux processus différentes n'a pas d'autre effet sur les séquences que de court-circuiter un état. Aucun nouvel état ne peut donc être atteint et il y a bien conservation des invariants du programme. On en déduit :

$$\frac{\texttt{invariant } P \texttt{ in } \langle \| i : 1 \leqslant i \leqslant N :: P_i \rangle}{\texttt{invariant } P \texttt{ in } \langle \| i : 1 \leqslant i \leqslant N-2 :: P_i \rangle \| (P_{N-1} \| \| P_N)}$$

et par récurrence sur le nombre d'occurrences de l'opérateur d'union, ceci démontre la relation (7.3).

En ce qui concerne la première relation portant sur les propriétés de type *leads-to*, on montre, à partir du lemme (7.7) :

$$\frac{P \mapsto Q \texttt{ in } \langle \| i : 1 \leqslant i \leqslant N :: P_i \rangle \wedge Q.Var \subset \langle \bigcup i : 1 \leqslant i \leqslant N-2 :: P_i.declare \rangle}{P \mapsto Q \texttt{ in } \langle \| i : 1 \leqslant i \leqslant N-2 :: P_i \rangle \| (P_{N-1} \| \| P_N)} \tag{7.8}$$

En effet, l'hypothèse $P \mapsto Q$ **in** $R \| F \| G$ permet d'affirmer que dans toute séquence σ' de $\mathcal{O}bs.(R \| F \| G)$, un état satisfaisant P est nécessairement suivi par un état satisfaisant Q. La seule manière, pour une séquence σ de $\mathcal{O}bs.(R \| (F \| \| G))$ de ne pas satisfaire la propriété consiste à court-circuiter le dernier état satisfaisant Q (s'ils sont en nombre fini) ou tous les états satisfaisant Q à partir d'un certain rang (s'ils sont en nombre infini). Dans les deux cas, l'exécution de $s_F \| s_G$ doit contourner un état satisfaisant Q entre deux états ne satisfaisant pas Q (Fig. 7.2). De plus, on a vu qu'on pouvait choisir σ' de telle

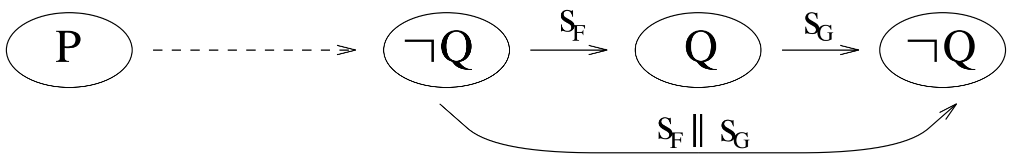

FIG. 7.2 – *Cas de falsification d'une propriété* leads-to.

sorte que les variables images n'évoluent pas pendant l'exécution de l'instruction s_F (lemme (7.7)). Le prédicat Q ne portant que sur des variables de R, la transition $\neg Q \xrightarrow{s_F} Q$ est donc impossible, ce qui montre la propriété (7.8). Si le prédicat Q les local à un processus, on peut appliquer cette propriété $N-1$ fois en laissant toujours la partie du programme qui contient Q à droite de l'union considérée. On en déduit la relation (7.4).

Enfin, la seconde relation portant sur les propriétés de type *leads-to* se démontre directement. En effet, l'hypothèse `stable` Q `in` $R \| F \| G$ interdit la transition $Q \xrightarrow{s_G} \neg Q$. On en déduit la relation (7.5).

7.4.3 Démonstration du théorème (7.6)

Étant donné que nous avons pris soin de ne pas imposer d'équité sur les instructions introduites par le produit, la propriété de raffinement de la transformation du produit vers l'union se montre facilement.

Soit $\sigma' \in \mathcal{O}bs.(\langle \| i :: P_i \rangle)$. La contrainte d'équité faible force l'exécution, une infinité de fois, de l'instruction $fair := \neg fair$. Comme aucune autre instruction ne peut modifier la variable *fair*, on en déduit que l'exécution σ' comporte une infinité d'états dans lesquels $fair \equiv false$. On construit la séquence $\sigma \in \mathcal{O}bs.(\langle [\![\!]\!] i :: P_i \rangle)$ en exécutant les mêmes instructions que dans σ', et en insérant une infinité de fois les instructions supplémentaires à partir d'un état où $fair \equiv false$. Les exécutions des instructions supplémentaires n'ont donc aucun effet sur l'état du programme et la contrainte d'équité est respectée. Par conséquent, les exécutions σ et σ' sont équivalentes au bégaiement près. Ceci montre :

$$\mathcal{O}bs.(\langle \| i :: P_i \rangle) \subset \Gamma.\mathcal{O}bs.(\langle [\![\!]\!] i :: P_i \rangle)$$

et par conséquent, le théorème (7.6).

Il faut noter que, telle que la séquence σ est construite, elle bégaie une infinité de fois à partir d'états où $fair \equiv false$. Une séquence de $\mathcal{O}bs.(\langle \| i :: P_i \rangle)$ qui n'exécute l'instruction *skip* que dans des états où $fair \equiv true$ risque de ne pas présenter ce bégaiement. C'est pourquoi on n'a pas, en général, l'inclusion des sémantiques opérationnelles :

$$\mathcal{O}bs.(\langle \| i :: P_i \rangle) \not\subset \mathcal{O}bs.(\langle [\![\!]\!] i :: P_i \rangle)$$

D'autre part, il faut noter que la propriété (7.6) ne serait pas valide si les instructions introduites par le produit étaient soumises à l'équité faible. C'est le cas, par exemple, pour les processus suivants :

Program F **Declare** x : Int; **Assign** x := 1 $[\!]$ x := 0 **End** F	**Program** G **Declare** y : Int; **Assign** y := 1 $[\!]$ y := 0 **End** G

Si les instructions introduites étaient soumises à l'équité faible, le programme $F [\![\!]\!] G$ comporterait l'instruction $x, y := 1, 1$. Par conséquent, il vérifierait la propriété $true \;\mapsto\; x + y = 2$ qui n'est pas vérifiée dans $F \| G$.

La définition du produit et les théorèmes de raffinement qui lui sont associés ont fait l'objet du rapport de recherche [Cha96] et ont été présentés dans [Cha97].

Chapitre 8

L'assistant de preuve DADA

Dans ce chapitre, nous présentons le prototype d'assistant de preuve que nous avons développé : DADA (*Distributed Algorithms Design Assistant*). Cet assistant se présente sous la forme d'un éditeur syntaxique du langage et de la logique UNITY. Il fait appel à un démonstrateur automatique de théorèmes et à une procédure de décision arithmétique pour vérifier dynamiquement et automatiquement la consistance entre la spécification et le programme décrits.

Nous commençons par décrire l'environnement en terme de son interface et de ses fonctionnalités. Trois exemples de programmes validés à l'aide de l'environnement sont présentés.

Nous exposons ensuite la manière dont l'environnement utilise un démonstrateur automatique de théorèmes. Le démonstrateur utilisé est OTTER, un démonstrateur fonctionnant par réfutation / résolution. Nous présentons brièvement ce démonstrateur, puis la technique retenue pour l'utiliser dans le cadre de la preuve de programmes UNITY. Cette partie se termine par la description de quelques heuristiques permettant de simplifier des obligations de preuve trop imposantes.

Afin de mieux gérer les aspect arithmétiques des preuves, notre environnement utilise également une procédure de décision pour l'arithmétique de Presburger appelée OMEGA. Nous rappelons rapidement ce que sont les formules de l'arithmétique de Presburger, puis nous détaillons la façon dont la procédure est utilisée par l'environnement.

8.1 L'environnement de développement

8.1.1 Description

L'environnement réalisé se présente sous la forme d'un éditeur syntaxique (du langage UNITY et de sa logique) sous X WINDOW SYSTEM. Cet éditeur a été engendré par le GRAMMATECH SYNTHESIZER GENERATOR et possède toutes les fonctionnalités des éditeurs engendrés par cet outil.

Cet environnement gère un texte composé de deux parties :

- l'expression d'un algorithme dans le langage UNITY ;

– une spécification des propriétés de sûreté et de vivacité de cet algorithme exprimée dans la logique UNITY.

La figure 8.1 montre un exemple de l'interface utilisateur offerte par l'environnement. Chaque instruction du programme est suivie par la liste des références

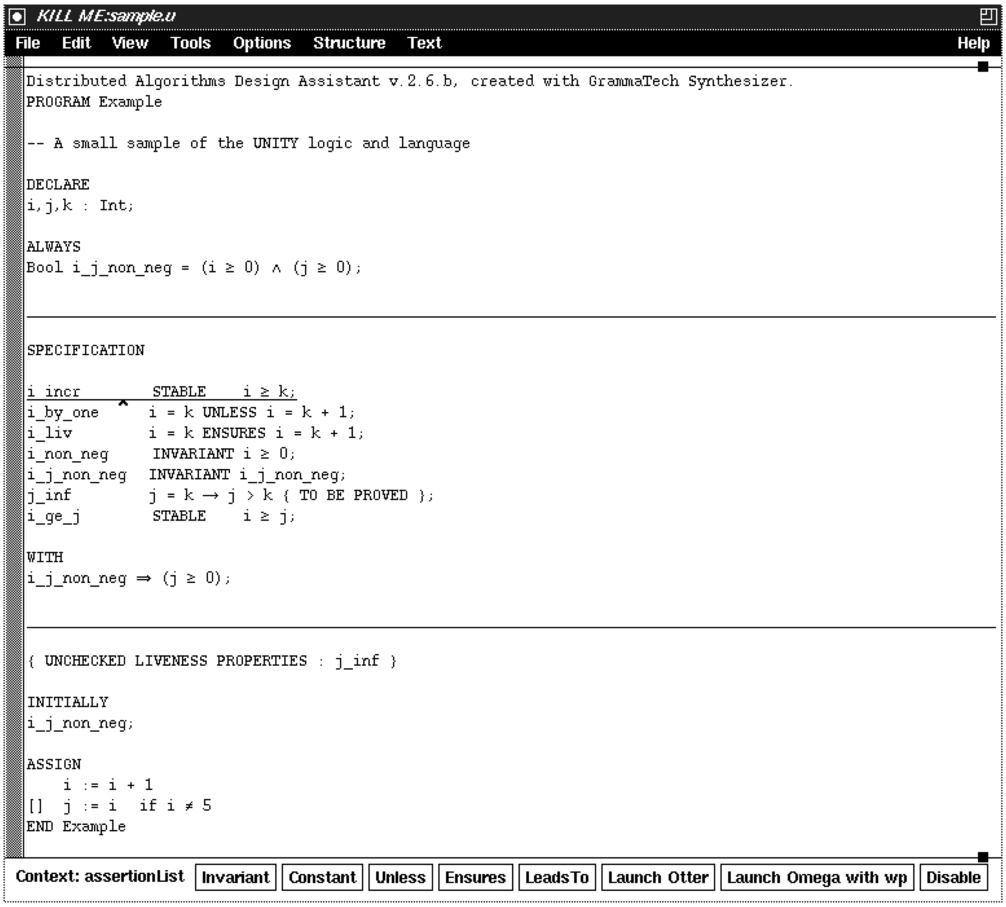

FIG. 8.1 – *Exemple de l'interface utilisateur.*

aux propriétés à vérifier. Chaque référence disparaît si la propriété correspondante parvient à être démontrée.

L'objectif est de garantir automatiquement la cohérence entre le programme et sa spécification (c'est-à-dire de vérifier que le programme respecte bien toutes les contraintes spécifiées). Pour cela, nous utilisons deux outils automatiques : le démonstrateur de théorèmes OTTER et le calculateur de formules *Presburger* OMEGA.

Actuellement, seul un sous-ensemble des caractéristiques de UNITY sont prises en compte par l'environnement. En ce qui concerne le langage, seules les variables scalaires entières et booléennes sont acceptées. L'absence de tableaux rend inutile la quantification des instructions. Cette limitation est une option temporaire facilitant une première réalisation. En ce qui concerne la logique, l'opérateur *leads-to* n'est pas traité. Les autres opérateurs sont implantés dans la sémantique de K.M. Chandy et J. Misra (sans axiome de substitution).

8.1.2 Utilisation

Des exemples d'utilisation de l'environnement ont été étudiés dans différents domaines, en particulier ceux des systèmes répartis, des circuits logiques ou encore de la robotique. Nous présentons ici trois exemples consacrés à la synchronisation dans les systèmes et ayant souvent fait l'objet de preuve dans le contexte d'autres projets.

On retrouve dans ces exemples les différentes sections d'un programme UNITY (*declare*, *always*, *initially* et *assign*) ainsi qu'une section particulière (*with*) qui fournit une interface directe avec les démonstrateurs. Elle permet d'obtenir directement d'un démonstrateur la preuve de prédicats (non temporels) portant sur les variables du programme. Ces preuves sont indépendantes de celles associées au contrôle des spécifications.

Algorithme d'exclusion mutuelle de Peterson

Ce premier exemple décrit l'algorithme de Peterson qui assure l'exclusion mutuelle entre deux processus. Les cinq premières instructions modélisent le comportement du premier processus, les cinq suivantes celui du second. Les deux dernières affectations représentent le non déterminisme des processus susceptibles de ne jamais entrer en zone critique. Les variables m et n sont les compteurs ordinaux des deux processus. En particulier, l'égalité $m = critical$ signifie que le premier processus est en zone critique.

Onze propriétés de sûreté et de vivacité sont exprimées dans la spécification de ce programme. La première :

$$\texttt{invariant}\ I\_ONE$$

permet d'établir la propriété de sûreté fondamentale de ce type d'algorithme : l'exclusion mutuelle, exprimée par le fait que $EXCLUSION$ est une propriété *toujours vraie* du programme. On notera que la preuve de l'implication :

$$I\_ONE \Rightarrow EXCLUSION$$

est réalisée par l'un des démonstrateurs en utilisant la section *with*.

Il est important de noter que les invariants $three$, $five$, six et $seven$ ne sont pas validés par l'environnement. Ces propriétés correspondent en fait à une démonstration (manuelle) erronée dans la présentation originale de cet exemple (cf. bibliographie).

Introduit dans l'environnement, le programme se présente sous la forme suivante (l'expression LaTeX du programme est produite par l'environnement lui-même) :

Distributed Algorithms Design Assistant v.2.6.b, created with GrammaTech Synthesizer.

Program `Mutex`
```
-- Peterson's mutual exclusion algorithm from UV user manual (M. Kaltenbach)
-- written and specified by J. Misra
-- Properties three, five, six and seven are wrong.
```
Declare
```
m, n : Int;
u, v, p : Bool;
hu, hv : Bool;
```
Always
```
Int noncritical  = 0;
Int requesting   = 1;
Int trying       = 2;
Int critical     = 3;
Int exiting      = 4;
Bool EXCLUSION   = ¬((m = critical) ∧ (n = critical));
Bool I_EIGHT     = (u ⇔ ((m ⩾ requesting) ∧ (m ⩽ critical))) ∧ ((m = critical) ⇒ ¬p);
Bool I_NINE      = (v ⇔ ((n ⩾ requesting) ∧ (n ⩽ critical))) ∧ ((n = critical) ⇒ p);
Bool I_ONE       = EXCLUSION ∧ I_EIGHT ∧ I_NINE;
```

Specification
```
one    invariant I_ONE;
two    invariant u ⇔ ((m ⩾ requesting) ∧ (m ⩽ critical));
three  invariant ((m = critical) ∨ (m = exiting)) ⇒ ¬p { TO BE PROVED };
four   invariant v ⇔ ((n ⩾ requesting) ∧ (n ⩽ critical));
five   invariant ((n = critical) ∨ (n = exiting)) ⇒ p { TO BE PROVED };
six    invariant (u ⇔ ((m ⩾ requesting) ∧ (m ⩽ critical))) ∧ (((m = critical) ∨
       (m = exiting)) ⇒ ¬p) { TO BE PROVED };
seven  invariant (v ⇔ ((n ⩾ requesting) ∧ (n ⩽ critical))) ∧ (((n = critical) ∨
       (n = exiting)) ⇒ p) { TO BE PROVED };
eight  invariant (u ⇔ ((m ⩾ requesting) ∧ (m ⩽ critical))) ∧ ((m = critical) ⇒ ¬p);
nine   invariant (v ⇔ ((n ⩾ requesting) ∧ (n ⩽ critical))) ∧ ((n = critical) ⇒ p);
ten    m = trying unless m = critical;
eleven m = requesting ensures (p ⇔ v) ∧ (m = trying);
```
With
```
I_ONE ⇒ EXCLUSION ;
```

Initially
```
¬u;
¬v;
m = noncritical;
n = noncritical;
```
Assign
```
  u, m:= true, requesting if hu ∧ (m = noncritical)
‖
  p, m:= v, trying if m = requesting { five seven }
‖
  m:= critical if ¬p ∧ (m = trying)
‖
  u, m:= false, exiting if m = critical
‖
  p, m:= true, noncritical if m = exiting
‖
  v, n:= true, requesting if hv ∧ (n = noncritical)
‖
  p, n:= ¬u, trying if n = requesting { three six }
‖
  n:= critical if p ∧ (n = trying)
‖
  v, n:= false, exiting if n = critical
‖
  p, n:= false, noncritical if n = exiting
‖
  hu := ¬hu
‖
  hv := ¬hv
```
End `Mutex`

Exclusion mutuelle avec tolérance aux fautes

Dans cet exemple, nous étudions le problème de l'exclusion mutuelle dans un contexte réparti en mettant l'accent sur l'aspect tolérance aux fautes. Une solution classique de ce problème consiste à faire circuler un message unique entre les sites. Cet algorithme dit du "jeton" assure la propriété d'exclusion mutuelle dans la mesure où seul le site possédant le message-jeton possède le droit (privilège) d'entrer en exclusion. A cette propriété de sûreté s'ajoute une propriété de vivacité devant garantir à tout site demandeur qu'il entrera finalement en exclusion. Cette équité est souvent simplement obtenue en faisant circuler le jeton sur un anneau logique passant par tous les sites.

En UNITY, il nous faut donc décrire les propriétés de sûreté et de vivacité précédentes. Une solution est présentée dans le programme ci-dessous. On suppose une architecture comportant trois sites. Chaque site est représenté par une suite d'instructions similaires aux noms des variables près. En effet, tous les sites ont le même comportement générique.

Un site i quelconque peut demander à entrer en exclusion en affectant la valeur vraie à la variable s_i_req. Ce site n'entrera en exclusion que lorsque la variable s_i_excl deviendra vraie. Ceci est conditionné à la présence du jeton sur le site i, mémorisée par la variable s_i_tok. Après quoi, la sortie d'exclusion se traduira par une remise à faux des variables s_i_req et s_i_excl. Ce comportement peut être répété à l'infini.

La circulation du jeton est décrite, quant à elle, par la mise à jour d'un couple quelconque de variables (s_i_tok, s_k_tok). L'unicité du jeton repose sur la preuve que dans tout état du programme, une variable s_i_tok et une seule est vraie.

En outre, on suppose possible la perte du jeton par l'un quelconque des sites. Un site privilégié (ici, le site 1) peut alors débuter une procédure de régénération du jeton. Il propage alors une enquête sur l'anneau à l'aide des variables s_i_invst. Si le résultat de cette enquête lui permet de confirmer que le jeton n'est effectivement présent sur aucun site, il procède à la création d'un nouveau jeton.

Distributed Algorithms Design Assistant v.2.6.b, created with GrammaTech Synthesizer.

```
Program jeton
  -- Mutual exclusion by token (with loss tolerance)
Declare
  s1_req, s2_req, s3_req : Bool;
  s1_excl, s2_excl, s3_excl : Bool;
  s1_tok, s2_tok, s3_tok : Bool;
  s1_invst, s2_invst, s3_invst : Bool;
Always
  Bool TOK_ON_S1    = s1_tok ∧ ¬s2_tok ∧ ¬s3_tok;
  Bool TOK_ON_S2    = s2_tok ∧ ¬s3_tok ∧ ¬s1_tok;
  Bool TOK_ON_S3    = s3_tok ∧ ¬s1_tok ∧ ¬s2_tok;
  Bool TOK_UNIQUE   = TOK_ON_S1 ∨ TOK_ON_S2 ∨ TOK_ON_S3;
  Bool TOK_EXISTS   = s1_tok ∨ s2_tok ∨ s3_tok;
  Bool CRITICAL_S1  = s1_req ∧ s1_excl ∧ ¬s2_excl ∧ ¬s3_excl;
  Bool CRITICAL_S2  = s2_req ∧ s2_excl ∧ ¬s1_excl ∧ ¬s3_excl;
  Bool CRITICAL_S3  = s3_req ∧ s3_excl ∧ ¬s1_excl ∧ ¬s2_excl;
  Bool EXCL_FREE    = ¬(s1_excl ∨ s2_excl ∨ s3_excl);
  Bool EXCLUSION    = CRITICAL_S1 ∨ CRITICAL_S2 ∨ CRITICAL_S3;
  Bool WHEN_INVST   = (s1_invst ⇒ (¬s1_tok ∧ ¬s1_excl)) ∧ (s2_invst ⇒ (¬s2_tok ∧ ¬s2_excl)) ∧
                      (s3_invst ⇒ (¬s3_tok ∧ ¬s3_excl));
  Bool WHEN_TOK     = (s1_tok ⇒ ¬s1_invst) ∧ (s2_tok ⇒ ¬s2_invst) ∧ (s3_tok ⇒ ¬s3_invst);
  Bool SINGLE       = (TOK_UNIQUE ∨ ¬TOK_EXISTS) ∧ WHEN_TOK;
  Bool WHEN_EXCL    = (s1_excl ⇒ (s1_req ∧ ¬s2_tok ∧ ¬s3_tok)) ∧ (s2_excl ⇒ (s2_req ∧ ¬s1_tok ∧
                      ¬s3_tok)) ∧ (s3_excl ⇒ (s3_req ∧ ¬s1_tok ∧ ¬s2_tok)) ∧ SINGLE ∧ WHEN_INVST;
```

```
Specification
  WHEN_EXCL   invariant WHEN_EXCL;
  WHEN_INVST  invariant WHEN_INVST;
  WHEN_TOK    invariant WHEN_TOK;
  SINGLE      invariant SINGLE;
  MUTEX       invariant (EXCLUSION ∨ EXCL_FREE) ∧ WHEN_EXCL;
With
  TOK_ON_S1 ⇒ ¬(TOK_ON_S2 ∨ TOK_ON_S3) ;
  TOK_ON_S1 ⇔ (TOK_EXISTS ∧ ¬s2_tok ∧ ¬s3_tok) ;
  (s1_tok ∧ ¬s2_tok ∧ ¬s3_tok ∧ ¬s1_req ∧ ¬s2_req ∧ ¬s3_req ∧ ¬s1_excl ∧ ¬s2_excl ∧ ¬s3_excl ∧
    ¬s1_invst ∧ ¬s2_invst ∧ ¬s3_invst) ⇒ WHEN_EXCL ;
```

```
Initially
  s1_tok ∧ ¬s2_tok ∧ ¬s3_tok;
  ¬s1_req ∧ ¬s2_req ∧ ¬s3_req;
  ¬s1_excl ∧ ¬s2_excl ∧ ¬s3_excl;
  ¬s1_invst ∧ ¬s2_invst ∧ ¬s3_invst;
Assign
  s1_req := true
‖
  s1_excl := true if s1_tok ∧ s1_req
‖
  s1_req, s1_excl := false, false if s1_excl
‖
  s1_tok := false
‖
  s2_req := true
‖
  s2_excl := true if s2_tok ∧ s2_req
‖
  s2_req, s2_excl := false, false if s2_excl
‖
  s2_tok := false
‖
  s3_req := true
‖
  s3_excl := true if s3_tok ∧ s3_req
‖
  s3_req, s3_excl := false, false if s3_excl
‖
  s3_tok := false
‖
```

```
  s1_tok, s2_tok:=false, true if s1_tok ∧ ¬s1_req
 ||
  s2_invst:=false if s1_tok ∧ ¬s1_req
[]
  s2_tok, s3_tok:=false, true if s2_tok ∧ ¬s2_req
 ||
  s3_invst:=false if s2_tok ∧ ¬s2_req
[]
  s3_tok, s1_tok:=false, true if s3_tok ∧ ¬s3_req
 ||
  s1_invst:=false if s3_tok ∧ ¬s3_req
[]
  s1_invst:=true if ¬s1_tok ∧ ¬s1_excl
[]
  s2_invst:=s1_invst if ¬s2_tok ∧ ¬s2_excl ∧ s1_invst
[]
  s3_invst:=s2_invst if ¬s3_tok ∧ ¬s3_excl ∧ s2_invst
[]
  s1_tok, s1_invst:=true, false if s1_invst ∧ s2_invst ∧ s3_invst

End jeton
```

Le problème du parking

Le problème du parking modélise l'allocation de ressources critiques à des processus concurrents. La capacité C du parking représente le nombre de ressources disponibles. L'arrivée d'une voiture traduit une demande de ressource. L'entrée d'une voiture dans le parking correspond à l'allocation effective d'une ressource et la sortie à la libération de cette ressource. La spécification du problème repose sur des compteurs d'événements représentant respectivement le nombre d'arrivées A, d'entrées E et de sorties D.

La propriété caractéristique invariante consiste à garantir que le nombre de voitures dans le parking reste inférieur ou égal à la capacité du parking, soit en terme d'événements :

$$E - D + F = C$$

où F désigne le nombre de places disponibles et E, D et F sont positifs ou nuls.

Une mise en œuvre répartie peut être modélisée par un parking à plusieurs points d'accès. Chaque point d'accès possède alors ses propres compteurs d'événements et de places disponibles a_i, e_i, d_i et f_i avec les relations de représentation :

$$A = \sum_i a_i, \quad E = \sum_i e_i, \quad D = \sum_i d_i, \quad F = \sum_i f_i$$

Dans le programme suivant, on suppose décrit un parking à trois points d'accès. Les instructions UNITY traduisent l'évolution des compteurs d'événements et l'échange de places disponibles (f_i) entre les points d'accès selon une structure d'anneau.

```
Distributed Algorithms Design Assistant v.2.6.b, created with GrammaTech Synthesizer.
Program Parking
Declare
   C : Int;
   a1, a2, a3, e1, e2, e3, f1, f2, f3, d1, d2, d3 : Int;
   ka, ke, kd : Int;
Always
   Int D            = d1 + d2 + d3;
   Int E            = e1 + e2 + e3;
   Int A            = a1 + a2 + a3;
   Int F            = f1 + f2 + f3;
   Bool INV_GL_IMPL = (E − D + F = C) ∧ (F ⩾ 0) ∧ (f1 ⩾ 0) ∧ (f2 ⩾ 0) ∧ (f3 ⩾ 0);
--------------------------------------------------------------------------------
Specification
   Inv_gl_Impl  invariant INV_GL_IMPL;
   f123_ge_0    invariant (f1 ⩾ 0) ∧ (f2 ⩾ 0) ∧ (f3 ⩾ 0);
   a123_ge_0    invariant (a1 ⩾ 0) ∧ (a2 ⩾ 0) ∧ (a3 ⩾ 0);
   e123_ge_0    invariant (e1 ⩾ 0) ∧ (e2 ⩾ 0) ∧ (e3 ⩾ 0);
   d123_ge_0    invariant (d1 ⩾ 0) ∧ (d2 ⩾ 0) ∧ (d3 ⩾ 0);
   a1_incr      a1 = ka unless a1 > ka;
   e1_incr      e1 = ke unless e1 > ke;
   s1_incr      d1 = kd unless d1 > kd;
With
   INV_GL_IMPL ⇒ ((E − D) ⩽ C) ;
--------------------------------------------------------------------------------
Initially
   (a1 = 0) ∧ (a2 = 0) ∧ (a3 = 0) ∧ (e1 = 0) ∧ (e2 = 0) ∧ (e3 = 0) ∧ (d1 = 0) ∧ (d2 = 0) ∧ (d3 = 0);
   (f1 = 2) ∧ (f2 = 2) ∧ (f3 = 1) ∧ (C = 5);
Assign
   a1 := a1 + 1
[]
   e1, f1 := e1 + 1, f1 − 1 if (e1 < a1) ∧ (f1 > 0)
[]
   d1, f1 := d1 + 1, f1 + 1 if d1 < e1
[]
   a2 := a2 + 1
[]
   e2, f2 := e2 + 1, f2 − 1 if (e2 < a2) ∧ (f2 > 0)
[]
   d2, f2 := d2 + 1, f2 + 1 if d2 < e2
[]
   a3 := a3 + 1
[]
   e3, f3 := e3 + 1, f3 − 1 if (e3 < a3) ∧ (f3 > 0)
[]
   d3, f3 := d3 + 1, f3 + 1 if d3 < e3
[]
   f1, f2 := f1 − 1, f2 + 1 if f1 > 0
[]
   f2, f3 := f2 − 1, f3 + 1 if f2 > 0
[]
   f3, f1 := f3 − 1, f1 + 1 if f3 > 0
End Parking
```

8.1.3 Performances

Le tableau suivant présente quelques statistiques recueillies lors de l'utilisation de l'environnement de preuve sur les exemples précédents. Pour chaque exemple sont indiqués le démonstrateur utilisé, le nombre d'instructions dans le programme, le nombre de propriétés dans la spécification, le nombre de contraintes logiques engendrées et soumises au démonstrateur et la durée totale de la preuve de l'ensemble de ces contraintes élémentaires.

D'une manière générale, les performances en temps du calculateur OMEGA sont meilleures que celles de OTTER. Malheureusement, les calculs effectués par OMEGA sont extrêmement gourmands en mémoire lorsque la taille des formules devient trop grande. C'est pourquoi certaines propriétés de l'algorithme du jeton n'ont pu être prouvées avec ce démonstrateur et OTTER a donc été utilisé.

Ces mesures de performances ont été obtenues sur une station de travail ***Sun* SPARCstation 5** équipée de 64 MB de mémoire.

Algorithme	Prouveur	Nombre d' instructions	Nombre de propriétés	Nombre de preuves	Temps de preuve
Peterson	OMEGA	13	12	153	1mn 10s
Jeton	OTTER	20	7	726	13mn
Parking	OMEGA	13	9	102	20s

8.2 Exploitation d'un démonstrateur automatique de théorèmes

Pour vérifier la cohérence d'un programme par rapport à sa spécification, nous sommes amenés à exploiter un démonstrateur automatique de théorèmes, en l'occurrence OTTER. La composante éditeur de l'environnement engendre en effet l'ensemble des prédicats à valider pour s'assurer que le programme vérifie sa spécification. Cette traduction, effectuée par le biais de grammaires attribuées, est fondée sur le calcul de *weakest precondition*.

8.2.1 Présentation du démonstrateur Otter

Les techniques utilisées en démonstration automatique sont diverses et ont donné naissance à des types variés d'outils. Les systèmes de preuve par réfutation fondées sur la résolution et les systèmes de réécriture constituent deux branches parmi les plus importantes. Le démonstrateur actuellement utilisé est OTTER (*Organized Techniques for Theorem-proving and Effective Research*) qui est un système de preuve de prédicats du premier ordre par réfutation / résolution. Ses atouts sont sa simplicité d'utilisation (ce qui facilite l'interfaçage avec l'éditeur), sa rapidité d'exécution et son aptitude à travailler de manière *entièrement* automatique. En effet, les formules que nous avons à traiter sont relativement simples mais très longues et très nombreuses. Il était donc nécessaire de disposer d'un outil automatique et rapide.

L'exemple suivant, volontairement choisi hors du domaine pour des raisons de clarté et de concision, se traduit directement dans la syntaxe de OTTER par trois axiomes correspondant aux hypothèses, une règle de réécriture[1] et la négation de la conclusion qui devient le but à réfuter.

1. Nous utilisons ici la possibilité offerte par OTTER d'appliquer des règles de réécriture aussi bien à des atomes qu'à des termes.

Exemple : *Mon père est le grand-père de mon fils*

Hypothèses : $\forall x \ \forall y \ \forall z \quad pere(x,y) \ \wedge \ pere(y,z) \ \Rightarrow \ grand\_pere(x,z)$

$$fils(Paul, Jacques)$$
$$pere(Paul, Pierre)$$

Règle orientée : $fils(x,y) \ \rightarrow \ pere(y,x)$

Conclusion : $grand\_pere(Jacques, Pierre)$?

A partir des hypothèses et de la conclusion précédentes, Otter produit la preuve suivante :

```
----> UNIT CONFLICT at   0.06 sec ----> 9 [binary,8,4] .

Level of proof is 2, length is 3.

---------------- PROOF ----------------

1 [] (fils(x,y) = pere(y,x)).
2 [] -pere(x,y) | -pere(y,z) | grand_pere(x,z).
3 [demod,1] pere(Jacques,Paul).
4 [] pere(Paul,Pierre).
6 [] -grand_pere(Jacques,Pierre).
7 [binary,6,2] -pere(Jacques,x) | -pere(x,Pierre).
8 [binary,7,3] -pere(Paul,Pierre).
9 [binary,8,4] .

------------ end of proof -------------
```

8.2.2 Principe de validation des spécifications

Un prédicat P est une pré-condition valable à l'établissement de Q s'il constitue une contrainte *plus forte* que la wp, c'est-à-dire s'il l'implique. On a donc :

$$\{P\} \ s \ \{Q\} \quad ssi \quad P \Rightarrow wp\,(s, Q)$$

La fonction $P \rightarrow wp\,(s, P)$, de l'ensemble des prédicats dans lui-même, constitue donc une formalisation de la sémantique de l'instruction s.

L'avantage de cette approche est lié à la calculabilité de la wp. En effet, la fonction précédente est parfaitement définie et facilement programmable pour toutes les affectations gardées du langage Unity. Il est ainsi possible de calculer automatiquement $wp\,(s, Q)$ pour toute instruction s, et donc de ramener un problème de spécification $\{P\} \ s \ \{Q\}$ à celui de la validité de la formule $P \ \Rightarrow \ wp\,(s, Q)$.

Cas des instructions UNITY

Considéré comme langage de programmation, Unity n'utilise qu'un unique type d'instruction : l'affectation multiple. La seule wp à calculer est donc celle qui définit la sémantique de l'affectation. Une formule Q sera valide après l'exécution d'une affectation si la formule dont elle est issue par l'affectation était valide avant l'exécution de l'instruction. La wp de l'affectation se calcule donc facilement par substitution :

$$wp\,(v_1, \cdots, v_n \ := e_1, \cdots, e_n \ , \ Q) \quad \equiv \quad Q|^{v_1,\cdots,v_n}_{e_1,\cdots,e_n}$$

Ce principe de calcul s'étend au cas où les affectations sont gardées. Une affectation n fois gardée peut être vue comme n affectations différentes selon la valeur des gardes. La wp doit donc être suffisante quelle que soit l'affectation réellement exécutée. Par contre, la validité d'une garde peut être admise dans la considération d'une affectation particulière et venir renforcer la connaissance liée à la pré-condition. De plus, à ces n instructions vient s'ajouter l'instruction *skip* (de wp identité) pour le cas où aucune garde n'est valide (aucune des n affectations n'est exécutée).

Pour une instruction s de la forme :

$$\begin{array}{rcll} v_1, v_2, \cdots, v_n & := & e^1_1, e^1_2, \cdots, e^1_n & \texttt{if} \;\; g_1 \\ & \sim & e^2_1, e^2_2, \cdots, e^2_n & \texttt{if} \;\; g_2 \\ & \cdots & \cdots & \texttt{if} \;\; \cdots \\ & \sim & e^p_1, e^p_2, \cdots, e^p_n & \texttt{if} \;\; g_p \end{array}$$

on a donc :

$$\{P\}\ s\ \{Q\}\ \ ssi\ \ \bigwedge_{i=1}^{p} (P \wedge g_i \Rightarrow wp\,(\alpha_i, Q))\ \bigwedge\ (P \wedge \neg g_1 \wedge \cdots \wedge \neg g_p \Rightarrow Q)$$

où α_i dénote l'affectation $v_1, v_2, \cdots, v_n \ := \ e^i_1, e^i_2, \cdots, e^i_n$ et où chaque $wp\,(\alpha_i, Q)$ est calculée par substitution sur Q.

Cas de la logique UNITY

Dans le cas de la formule temporelle X `unless` Y, les prédicats P et Q précédents valent respectivement $X \wedge \neg Y$ et $X \vee Y$. Comme $X \wedge \neg Y \Rightarrow X \vee Y$ pour tout X et tout Y, le cas d'une affectation pour laquelle aucune garde n'est vraie n'est pas à considérer. En effet, la contrainte est nécessairement vérifiée par l'instruction *skip*.

Le résultat reste valable pour les contraintes *stable*, *constant* et *invariant* qui ne sont que des cas particuliers de *unless*.

8.2.3 Optimisation des preuves

Un certain nombre de preuves peuvent être optimisées à partir de la constatation suivante : lorsqu'un prédicat P ne référence pas les variables d'une instruction s, on a $wp\,(s, P) = P$. Ce cas de figure, relativement fréquent en pratique, peut être effectivement exploité.

Cas des invariants conjonctifs : pour montrer qu'une propriété P_0 est *toujours vraie* lors de l'exécution d'un programme, on peut être amené à montrer que $\bigwedge_{i\geqslant 0} P_i$ est un invariant de ce programme. Vis-à-vis d'une instruction s, la formule à démontrer est alors :

$$\begin{array}{rl} & \bigwedge P_i \Rightarrow wp\,(s, \bigwedge P_i) \\ \equiv & \bigwedge P_i \Rightarrow \bigwedge wp\,(s, P_i) \\ \equiv & \forall k \ : \ \bigwedge P_i \Rightarrow wp\,(s, P_k) \end{array}$$

Or, dans ce cas d'invariants conjonctifs de taille souvent très importante, l'instruction s ne modifie qu'un sous-ensemble limité des variables des prédicats P_k. Par conséquent, il apparaît un nombre non négligeable de tautologies du type $\bigwedge P_i \Rightarrow P_k$.

Cas des invariants implicatifs : une autre manière d'établir qu'une propriété Q est *toujours vraie* sur un programme est de montrer que $P \Rightarrow Q$ et P sont des invariants de ce programme. Pour une instruction s donnée, il faut alors prouver

$$(P \Rightarrow Q) \Rightarrow wp\,(s, P \Rightarrow Q)$$

c'est-à-dire

$$(P \Rightarrow Q) \Rightarrow (wp\,(s, P) \Rightarrow wp\,(s, Q))$$

Si l'affectation s considérée ne concerne que des variables à droite de l'implication, la formule se simplifie en :

$$(P \Rightarrow Q) \Rightarrow (P \Rightarrow wp\,(s, Q))$$

De la même manière, si l'affectation s ne modifie que des variables apparaissant à gauche de l'implication, la formule à vérifier est de la forme :

$$(P \Rightarrow Q) \Rightarrow (wp\,(s, P) \Rightarrow Q)$$

De telles simplifications se généralisent au cas où l'implication considérée fait partie d'une conjonction. On utilise alors les équivalences suivantes :

$$\begin{array}{rcl} ((P \Rightarrow Q) \wedge R) \Rightarrow (P \Rightarrow wp\,(s, Q)) & \equiv & (P \wedge R) \Rightarrow (Q \Rightarrow wp\,(s, Q)) \\ & \text{et} & \\ ((P \Rightarrow Q) \wedge R) \Rightarrow (wp\,(s, P) \Rightarrow Q) & \equiv & (\neg Q \wedge R) \Rightarrow (wp\,(s, P) \Rightarrow P) \end{array}$$

Exemple : Dans l'exemple d'exclusion mutuelle avec tolérance aux fautes, considérons l'invariant :

$$WHEN\_TOK \ = \ \bigwedge_{i=1}^{3} (s_i\_tok \Rightarrow \neg s_i\_invst)$$

et la seizième instruction :

$$s_{16} \ : \ s_1\_invst \ := \ true \ \ \texttt{if} \ \ (\neg s_1\_tok \wedge \neg s_1\_excl)$$

Le prédicat à valider est alors

$$WHEN\_TOK \wedge (\neg s_1\_tok \wedge \neg s_1\_excl) \Rightarrow wp\,(s_{16}, WHEN\_TOK)$$

Comme les deux dernières conjonctions de $WHEN\_TOK$ ne référencent pas la variable s_1_invst affectée par l'instruction s_{16}, la formule à vérifier devient :

$$WHEN\_TOK \wedge (\neg s_1\_excl \wedge \neg s_1\_excl) \Rightarrow wp\,(s_{16}, s_1\_tok \Rightarrow \neg s_1\_invst)$$

Comme de plus s_1_invst n'apparaît qu'à droite de l'implication, une seconde simplification est possible et la contrainte devient :

$$\begin{array}{c} s_1\_tok \wedge \bigwedge_{i=2}^{3}(s_i\_tok \Rightarrow \neg s_i\_invst) \wedge (\neg s_1\_tok \wedge \neg s_1\_excl) \\ \Rightarrow \\ (\neg s_1\_invst \Rightarrow wp\,(s_{16}, \neg s_1\_invst)) \end{array}$$

Cette dernière formule est bien valide puisque la conjonction $s_1\_tok \wedge \neg s_1\_tok$ apparaît en partie gauche de l'implication.

8.2.4 Implantation

Nous précisons ici quelques points d'implantation liés à l'exploitation du démonstrateur OTTER. Nous abordons successivement le problème de l'arithmétique, du pré-traitement et de la généralisation des formules engendrées.

Problème de l'arithmétique : la description d'algorithmes en UNITY fait appel fréquemment à la notion de compteurs, représentés par des variables entières. Ces compteurs, mis à jour par les affectations, sont à la base de la plupart des gardes. Typiquement, une instruction est exécutable si la valeur d'un compteur se situe dans un intervalle donné. L'affectation

$$a := a + 1 \quad \texttt{if} \quad a < b$$

vérifie la propriété

$$\texttt{stable}\ a \leqslant b$$

La vérification de cette contrainte se fait par la preuve de la formule

$$a < b \quad (\wedge \quad a \leqslant b) \quad \Rightarrow \quad a + 1 \leqslant b$$

Ceci suppose donc un minimum de connaissances arithmétiques de la part du démonstrateur utilisé. Malheureusement, OTTER ne possède aucune notion intrinsèque d'addition ou de relations d'ordre. Il ne manipule ces fonctions qu'au sein de constantes numériques et ne dispose d'aucune arithmétique *symbolique* interne.

Il a donc été nécessaire de lui fournir une base d'axiomes lui permettant d'aborder les problèmes liés à la gestion des compteurs. Celle-ci se limite aux opérations d'addition et de soustraction, et aux relations d'ordre larges et strictes sur les nombres entiers, ce qui constitue une partie décidable de l'arithmétique. Ceci permet donc de ne pas nuire à la complétude théorique de OTTER.

Malheureusement, l'introduction de ces axiomes multiplie considérablement la complexité des démonstrations réalisées par OTTER. On assiste en effet à une explosion combinatoire du nombre de clauses engendrées par le démonstrateur dont la complétude n'est assurée que par un parcours en largeur de l'arbre de recherche. La multiplication des clauses rend ce parcours très coûteux et limite la profondeur des démonstrations accessibles en un temps raisonnable.

Prise en compte de particularités de programmation : certaines simplifications de formules sont étroitement liées à des choix ou à des nécessités de programmation, en particulier au niveau du calcul des *wp*. Par exemple, la réduction d'expressions numériques (plus efficace au niveau du Grammatech Synthesizer Generator qu'au niveau de Otter), la simplification de constantes booléennes (introduites notamment comme valeurs de fin de récursivité) ou encore la suppression de termes redondants (introduits à différents niveaux du calcul). Ces simplifications, très sommaires, ne sont pas liées à la sémantique des spécifications ou des programmes Unity.

Généralisation des formules : pour des raisons pratiques, une durée maximum est imposée à toute tentative de démonstration. Or le démonstrateur peut s'avérer capable de valider rapidement une formule à partir de certaines hypothèses alors qu'un temps beaucoup plus long lui est nécessaire avec des hypothèses plus fortes (donc une formule plus longue). Ce phénomène peut donc conduire à un échec apparent d'une validation à cause de l'expiration du délai.

Pour cette raison, si (et seulement si) la démonstration de $P \wedge Q \Rightarrow R$ a échoué, on peut essayer d'établir $P \Rightarrow R$ ou $Q \Rightarrow R$. En particulier, si une formule Q doit être prouvée stable pour une instruction s gardée par G, on est amené à prouver

$$Q \wedge G \Rightarrow wp\,(s, Q)$$

Si la garde n'apporte aucun renseignement utile vis-à-vis de la propriété étudiée, il est très intéressant de remplacer le problème par $Q \Rightarrow wp\,(Q)$ ce qui permet de tenir compte ensuite de la structure particulière de Q (disjonction, conjonction, implication).

Par exemple, dans le problème du parking, la vérification de la propriété :

$$\texttt{invariant}\ f_1 \geqslant 0 \wedge f_2 \geqslant 0 \wedge f_3 \geqslant 0$$

au niveau de l'instruction :

$$d_1, f_1 \ := \ d_1 + 1, f_1 + 1\ \texttt{if}\ d_1 < e_1$$

ne nécessite pas de considérer la garde $d_1 < e_1$.

De la même manière, on peut être amené (toujours en cas d'échec de la démonstration directe) à remplacer $P \vee Q$ par P (ou Q), ou encore $P \Rightarrow Q \vee R$ par $P \Rightarrow Q$ (ou $P \Rightarrow R$).

L'inconvénient majeur de cette généralisation des formules est qu'elle amène, dans le cas où une formule *n'est pas* valide, à considérer une multitude de problèmes dérivés qui nécessairement ne sont pas valides.

8.3 Exploitation d'un calculateur de formules *Presburger*

8.3.1 Présentation du calculateur Omega

Devant la difficulté de mettre en œuvre, au sein d'un démonstrateur automatique de théorèmes classique tel que Otter, les simplifications arithmétiques nécessaires à la vérification des formules, un autre type d'outil a été utilisé. Il s'agit du calculateur OMEGA qui n'est pas à proprement parler un démonstrateur automatique de théorèmes, mais qui se limite à la manipulation d'ensembles et de relations entre n-uplets d'entiers. Ces ensembles et relations sont définis par des formules en arithmétique *Presburger*, une classe de formules construites à partir de contraintes linéaires sur des variables entières.

Une telle formule est donc de la forme :

$$\begin{array}{ll} & \phi \wedge \psi \\ ou & \phi \vee \psi \\ ou & \neg\phi \\ ou & \exists\, v_1, v_2, \cdots, v_n : \phi \\ ou & \forall\, v_1, v_2, \cdots, v_n : \phi \\ ou & C \end{array}$$

où ϕ, ψ sont des formules Presburger et C est une contrainte de la forme $e_1, e_2, \cdots, e_n \;\; op \;\; e'_1, e'_2, \cdots, e'_n$ où op est un opérateur parmi $=, \neq, <, >, \leqslant, \geqslant$. Les e'_i sont des expressions linéaires des variables.

Un ensemble est donc défini de la manière suivante :

$$\{[u_1, u_2, \cdots, u_n] : \phi\};$$

où ϕ est une formule Presburger, et une relation est de la forme :

$$\{[u_1, u_2, \cdots, u_n] \rightarrow [v_1, v_2, \cdots, v_n] : \phi\};$$

OMEGA propose alors une série d'opérations ensemblistes et relationnelles via une interface textuelle.

Exemple :

$$\begin{array}{l} pairs := \{[x] : \exists\, k : x = 2k\}; \\ impairs := \{[x]\} \;-\; pairs; \\ double := \{[x] \rightarrow [2x]\}; \\ double(impairs) \;\; subset \;\; pairs; \end{array}$$

Les trois premières requêtes à OMEGA mettent à jour trois variables. La quatrième exprime que le double d'un nombre impair est un nombre pair (OMEGA calcule la réponse *TRUE*).

8.3.2 Principe de validation des spécifications

Une seconde interface a donc été ajoutée à l'éditeur syntaxique pour permettre une vérification des spécifications UNITY par le calculateur OMEGA. Le principe est de traduire les pré-conditions et post-conditions par des ensembles d'états et l'affectation gardée par une relation entre n-uplets.

La vérification de $\{P\}\, s\, \{Q\}$ va consister à évaluer la *plus forte post-condition* (sp) de P vis-à-vis de s et à vérifier que $sp\,(s, P) \Rightarrow Q$.

Pour une instruction s de la forme :

$$\begin{array}{lcll} v_1, v_2, \cdots, v_n & := & e_1^1, e_2^1, \cdots, e_n^1 & \texttt{if} \quad g_1 \\ & \sim & e_1^2, e_2^2, \cdots, e_n^2 & \texttt{if} \quad g_2 \\ & \cdots & \cdots & \texttt{if} \quad \cdots \\ & \sim & e_1^p, e_2^p, \cdots, e_n^p & \texttt{if} \quad g_p \end{array}$$

on a donc :

$$\{P\}\ s\ \{Q\}\ \ ssi\ \ \bigwedge_{i=1}^{p} (sp\,(\alpha_i, P \wedge g_i) \Rightarrow Q)\ \bigwedge\ (P \wedge \neg g_1 \wedge \cdots \wedge \neg g_p \Rightarrow Q)$$

où α_i dénote l'affectation $v_1, v_2, \cdots, v_n\ :=\ e_1^i, e_2^i, \cdots, e_n^i$.

8.3.3 Implantation

Un état est défini par le n-uplet des valeurs courantes des variables du programme. Au lieu de manipuler directement les prédicats eux-mêmes, on considère des ensembles d'états vérifiant ces prédicats. En ce qui concerne les affectations, elles sont considérées comme des fonctions définies sur l'ensemble des états. L'implication $sp\,(s, P) \Rightarrow Q$ est donc vue comme l'inclusion $\mathcal{S}\,(\mathcal{P}) \subset \mathcal{Q}$ où $\mathcal{S}$ est la fonction associée à l'instruction s et $\mathcal{P}$ et $\mathcal{Q}$ sont les ensembles d'états associés aux prédicats P et Q. Cette approche évite le calcul explicite de la sp et permet de ne calculer ni sp ni wp au niveau de l'éditeur.

La vérification d'une spécification de la forme $\{P\}\,s\,\{Q\}$ où s est l'instruction

$$\begin{array}{lcll} v_1, v_2, \cdots, v_n & := & e_1^1, e_2^1, \cdots, e_n^1 & \texttt{if} \quad g_1 \\ & \sim & e_1^2, e_2^2, \cdots, e_n^2 & \texttt{if} \quad g_2 \\ & \cdots & \cdots & \texttt{if} \quad \cdots \\ & \sim & e_1^p, e_2^p, \cdots, e_n^p & \texttt{if} \quad g_p \end{array}$$

engendre donc les requêtes suivantes à OMEGA :

$$\begin{array}{ll} pre := & \{[v_1, v_2, \cdots, v_n] : P\}; \\ post := & \{[v_1, v_2, \cdots, v_n] : Q\}; \\ s := & \{[v_1, v_2, \cdots, v_n] \rightarrow [e_1^1, e_2^1, \cdots, e_n^1] : g_1\} \\ union & \{[v_1, v_2, \cdots, v_n] \rightarrow [e_1^2, e_2^2, \cdots, e_n^2] : g_2\} \\ \cdots & \cdots \\ union & \{[v_1, v_2, \cdots, v_n] \rightarrow [e_1^p, e_2^p, \cdots, e_n^p] : g_p\} \\ union & \{[v_1, v_2, \cdots, v_n] \rightarrow [v_1, v_2, \cdots, v_n] : \neg g_1 \wedge \neg g_2 \wedge \cdots \wedge \neg g_p\}; \\ s(pre) & subset\ \ post; \end{array}$$

On remarquera que, la notion de variable booléenne n'existant pas dans les formules *Presburger*, le traitement des affectations booléennes est un peu particulier. Une affectation de la forme $p := q$ `if` r sera traduite par :

$$\begin{array}{ll} \{[p] \rightarrow [1] : q \wedge r\} & union \\ \{[p] \rightarrow [0] : \neg q \wedge r\} & union \\ \{[p] \rightarrow [p] : \neg r\} & \end{array}$$

8.4 Bibliographie

Les travaux présentés dans ce chapitre ont été exposés à [CHP95] et publiés dans [CHP96].

L'exemple de l'exclusion mutuelle de Peterson est tiré du manuel d'utilisation du UNITY Verifier de M. Kaltenbach [Kal94b]. Le programme et sa spécification sont dus à J. Misra [Mis90a]. Les propriétés non vérifiées étaient prouvées fausses par le UNITY Verifier et correspondent en fait à une démonstration (manuelle) erronée dans la présentation originale de [Mis90a]. Le problème du parking est décrit dans [AHV83] .

En ce qui concerne les outils utilisés, le GRAMMATECH SYNTHESIZER GENERATOR est décrit dans [RT89]. Le démonstrateur OTTER est présenté dans [McC90, McC91] et la procédure de décision OMEGA dans [Pug92, KMP+94].

Conclusion

Dans le contexte de la preuve de systèmes réactifs répartis, le but de ce travail est de faciliter l'utilisation de démonstrateurs automatiques. Pour cela, il nous semble nécessaire de prendre en compte certaines particularités de l'algorithmique répartie au niveau même du formalisme. En effet, l'utilisation brutale de démonstrateurs sur une description détaillée d'un algorithme réparti n'est pas en mesure aujourd'hui d'apporter beaucoup d'automatisme dans la construction de la preuve de correction. Elle se borne généralement à traduire le problème de départ d'un formalisme de preuve de programmes vers des modèles logiques plus simples et plus familiers.

Nous avons choisi de mener cette étude dans le cadre du formalisme UNITY. Celui-ci présente l'avantage de pouvoir exprimer des propriétés de correction, non seulement de sûreté, mais aussi de vivacité. Par son côté universel, il permet aussi de définir différentes notions de raffinement, utilisables de la spécification jusqu'à l'implantation détaillée d'un programme.

Le premier aspect de la répartition auquel nous nous sommes attachés est celui de la communication entre les processus d'un système réparti. Nous avons montré que l'utilisation d'une représentation abstraite de la communication, bien intégrée au modèle d'exécution de UNITY et à sa logique, permet de simplifier l'écriture et la preuve de programmes répartis.

Ce mécanisme abstrait de communication s'attache au seul problème de la communication proprement dite, en la dissociant notamment des aspects de synchronisation. Nous montrons en particulier que cette communication élémentaire ne transmet à un processus aucun des liens de causalité présents dans son environnement. L'écriture de nombreux algorithmes répartis repose alors sur une description explicite de ces liens de causalité, notamment à l'aide d'horloges.

L'autre aspect de la répartition que nous avons formalisé est l'étape finale d'un développement dans le formalisme UNITY, c'est-à-dire le problème du placement (*mapping*) sur une architecture donnée. Nous avons montré que ce mapping ne prenait généralement pas en compte une certaine forme de parallélisme propre aux architectures réparties. Nous avons donc proposé un opérateur de placement pour pallier cette difficulté, et nous avons montré qu'un lien de raffinement unissait cet opérateur à la composition par union de UNITY.

Enfin, nous avons présenté le développement d'un assistant de preuve. Celui-ci a été conçu avec l'obsession d'un automatisme complet du processus de preuve. Il est en cela cohérent avec notre démarche qui veut faire porter l'effort humain en amont de l'utilisation d'outils de démonstration.

L'étape suivante de ce travail serait maintenant de définir un mécanisme qui pourrait s'associer à l'observation pour représenter les liens de synchronisation que celle-ci ignore. Il semble qu'une abstraction du transfert des liens de causalité permettrait d'enrichir la relation d'observation et de se dispenser de l'utilisation explicite d'horloges.

Enfin, à plus long terme, la redéfinition de l'observation dans une approche plus axiomatique pourrait permettre de simplifier considérablement un raisonnement qui porte actuellement directement sur les exécutions des programmes et qui n'y gagne pas en clarté et en simplicité.

Bibliographie

Bibliographie

[AEdC90] E. Audureau, P. Enjalbert, et L. Fariñas del Cerro. *Logique temporelle, sémantique et validation de programmes parallèles*. Études et recherches en informatique. Masson, 1990.

[AHV83] F. André, D. Herman, et J.-P. Verjus. *Synchronisation de programmes parallèles*. Dunod Informatique, 1983.

[AL91] M. Abadi and L. Lamport. The existence of refinement mappings. *Theoretical Computer Science*, 82(2):253–284, May 1991.

[And92] F. Andersen. *A Theorem Prover for* Unity *in Higher Order Logic*. PhD thesis, Technical University of Denmark, 1992.

[APP93] F. Andersen, K.D. Petersen, and J.S. Pettersson. Program verification using HOL-Unity. In *HOL User's Group Workshop (HUG'93)*, pages 1–17, 1993.

[APP95] F. Andersen, K.D. Petersen, and J.S. Pettersson. A graphical tool for proving Unity progress. Technical report, Tele Danmark Research, 1995.

[AS85] B. Alpern and F.B. Schneider. Defining liveness. *Information Processing Letters*, 21(4):181–185, October 1985.

[Bac92] R.J.R. Back. Refinement of parallel and reactive programs. Technical report, Marktoberdorf Summer School on Programming Logics, 1992.

[BM88] R.S. Boyer and J.S. Moore. *A Computational Logic Handbook*, volume 23 of *Perspectives in Computing*. Academic Press, 1988.

[BM93] N. Brown and D. Méry. A proof environment for concurrent programs. In *International Symposium of Formal Methods Europe (FME'93)*, volume 670 of *Lecture Notes in Computer Science*, pages 196–215. Springer-Verlag, 1993.

[BM95] N. Brown and A. Mokkedem. On mechanizing proofs within a complete proof system for Unity. In *Algebraic Methodology and Software Technology*, Concordia University of Montréal, 1995.

[BP 91] BP Innovation Centre and Edinburgh Portable Compilers Ltd. *B-Tool Version 1.1, Reference Manual*, 1991.

[Bro94] N. Brown. Vérification et mise en œuvre distribuée des programmes UNITY. In L. Bougé, editor, *Actes des sixièmes Rencontres francophones du Parallélisme (RenPar'6)*, pages 259–262, France, juin 1994. École Normale Supérieure de Lyon.

[CFM+96a] M. Charpentier, M. Filali, P. Mauran, G. Padiou, and P. Quéinnec. Abstracting communication to reason about distributed algorithms. In Ö. Babaoğlu and K. Marzullo, editors, *Tenth International Workshop on Distributed Algorithms (WDAG'96)*, volume 1151 of *Lecture Notes in Computer Science*, pages 89–104. Springer-Verlag, October 1996.

[CFM+96b] M. Charpentier, M. Filali, P. Mauran, G. Padiou, et P. Quéinnec. Répartition par observation dans UNITY. Rapport de recherche 96-01-R, Institut de Recherche en Informatique de Toulouse, France, janvier 1996.

[CFM+97] M. Charpentier, M. Filali, P. Mauran, G. Padiou, and P. Quéinnec. Tailoring UNITY to distributed program design, 1997. Soumis à FMPPTA'98.

[Cha96] M. Charpentier. Un opérateur de composition de programmes UNITY adapté à la répartition. Rapport de recherche 96-38-R, Institut de Recherche en Informatique de Toulouse, France, octobre 1996.

[Cha97] M. Charpentier. A UNITY mapping operator for distributed programs. In J. Fitzgerald, C.B. Jones, and P. Lucas, editors, *Fourth International Symposium of Formal Methods Europe (FME'97)*, volume 1313 of *Lecture Notes in Computer Science*, pages 665–684. Springer-Verlag, September 1997.

[Che95] B. Chetali. Formal verification of concurrent programs using the Larch Prover. In TACAS'95 [TAC95], pages 174–186.

[Che96] B. Chetali. *Vérification Formelle des Systèmes Parallèles décrits en* UNITY *à l'aide d'un outil de Démonstration Automatique.* Thèse de doctorat, Université Henri Poincaré, France, mai 1996.

[CHP95] M. Charpentier, A. El Hadri, and G. Padiou. A UNITY-based algorithm design assistant. In TACAS'95 [TAC95], pages 131–145.

[CHP96] M. Charpentier, A. El Hadri, et G. Padiou. Preuve automatique dans un environnement de développement UNITY. *Technique et Science Informatiques (TSI)*, 15(1), janvier 1996.

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

[Col94] P. Collette. *Design of Compositional Proof Systems Based on Assumption-Commitment Specifications. Application to* UNITY. Thèse de doctorat, Faculté des Sciences Appliquées, Université Catholique de Louvain, juin 1994.

[Coq96] Projet Coq. *The Coq Proof Assistant (Version 6.0)*. ENS-INRIA, Rocquencourt, France, 1996.

[CP97] M. Charpentier and G. Padiou. Specification and verification of the ATMR protocol using UNITY. In D. Méry, editor, *International workshop on Formal Methods for Parallel Programming: Theory and Applications (FMPPTA'97)*, pages 26–36, University of Geneva, Switzerland, April 1997.

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

[GG91] S.V. Garland and J.V. Guttag. A guide to LP, the Larch Prover. Technical Report 82, Digital Systems Research Center, 130 Lytton Ave., Palo Alto, CA 94301, USA, 1991.

[GM93] M.J.C. Gordon and T.F. Melham. *Introduction to HOL : A Theorem Proving Environnement for Higher Order Logic*. Cambridge University Press, 1993.

[Gol92] D.M. Goldschlag. *Mechanically Verifying Concurrent Programs*. PhD thesis, University of Texas at Austin, May 1992.

[HC96] B. Heyd and P. Crégut. A modular coding of UNITY in Coq. In J. von Wright, J. Grundy, and J. Harrison, editors, *Theorem Proving in Higher Order Logic*, volume 1125 of *Lecture Notes in Computer Science*, pages 251–266. Springer-Verlag, August 1996.

[HK93] P. Herrmann and H. Krumm. Report on analysis and verification techniques. Technical Report 485/1993, Universität Dortmund, Computer Networks and Distributed Systems Research Group, D-44221 Dortmund, Germany, 1993.

[Hoa84] C.A.R. Hoare. *Coomunicating Sequential Processes*. Prentice-Hall International, 1984.

[Kal94a] M. Kaltenbach. Model-checking for UNITY. Technical Report TR-94-31, Department of Computer Sciences, University of Texas at Austin, December 1994.

[Kal94b] M. Kaltenbach. *Model Checking for* UNITY*: The UV System*. Department of Computer Sciences, University of Texas at Austin, May 1994. Revision 1.10.

[Kal94c] M. Kaltenbach. *The UV System: User Interface Manual.* Department of Computer Sciences, University of Texas at Austin, December 1994. Revision 1.13.

[Kal96] M. Kaltenbach. *Interactive Verification. Exploiting Program Design Knowledge: A Model-Checker for* UNITY. PhD thesis, University of Texas at Austin, December 1996.

[KMP+94] W. Kelly, V. Maslov, W. Pugh, E. Rosser, T. Shpeisman, and D. Wonnacott. *The Omega Calculator and Library, version 0.90.* Department of Computer Science and Institute for Advanced Computer Studies, University of Maryland, College Park, MD 20742, November 1994.

[Kna90] E. Knapp. *Refinement as a basis of Concurrent Program Design.* PhD thesis, University of Texas at Austin, 1990.

[Lad96] P. Ladagnous. Vers la vérification automatique de programmes UNITY. In *Actes des journées sur la Formalisation des Activités Concurrentes (FAC'96)*, pages 107–120, Toulouse, France, 1996. CERT-IRIT-LAAS.

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

[Lam94] L. Lamport. The Temporal Logic of Actions. *ACM Transactions on Programming Languages and Systems*, 16(3):872–923, May 1994.

[Ma94] Z. Manna and al. STeP : The Stanford Temporal Prover. Technical Report STAN-CS-TR-94-1518, Stanford University, Department of Computer Science, June 1994.

[Ma95] Z. Manna and al. *STeP : The Stanford Temporal Prover: User's manual*, November 1995. Educational Release, version 1.0.

[Mat87] F. Mattern. Algorithms for distributed termination detection. *Distributed Computing*, 2:161–175, 1987.

[McC90] W. McCune. OTTER 2.0 users guide. Technical Report ANL-90/9, Mathemetics and Computer Science Division, Argonne National Laboratory, Illinois, March 1990.

[McC91] W. McCune. What's new in OTTER 2.2. Technical Report ANL/MCS-TM-153, Mathemetics and Computer Science Division, Argonne National Laboratory, Illinois, July 1991.

[MGRV94] C. Morgan, P. Gardiner, K. Robinson, and T. Vickers. *On the Refinement Calculus.* FACIT. Springer-Verlag, 1994.

[Mis90a] J. Misra. A family of 2-process mutual exclusion algorithms. In Notes on Unity [Mis90c]. Note 13.

[Mis90b] J. Misra. The importance of ensuring. In Notes on Unity [Mis90c]. Note 11.

[Mis90c] J. Misra. *Notes on Unity.* Department of Computer Science, University of Texas at Austin, 1990.

[Mis90d] J. Misra. Soundness of the substitution axiom. In Notes on Unity [Mis90c]. Note 14.

[Mis94a] J. Misra. Asynchronous composition of programs. In "New Unity" [Mis94c], chapitre 5.

[Mis94b] J. Misra. Closures properties. In "New Unity" [Mis94c], chapitre 6.

[Mis94c] J. Misra. *A Logic for Concurrent Programming.* Technical Report ("New Unity"), Department of Computer Science, University of Texas at Austin, 1994.

[Mis94d] J. Misra. Progress. In "New Unity" [Mis94c], chapitre 4.

[Mis94e] J. Misra. Safety properties. In "New Unity" [Mis94c], chapitre 3.

[MP92] Z. Manna and A. Pnueli. *The Temporal Logic of Reactive and Concurrent Systems: Specification.* Springer-Verlag, 1992.

[Pac92] J. Pachl. A simple proof of a completeness result for *leads-to* in the Unity logic. *Information Processing Letters*, 41:35–38, 1992.

[Pra94] I.S.W.B. Prasetya. Error in the Unity substitution rule for subscripted operators. *Formal Aspects of Computing*, 6:466–470, 1994.

[Pug92] W. Pugh. The Omega test : a fast and practical integer programming algorithm for dependence analysis. *Communications of the ACM*, August 1992.

[Rao95] J.R. Rao. *Extensions of the* Unity *Methodology: Compositionality, Fairness and Probability in Parallelism*, volume 908 of *Lecture Notes in Computer Science.* Springer-Verlag, 1995. Based on his PhD thesis, University of Texas at Austin, August 1992.

[Ray92] M. Raynal. *Synchronisation et état global dans les systèmes répartis*, chapitre Détection répartie de la terminaison, pages 163–172. Collection de la Direction des Etudes et Recherches d'Electricité de France. Éditions Eyrolles, 1992.

[RT89] T.W. Reps and T. Teitelbaum. *The Synthesizer Generator Reference Manual.* Texts and Monographs in Computer Science. Springer-Verlag, 1989.

[San91] B.A. Sanders. Eliminating the substitution axiom from UNITY logic. *Formal Aspects of Computing*, 3(2):189–205, April–June 1991.

[Sha93] A.U. Shankar. An introduction to assertional reasoning for concurrent systems. *ACM Computing Surveys*, 25(3):225–262, September 1993.

[Sin93] A.K. Singh. Program refinement in fair transition systems. *Acta Informatica*, 30:503–535, May 1993.

[TAC95] *Workshop on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'95)*, Aarhus, Denmark, May 1995. BRICS Notes Series NS-95-2.

[Udi95] R.T. Udink. *Program Refinement in* UNITY*-like Environments*. PhD thesis, Utrecht University, September 1995.

[UK93] R.T. Udink and J.N. Kok. On the relation between UNITY properties and sequences of states. In J.W de Bakker, W.P. de Roever, and G. Rozenberg, editors, *Semantics: Foundations and Applications*, volume 666 of *Lecture Notes in Computer Science*, pages 594–608. Springer-Verlag, 1993.

[WN95] G. Winskel and M. Nielsen. *Handbook of Logic in Computer Science*, volume 4, chapitre Models for concurrency, pages 1–148. Clarendon Press, Oxford, 1995.

[WSL93] M. Weber, M. Simons, and C. Lafontaine. *The Generic Development Language Deva: Presentation and Case Studies*, volume 738 of *Lecture Notes in Computer Science*. Springer-Verlag, 1993.